Diplomarbeit

Entwurf einer Abbildung von OCL-Constraints in Formeln einer dynamischen Logik für JavaCard

von

Uwe Keller

Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe

Betreuer: Dr. Bernhard Beckert, Prof. Dr. Peter H. Schmitt

Contributions
to
Formal Software Engineering

2. April 2002

Inhalt der Arbeit

Im Rahmen dieser Arbeit wurde eine Methodik entwickelt, mit der eine Übersetzung *beliebiger* Constraints und Ausdrücke der Object Constraint Language (OCL) in semantisch gleichbedeutende Formeln und Terme *beliebiger* Logiksprachen, welche Erweiterungen einer klassischen prädikatenlogischen Sprache darstellen, realisiert werden kann.

Das primäre Ziel war nicht, eine weitere Übersetzung von OCL in eine formale Sprache mit präziser Semantik anzugeben, um die formale Semantik von OCL festzulegen, sondern ein Verfahren zu entwickeln und zu implementieren, welches beliebigen Anwendungen das logische Schließen über UML/OCL-Modellen ermöglicht.

Diese Methodik wurde als Rahmenwerk gestaltet und ist flexibel an die Bedürfnisse unterschiedlichster Anwendungen anpaßbar, die darauf abzielen, in irgendeiner Weise UML/OCL-Modelle deduktiv zu verarbeiten. Besonderes Augenmerk wurde dabei auf Techniken gelegt, welche konkreten Anwendungen Möglichkeiten zur Optimierung der Übersetzung gemäß ihrer speziellen Vorstellungen bieten.

Der gesamte Entwurfs-, Anpassungs- und Optimierungsprozeß einer *anwendungsspezifischen* Übersetzung von OCL in eine konkrete formale Methode unter Verwendung des allgemeinen Rahmenwerks wurde am Beispiel des KeY-Projekts vollständig durchgeführt.

Die Ergebnisse der Arbeit sind schließlich zusammen mit einer Testumgebung für die Entwicklung einer anwendungsspezifischen Übersetzung von OCL in einer umfangreichen Implementierung umgesetzt worden und im Internet allgemein verfügbar.

Eine wesentlich kürzere Darstellung des Kerns der hier entwickelten Methodik [BKS01], viele weitere interessante Informationen über das KeY-Projekt, sowie Demoversionen des KeY-Systems und der Übersetzung an sich, finden sich im Internet unter der Adresse `i12www.ira.uka.de/~key`.

Danksagung

Es gibt natürlich viele Menschen, denen ich zu Dank verpflichtet bin, da sie durch ihre Hilfe ermöglicht haben, daß diese Arbeit in der Form entstehen konnte, die sie heute angenommen hat. Deshalb möchte ich diese Stelle nutzen, um all diesen lieben Menschen meinen herzlichen Dank auszusprechen, denn durch sie habe ich sehr viele Dinge gelernt.

Zunächst möchte ich *allen* – wissenschaftlichen *und* studentischen – Mitarbeitern des KeY-Projekts bedanken. Die Arbeit mit Euch ist für mich ein echtes Vergnügen, an dem ich noch lange meine Freude haben werde.

Einen wesentlichen Anteil an wichtigen Entscheidungen, die ich im Verlaufe dieser Arbeit getroffen habe, hat mit Sicherheit Thomas Baar, der mir immer bereitwillig zugehört hat und sich niemals scheute, mit kritischen Anmerkungen meine Arbeit zu hinterfragen, und mir so desöfteren neue Wege aufzeigte. Ich erinnere mich sehr gerne an die Erstellung des ersten Prototypen des KeY-Systems, bei dem Du unglaublich viel Engagement bewiesen hast. Die Diskussionen in dieser Zeit waren sehr wertvoll für mich. Und nicht zuletzt hast Du ganz uneigennützig einige Stunden Deiner Zeit geopfert, um mit mir an der Endversion des aus dieser Arbeit hervorgegangenen Papiers zu feilen. Ein ganz herzliches Dankeschön an Dich!

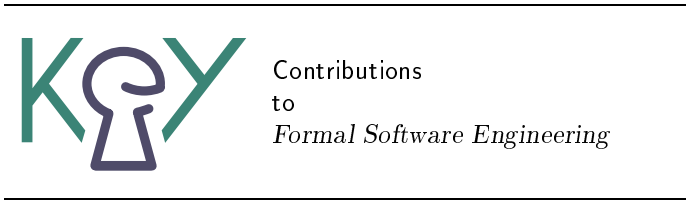
Mit zahlreichen Anregungen und Anmerkungen hat auch Herr Prof. Dr. Peter H. Schmitt wesentliches dazu beigetragen, daß ich die von mir in dieser Arbeit entwickelte Methode heute so verstehe, wie es in den folgenden Kapiteln präzise dargestellt wird. Der vorliegende Text wurde sicherlich stark von seiner Arbeit über die Semantik von OCL [Sch01b] inspiriert, einer Arbeit, die in meinen Augen zu dieser Zeit eine der interessantesten und schönsten Arbeiten zu der formalen Semantik von OCL war. Lieber Herr Schmitt, Sie haben sich unermüdlich bemüht, mich in schwierigen Phasen im Verlaufe dieser Arbeit immer wieder zu motivieren. Vielen Dank!

Mein ganz besonderer Dank gilt vor allem Bernhard Beckert, für seine unendliche Geduld, die er bei der Betreuung dieser Arbeit an den Tag legte. Er hatte immer ein offenes Ohr für meine Probleme – seien sie arbeitstechnischer oder auch privater Natur – und verstand es, mir mit Rat und Tat – und vielen sehr schlauen Ideen! – zur Seite zu stehen. Du hast mir auch schweren Zeiten Verständnis entgegengebracht und mir den Rücken freigehalten. Ich war sicherlich kein einfacher Schützling und habe meine Rolle im ganzen Prozeß erst sehr spät verstanden. Dafür möchte ich mich entschuldigen. Ich habe durch Dich auf jeden Fall sehr viel dazu gelernt – auch in Dingen, die über den Inhalt dieser Arbeit hinausgehen.

Lieber Bernhard, ich verdanke Dir sehr, sehr viel. Umsomehr tut es mir schrecklich leid, daß diese Arbeit ein solches Ausmaß angenommen hat, durch das Du Dich schließlich nocheinmal ,,durchquälen" mußt ;-). Aber wie ich Dir schon einmal erklärt habe, in gewisser Weise mußte ich mir etwas von der Seele schreiben. Ich hoffe, daß Du mit dem Ergebnis zufrieden bist.

Schließlich möchte ich mich bei meinen beiden Betreuern – Bernhard Beckert und Peter. H. Schmitt – für ein Erlebnis herzlich bedanken, welches eine ganz besondere

Bedeutung für mich hat: Die Möglichkeit, mit Ihnen zusammen aus dieser Arbeit heraus ein Papier zu verfassen und schließlich bei einer Konferenz einzureichen. Ich habe viele schöne Eindrücke während der gemeinsamen Arbeit an diesem Papier gewonnen. Dieses besondere Erlebnis gehört sicherlich zu den schönsten Erfahrungen, die ich in meinem Studium machen durfte. Es hat mir neues Selbstbewußtsein nach einer für mich etwas finsteren Zeit und sicherlich viel Motivation für die Zukunft gegeben. Tausend Dank!



Erklärung

Ich erkläre hiermit, diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet zu haben.

Wiesloch, den 2. April 2002

Uwe Keller

Widmung

„Begegne den Menschen so, daß sie es als Glück empfinden, Dir begegnet zu sein."
– Mutter Theresa

Meinen Eltern, Brundhilde und Walter Keller, für ihre Unterstützung und Liebe, die sie mir immer haben zuteil werden lassen. Ohne Euch hätte ich mein Studium niemals absolvieren und damit die Chancen wahrnehmen können, die Euch leider niemals offenstanden. Ich hoffe, daß ich Euch mit dieser Arbeit eine kleine Freude bereiten kann.

Meiner Familie und meinen Freunden, den wichtigsten Personen in meinem Leben, die wesentlichen Anteil daran haben, daß ich derjenige wurde, der ich heute bin, die mich in vielen Situationen beeinflußt, unterstützt und vorangebracht haben: Martina, Patricia, Raffael und Peter, Sebastian, Steffi, Andreas, Céline, Thomas, Obsti, Harald, Norbert, Jessica, Ursel, Manfred, sowie Karl und Helga Breyer. Wir alle mögen verschieden voneinander sein – der eine etwas mehr, der andere etwas weniger – doch ein Teil von Euch allen ist in mir und dafür bin ich Euch sehr dankbar. Gerade diese Vielfalt und Buntheit hat die Entwicklung meiner Persönlichkeit stark beeinflußt und geprägt und mir immens dabei geholfen, herauszufinden, was mir wichtig und erstrebenswert im Leben erscheint und wer ich eigentlich bin.

Meinen Lehrern, die mir den Spaß am Lernen und an der Erkenntnis vermitteln konnten und die mich in vielen Situationen uneigennützig förderten, insbesondere Peter Beck, Dr. Bernhard Beckert, Prof. Dr. Peter H. Schmitt und Prof. Dr. Wolfram Menzel.

Meinen musikalischen Mitstreitern in unserer Band, mit denen ich viele schöne Stunden erlebt habe und mit denen ich einige Auftritte bestritt, die mir lange in Erinnerung bleiben werden: Simon, Christian, Michaela, Jens und Daniel.

Und schließlich meinen Komilitonen, mit denen ich meinen Studienanfang erlebte, die mir das Studium nicht ganz so anonym erscheinen ließen und mit denen ich eine sehr schöne Zeit durchleben durfte: Alex, Thomas, Frank, Jürgen, Rainer, Bernhard und Stefan.

Ihr alle habt mich sehr stark bereichert. Ich hoffe und wünsche mir, daß ich diesen Reichtum und all meine positiven Erfahrungen irgendwann an andere weitergeben kann. Ihr alle liegt mir sehr am Herzen. Danke!

Inhaltsverzeichnis

Kapitel 1

Einführung und Umfeld der Arbeit

Wir wollen in diesem Kapitel zunächst das Umfeld, in dem diese Arbeit entstanden ist, kurz darlegen, um zu motivieren, warum diese Arbeit überhaupt angefertigt wurde. Anschließend formulieren wir die Aufgabenstellung, welche als Ausgangspunkt dieser Arbeit diente. Schließlich wenden wir uns in knapper Weise verwandten Arbeiten zu.

1.1 Softwareentwicklung und formale Methoden

In der letzten Dekade hat sicherlich ein Paradigma die Welt des Software-Engineering entscheidend geprägt und ist in einem unaufhaltsamen Siegeszug in die industrielle Praxis eingegangen: Die Objektorientierung.

Einer der Hauptfaktoren, die für die rasche und weite Verbreitung und Anwendung dieses Paradigmas in der Praxis gesorgt haben, mag die fortwährend steigende Komplexität der Softwaresysteme sein, die zu modellieren und implementieren sind, sowie die vorteilhafte Eigenschaft dieses Paradigmas, Modellierungs- und Implementierungssprachen hervorbringen zu können, welche es dem Software-Ingenieur recht erfolgreich erlauben, in natürlicher und vertrauter Weise Softwaresysteme zu modellieren und somit mit der wachsenden Komplexität solcher Systeme umzugehen[1].

Mit der steigenden Komplexität der zu entwickelnden Systeme erhöht sich jedoch auch die Möglichkeit von Fehlern in der Entwicklung und somit der Testaufwand erheblich. Werden Fehler aus frühen Phasen erst sehr spät in der Entwicklung eines solchen Systems gefunden, so sind die entstehenden Kosten immens. Für bestimmte sicherheitskritische Bereiche wird ein umfassendes Testen damit ähnlich teuer, wie der formale Nachweis der Korrektheit des Softwaresystems über eine formale Verifikation und der Einsatz von formalen Methoden könnte damit sogar aus finanzieller Sicht für die Praxis interessant werden.

[1] Die Ansicht, die Objektorientierung sei das *universelle* Allheilmittel im Software-Engineering, verliert in jüngerer Zeit immer mehr Anhänger. Immer mehr Software-Ingenieure sind der Meinung, daß die Objektorientierung zwar ein wichtiger Schritt in Richtung der Beherrschbarkeit komplexer Softwaresysteme sei, jedoch alleine nicht weit genug reiche. In jüngster Zeit rücken daher immer neue Paradigmen ins Rampenlicht der Software-Welt, welche in gewisser Weise Erweiterungen und Fortentwicklungen der objektorientierten Idee sind: Muster, Komponentensysteme, Aspektorientierte Programmierung, Agentensysteme und einige mehr.

Umso merkwürdiger mutet an, daß in der industriellen Praxis äußerst selten formale Methoden im Software-Engineering – d.h. Spezifikation und Verifikation von Softwaresystemen – eingesetzt werden ([DR96]). Doch welche Gründe führen dazu?

Eine kritische Analyse von wissenschaftlicher Seite aus in [ABB+00] hat vor allem die folgenden Aspekte als entscheidende Einflußfaktoren benannt:

- Die Anwendung formaler Methoden ist bisher *nicht* in den iterativen und inkrementellen Software-Entwicklungsprozeß eingebunden, der in der Industrie heute Anwendung findet.
- Die heutigen Werkzeuge zur formalen Spezifikation und Verifikation von Softwaresystemen sind *nicht* in die entsprechenden, diesen Entwicklungsprozeß unterstützenden Werkzeuge integriert. Tatsächlich gilt sogar, daß die Zielsprache des Verifikationswerkzeugs, in der die zu verifizierenden Programme verfaßt sein müssen, in fast keinem Falle einer ,,realen" Programmiersprache entspricht, welche in der industriellen Praxis eingesetzt wird.
- Von den Benutzern des Verifikationswerkzeugs wird die Kenntnis und Vertrautheit der Syntax und Semantik von einer oder mehreren komplexen formalen Sprachen erwartet. Darüber hinaus wird oftmals eingehendes Wissen über den benutzte logischen Kalküle, sowie Beweisstrategien des Werkzeugs benötigt.

Man beachte, daß die oben dargestellte Situation nicht etwa dadurch entsteht, daß es keine Methoden gäbe, die diese Aufgaben prinzipiell lösen könnten. In einigen größeren industriellen Projekten wurde von wissenschaftlicher Seite aus sogar konkret demonstriert, daß die vorhandenen Methoden prinzipiell die *gegebenen industriellen Fragestellungen und Probleme* lösen können!

Diese Mißstände anzugehen und zu beseitigen und somit den Einsatz formaler Methoden im industriellen Software-Prozeß wirklich zu ermöglichen, danach strebt das KeY-Projekt, das in Abschnitt 1.2 kurz vorgestellt wird.

1.1.1 Der Standard für Modellierungssprachen – UML/OCL

1.1.1.1 Unified Modeling Language (UML)

Seit der Einführung der Idee der Objektorientierung gibt es nun eine Fülle verschiedener Sprachen, die das neue, und vermeintlich ,,universelle" Prinzip in den Softwareentwicklungsprozeß einzubinden versuchen: Für die Implementierungsphase gibt es eine Reihe verschiedener Programmiersprachen, welche das Prinzip in unterschiedlichem Ausmaß in geeigneten Programmkonstrukten umsetzen, beispielweise C++, Smalltalk, Oberon, Eiffel, Java und viele andere mehr, es wurden jedoch auch für andere Phasen des Entwicklungsprozesses entsprechende Sprachen entwickelt. So gab es für die Analyse- und Entwurfsphasen gleichermaßen einige unterschiedliche *graphische* Sprachen, die sehr gut dazu geeignet waren, objektorientierte Analyse- und Entwurfsmodelle zu beschreiben. Die drei wichtigsten Notationen wurden wohl von Rumbaugh et. al. (OMT) [JR91], Booch [Boo91] und Jacobsen [Jac92] eingeführt.

Jede dieser graphischen Modellierungssprachen hatte ihre Stärken und Schwächen und ein langanhaltender Glaubenskrieg zwischen den einzelnen Lagern verhinderte schließlich, daß eine dieser Methoden einen wirklichen Durchbruch in der Praxis erleben konnte.

Das führte nun dazu, daß sich die drei „Gurus"[2] der einzelnen Lager entschlossen, die drei Methoden zu einer einzigen, einheitlichen Modellierungssprache zu verbinden, welche die Vorzüge der einzelnen Methoden in sich vereinen sollte: Die Unified Modelling Language[3].

Diese umfangreiche graphische Modellierungssprache konnte sich schließlich in der Praxis durchsetzen und erfreut sich heute als Standardnotation für die Modellierung objektorientierter Systeme großer und stetig wachsender Beliebtheit, da eine einheitliche Standardsprache zur Modellierung gewissermaßen als einheitliche Sprache in der Softwaremodellierung die Kommunikation zwischen Software-Ingenieuren erleichtert und verbessert und die Entwicklung entsprechender CASE-Werkzeuge wesentlich vorangetrieben hat.

Die UML ist ein Familie von ungefähr acht Diagrammarten, die jeweils unterschiedlich Facetten eines beliebigen – nicht notwendigerweise softwaretechnischen – Systems modellieren können: Klassendiagramme beschreiben beispielsweise die statische Struktur des Systems, sowie die Beziehungen, die zwischen den einzelnen modellierten Größen herrschen; Sequenzen- und Kollaborationsdiagramme stellen das dynamische Verhalten des Systems anhand des Informations- bzw. Nachrichtenfluß in einer beispielhaften Situation dar; Zustandsdiagramme beschreiben den Aspekt das dynamische Verhalten, welcher durch klasseninterne Zustandswechsel hervorgerufen wird; Use-Case-Diagramme charakterisieren die Anbindung des Systems an dies Außenwelt durch Beschreibung typischer Anwendungsfälle und deren Beziehungen zueinander, um nur einige dieser Diagrammtypen zu nennen.

Wir wollen hier nicht detaillierter auf die UML eingehen, da sie mittlerweile zum Standardkanon einer Softwaretechnik-Vorlesung gehört. Eine umfassende Darstellung findet sich beispielsweise in der Referenz [JR99] oder der UML-Spezifikation [Obj99b].

Anfänglich wurde die UML ausschließlich in präzisem Englisch und UML selbst[4] beschrieben und kann somit als semi-formale Beschreibungsprache betrachtet werden. Doch gerade seit den letzten Jahren gibt es starke Bemühungen, UML mit einer formale Semantik zu versehen und so wird der kommende Standard UML 2.0 wahrscheinlich mit einem soliden semantischen Unterbau aufwarten können.

1.1.1.2 Object Constraint Language (OCL)

Was augenblicklich hingegen dem allgemeingebildeten Informatiker eher unbekannt sein dürfte, ist das die UML als integralen Bestandteil eine formale, textuelle Sprache zur Formulierung von komplexeren Einschränkungen an ein UML-Modell, umfaßt: Die Object Constraint Language (OCL).

OCL wurde dazu entwickelt, einem Modellierer eines Systems die Beschreibung von Nuancen und subtilen Details zu ermöglichen, die durch die rein graphische Notationen in der UML nicht darzustellen sind. Beispielsweise herrschen in einer modellierten Miniwelt meist sehr viel komplexere Einschränkungen an die konkreten Beziehungen zwischen den einzelnen Entitäten, durch deren Zusammenwirken das System überhaupt

[2] Die sogenannten drei Amigos!

[3] Die *Unified Modelling Language*, wurde anfänglich auch als *Unified Method* bezeichnet. Wie der Name schon sagt, handelt es sich um eine Modellierungs*sprache*, mit der Sachverhalte einer realen oder abstrakten Welt in Form eines graphischen Modells wiedergegeben werden können. Sie umfaßt jedoch *kein* Vorgehensmodell, das - beispielsweise in Form von empirischen Richtlinien - beschreibt, wie die Methode anzuwenden ist! Verschiedene Vorgehensmodelle zur UML werden heute jedoch auch in der Literatur angeboten, so zum Beispiel der *Unified Process* der Firma RATIONAL.

[4] Durch das UML-Metamodell.

erst ensteht, als es die Multiplizitäten und Assoziationen zwischen den Klassen in einem zugehörigen Klassendiagramm wiederzugeben vermögen.

Zu diesem Zweck baut OCL auf mathematischen Konzepten wie Mengen, Multimengen und Sequenzen, Typen, Operatoren auf diesen Typen[5] und einem hierarchischen Typsystem auf. Gleichwohl sollte OCL – im Gegensatz zu vielen anderen mathematischen Spezifikationssprachen – auch für den allgemeingebildeten Software-Ingenieur bzw. Programmierer leicht erlernbar und anwendbar sein, weshalb die Notation von OCL-Ausdrücken eher an natürliche Sprachen erinnern sollte und komplexere Konzepte – wie zum Beispiel ineinander geschachtelte Kollektionen – beim Entwurf der Sprache vermieden wurden[6].

Wie Kapitel 2 dieser Arbeit noch deutlich gemacht wird, kann man OCL im Groben als eine Art ,,objektorientierte Prädikatenlogik" ansehen, die bestimmten Einschränkungen unterliegt. Die Signatur dieser Prädikatenlogik ergibt sich im wesentlichen aus dem UML-Modell $\mathcal{D}$, über dem die OCL-Ausdrücke bzw. OCL-Constraints formuliert werden, sowie den in OCL fest eingebauten Typen und den zugehörigen Operatoren.

Mithilfe von OCL lassen sich unter anderem auch Invarianten und Vor- und Nachbedingungen von Methodenaufrufen formulieren. Die Integration von OCL in UML erzeugt also eine Sprache, die zur formalen Spezifikation von Softwaresystemen herangezogen werden kann und die aufgrund der Entwurfsprinzipien der UML – als graphische Modellierungssprache – und der OCL – als textuelle Sprache, die zwischen streng mathematischen Notationen und umgangssprachlichen Beschreibungen zu vermitteln versucht – für den allgemeingebildeten Software-Ingenieur leicht zu erlernen und anzuwenden sein sollte.

OCL an sich ist daher keine allzu komplexe Sprache. Wir ersparen uns deshalb eine ausführlichere Einführung und Behandlung von OCL an dieser Stelle und verweisen auf die OCL-Spezifikation [Obj99a] oder das Buch [WK99]. Die weitere Darstellung ist so angelegt, daß die Bedeutung der einzelnen OCL-Konstrukte bei der Vorstellung der Übersetzung jeweils kurz erläutert wird und daher kein besonderes Vorwissen über OCL für das Verstehen dieser Arbeit notwendig ist.

Eine der ersten Anwendungen von OCL war übrigens die Spezifikation der UML selbst: Die UML-Spezifikation in der Version 1.0 basierte im wesentlichen auf natürlichsprachlichen Beschreibungen in ,,präzisem" Englisch. Man entdeckte jedoch alsbald Inkonsistenzen in dieser natürlichsprachlichen Beschreibung, welche auf Mehrdeutigkeiten zurückzuführen waren, wie sie oft in natürlichsprachlichen Beschreibungen zu finden sind, und entschloß sich daher, in der darauffolgenden Version UML 1.1 des Standards eine einfache formale Sprache bei Beschreibung des UML-Metamodells zu verwenden, um solchen Mehrdeutigkeiten vorzubeugen. Somit wurde OCL zum integralen Bestandteil des UML-Standards, der sogleich seine erste Anwendung im Standard selbst fand.

[5] Im OCL Jargon werden diese Operatoren als *Eigenschaften (Properties)* bezeichnet.

[6] Geschachtelte Kollektionen werden durch einen als *Flattening* bezeichneten Prozeß in OCL automatisch in eine ungeschachtelte Kollektion umgewandelt, weshalb es de facto keine geschachtelten Kollektionen in OCL gibt. Dieser Ansatz führt an manchen Stellen zu semantischen Mehrdeutigkeiten und ist damit nicht immer von Vorteil!

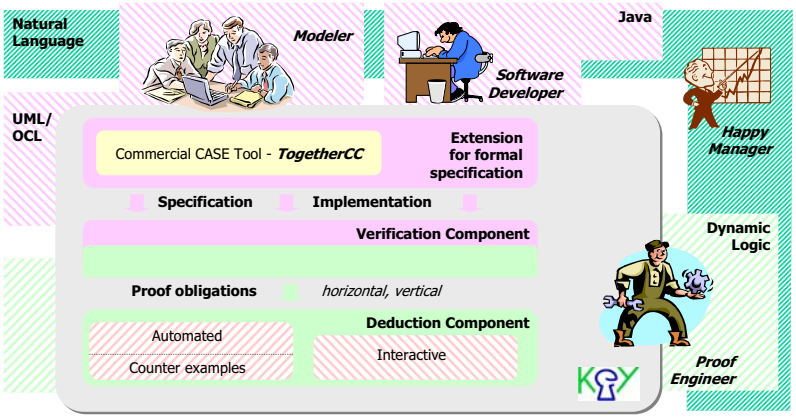


Abbildung 1.1: Das KeY-System.

1.2 Das KeY-Projekt

Wir wollen nun in wenigen Sätzen das KeY-Projekt vorstellen. Eine ausführlichere Darstellung findet sich in [ABB+00, ABB+02].

Das KeY-Projekt hat sich als ehrgeiziges Ziel gesetzt, die oben angegebenen Hinderungsgründe für die Anwendung formaler Methoden in der industriellen Praxis anzugehen und zu beseitigen.

Dementsprechend ist eines der Hauptanliegen des Projekts, durch Erweiterung eines kommerziellen CASE-Tools (TogetherCC[7]) eine Software-Entwicklungsumgebung – das KeY-System – zu entwerfen und zu implementieren, welches die formale Softwareentwicklung mit Hinblick auf die oben genannten Anforderungen unterstützt.

Die Architektur des KeY-Systems ist in Abbildung 1.1 dargestellt und soll nun kurz erläutert werden.

Ein kommerzielles CASE-Tool wird um Dienste zur formalen Spezifikation und Verifikation erweitert. Der Modellierer beschreibt die zu realisierende Software über ein UML-Modell mit OCL-Annotationen. Das im Projekt gewählte CASE-Tool erlaubt zudem die Implementierung des modellierten Systems in der Programmiersprache Java. Eine Verifikationskomponente stellt nun Dienste zur deduktiven Verarbeitung dieser formalen Beschreibungen dar und ist verantwortlich für die Erzeugung von sogenannten Beweisverpflichtungen (*Proof obligations*), d.h. Formeln der dynamischen Logik **DL** [Bec01], welche bestimmte Eigenschaften der Spezifikation (*horizontale Verifikation*) oder die Beziehung zwischen Spezifikation und Implementierung (*vertikale Verifikation*) formal charakterisieren. Schließlich wird eine interaktive, teilautomatisierte

[7] Together Control Center, siehe auch `www.togethersoft.com`

Deduktionskomponente eingesetzt, um die erzeugten Beweisverpflichtungen formal – und in Kooperation mit einem menschlichen Beweiser (*Proof engineer*) – zu beweisen oder – durch eine automatische Generierung eines geeigneten Gegenbeispiels – zu widerlegen.

1.3 Aufgabenstellung der Arbeit

Diese Arbeit hat nun gerade die Aufgabe, den in der Verifikationskomponente stattfindenden Übergang zwischen der Ebene der Spezifikation und der Ebene der Beweissystems zu ermöglichen. Dazu müssen prinzipiell beliebige UML/OCL-Modelle in eine äquivalente Beschreibung durch Formeln der dynamischen Logik **DL**[8] [Bec01] übersetzt werden.

Um den Rahmen der Arbeit nicht zu sprengen, beschränken wir uns auf die Behandlung von UML/OCL-Modellen, die ausschließlich Klassendiagramme umfassen.

Gegeben sei also ein Klassendiagramm $\mathcal{D}$, welches mit *beliebigen* OCL-Constraints $C_1, \ldots C_n$ annotiert ist.

Dann soll ein Verfahren entwickelt werden, welches das annotierte Klassendiagramm $\mathcal{D}$ in eine *äquivalente* Beschreibung durch eine geeignete Menge von **DL**-Formeln übersetzt. Besonderes Augenmerk liegt dabei auf der Übersetzung der einzelnen OCL-Constraints C_i $(i = 1, \ldots, n)$.

Das gesamte Verfahren soll im Rahmen des KeY-Systems implementiert werden.

1.4 Quell- und Zielsprache für diese Arbeit

Für den Entwurf einer solchen Übersetzung sind zunächst die betroffenen Sprachen, d.h. die Quellsprache und die Zielsprache der Abbildung zu untersuchen. Da die Übersetzung in einem noch zu präzisierenden Sinne korrekt sein soll, muß man sich insbesondere darüber Klarheit verschaffen, wie Beschreibungen in der Quell- und der Zielsprache inhaltlich zu interpretieren sind, d.h. wir müssen uns mit den formalen Semantiken der beiden Sprachen auseinandersetzen.

Wir wollen auch hier aus Platzgründen auf eine detaillierte Beschreibung dieser Sprachen verzichten und verweisen auf die Arbeiten [Sch01b] – für UML/OCL – und [Bec01] – für **DL**.

Bemerkung (Keine Panik). Die Ausführungen in den folgenden Kapiteln sollten in weiten Teilen auch für den Leser verständlich sein, der die oben genannten Arbeiten *nicht* kennt, da die Bedeutung der einzelnen OCL-Ausdrücke an den einzelnen Stellen immer angegeben sind und der Kern des Verfahrens im Prinzip auf klassischer Prädikatenlogik basiert, die allgemein bekannt sein sollte. □

1.5 Verwandte Arbeiten

Augenblicklich gibt es eine Vielzahl von Papieren, die sich der Entwicklung einer formalen Semantik für UML-Klassendiagramme annehmen und sich dafür einer Abbildung von UML-Klassendiagrammen in ein formale Sprache mit bekannter Semantik

[8] Wir bezeichnen im folgenden die *spezielle* Ausprägung einer dynamischen Logik, welche von Bernhard Beckert [Bec01] entwickelt wurde, durch **DL**.

bedienen, zum Beispiel CASL-LTL [RCA00], Z [Fra99], Object Z [KC99], die logische Sprache von PVS [Kri00], das Mathematical System Model (MSM) [BGH+98], EER Diagramme [GR98], Maude [AA00].

Die vorhandenen Beschreibungen zur aktuellen Version von OCL – das OMG Standard-Dokument [Obj99a], sowie das Buch [WK99] –, bieten keine formale Definition der Semantik von OCL. Stattdessen wurde die Semantik in präzisem Englisch angegeben, was jedoch in manchen Situationen zu Problemen geführt hat [HCH+98].

Seither gab es auch für OCL einige Arbeiten, die eine formale Semantik für OCL direkt definieren [Sch01b, RG98] bzw. für zukünftige Versionen von OCL festlegen wollen [BRI01].

Außerdem wurde mehrfach versucht, durch Abbildung von OCL in eine formale Sprache mit definierter Bedeutung eine formale Semantik für OCL selbst anzugeben, beispielweise die Larch Shared Language (LSL) [HHK98] oder die temporallogische Sprache BOTL [DKR00b, DKR00a].

Unsere Arbeit unterschiedet sich in ihrer vorliegenden Form jedoch von den bisher veröffentlichten Papieren durch einen wesentlich allgemeinere Ansatz:

Als Zielsprache verwenden wir weitestgehend eine *universelle* Logiksprache, anstelle eines spezialisierten Formalismus: Klassische Prädikatenlogik erster Stufe. Unser Verfahren stellt keine *spezialisierte* Abbildung von OCL in *eine* Zielsprache dar, sondern ein Rahmenwerk, welches für *viele* Zielsprachen geeignet sein sollte und an die speziellen Bedürfnissen einer konkreten Anwendung angepaßt werden kann.

Besonderes Augenmerk haben wir auf die Qualität der erzeugten Formeln gelegt, ein Aspekt der von keiner Arbeit zuvor betrachtet wurde. Wir zielen nicht auf die Definition einer formalen Semantik von OCL ab, sondern wollen vielmehr eine Erleichterung für eine breite deduktiven Nutzung bzw. Verarbeitung von UML/OCL-Modellen erreichen bzw. eine solche Verarbeitung ohne großen Entwicklungsaufwand für viele Anwendungen überhaupt erst ermöglichen.

Diese Arbeit nach unserem Wissen die bisher detaillierteste und genaueste Darstellung einer solchen Abbildung. Insbesondere untersuchen wir – im Gegensatz zu den uns bekannten Arbeiten zu Übersetzungen von OCL – den Korrektheitsbegriff für eine solche Abbildung (sowohl in beliebige Logiksprachen als auch in die **DL**) genauer und versuchen diesen abstrakten Begriff in einer formalen Definition zu fassen.

Soweit es uns bekannt ist, gibt es augenblicklich keine andere, allgemein verfügbare Implementierung einer solchen Abbildung.

Kapitel 2

Von OCL-Constraints zu logischen Formeln - eine erste Abbildung

In diesem Kapitel stellen wir eine Basisversion für eine Abbildung beliebiger OCL-Ausdrücke in Formeln einer dynamischen Logik für JAVA-CARD vor. Wir klären zunächst, welche besonderen Randbedingungen für eine solche Abbildung im allgemeinen bestehen, und präzisieren anschließend die Frage nach einem Korrektheitskriterium für derartige Abbildungen im Begriff *Informationserhaltung*. In Abschnitt 2.3 werden wir die Details der Transformation angeben. Abschließend verdeutlichen wir die Abbildung an einem umfangreicheren Anwendungsbeispiel.

2.1 Randbedingungen für die Abbildung

Betrachtet man die Problemstellung in einem allgemeinen Rahmen, so trifft man auf folgende Situation: Gegeben sei eine UML/OCL-Modellierung eines beliebigen Systems[1]. Die durch das Modell dargestellte Information soll nun innerhalb einer gewissen Anwendung genutzt und verarbeitet werden und muß dementsprechend durch eine Transformation des UML/OCL-Modells in eine für die Anwendung geeignete Zielsprache übersetzt werden; beispielsweise könnte man sich vorstellen, daß es sich bei dem UML/OCL-Modell um eine logische Modellierung einer Datenbasis handelt, die auf das (implementierte) Datenmodell eines konkret eingesetzten Datenbanksystems (relationales Modell/SQL, objektorientiertes Modell/OQL usw.) abgebildet werden muß, oder man möchte das modellierte System anhand der erstellten Beschreibung simulieren, um so die Modellierung zu evaluieren und mögliche Fehler (schon vor einer Implementierung) zu finden [RG00].

Als Randbedingungen an eine Abbildung von UML/OCL-Modellen auf dieser Ebene lassen sich folgende Eigenschaften identifizieren:

[1] Das modellierte System ist im allgemeinen *kein* Software-System, es kann sich sogar um ein nicht-technisches System handeln!

- **Rekursives Abbildungsverfahren**
  OCL selbst ist eine formale Sprache und in [Obj99a] unter anderem durch eine kontextfreie Grammatik beschrieben. In der formalen Definition der Menge $OCLExp_{\mathcal{D}}$ der OCL-Ausdrücke (zu einem gegebenen UML-Diagramm $\mathcal{D}$) in [Sch01b] wird eine induktive Konstruktion für $OCLExp_{\mathcal{D}}$ angegeben. Man kann daher erwarten, daß die Abbildung rekursiv vorgeht, d.h. der Ausdruck (in der Zielsprache), der einem OCL-Ausdruck `e` zugeordnet wird, entsteht im wesentlichen aus den Ausdrücken, die durch die Übersetzung der OCL-Teilausdrücke e_i des OCL-Ausdrucks `e` gewonnen werden.

- **Korrektheit der Abbildung**
  Da UML/OCL als Quellsprache der Abbildung eine festgelegte Semantik (siehe [Obj99b]) hat und auch die Zielsprache in einer bestimmten Weise interpretiert werden muß, macht eine Übersetzung natürlicherweise nur dann Sinn, wenn sie korrekt ist, d.h. wenn die Interpretation der generierten Ausdrücke der Zielsprache mit der Interpretation des ursprünglichen UML/OCL-Modells übereinstimmt. Ein solches Korrektheitskriterium ist anwendungsspezifisch und muß für jede Anwendung neu formuliert werden.

  Wir wollen später am Beispiel des KeY-Projekts die Formulierung eines solchen Kriteriums demonstrieren.

  Eine spezielle Eigenheit von OCL sei an dieser Stelle explizit hervorgehoben: Die Möglichkeit der Undefiniertheit für OCL-Ausdrücke. Ein OCL-Ausdruck kann unter gewissen Umständen undefiniert sein; OCL stellt dazu einen entsprechenden Wert (implizit) zur Verfügung. In vielen Zielsprachen wird es keinen solchen Wert geben. In solchen Fällen muß man also eine Strategie anwenden, um die Undefiniertheit eines OCL-Ausdrucks in der Zielsprache in angemessener Weise aufzulösen, damit die Abbildung korrekt bleibt.

- **Erfüllung eines Gütekriteriums**
  Im allgemeinen wird es viele Möglichkeiten geben, das gegebene Modell in die gewünschte Zielsprache zu übersetzen, aber nicht jede Übersetzung wird „gleich gut" sein. Um die erzeugten Ausdrücke innerhalb einer bestimmten Anwendung gewinnbringend einsetzen zu können, muß die Abbildung einem entsprechenden Gütekriterium genügen.

  Was das genau bedeutet, muß für die im einzelnen betrachtete Anwendung festgelegt werden. Für das KeY-Projekt werden wir das weiter unten tun.

- **Dynamisches Abbildungsverfahren**
  Aus Gründen der Optimierung der erzeugten Ausdrücke bezüglich des Gütekriteriums für die betrachtete Anwendung kann man ebenfalls erwarten, daß OCL-Ausdrücke nicht immer auf die gleichen Ausdrücke der Zielsprache abgebildet werden, sondern die Übersetzung eines OCL-Ausdrucks `e` abhängig von dem syntaktischen Kontext, in dem `e` auftritt, eine günstige Darstellung wählt.

  Die Notwendigkeit eines dynamischen Abbildungsverfahrens ergibt sich auch aus einem anderen Grund: In einer konkreten Anwendung können die generierten Ausdrücke für unterschiedliche Zwecke verwendet werden, was im allgemeinen eine unterschiedliche Übersetzung erfordert.

- **Leicht Anpassung an ähnliche Anwendungen**
  UML/OCL als die Standardsprache für die Modellierung beliebiger Systeme erfreut sich wachsender Beliebtheit in der Industrie und erreicht eine zunehmende Verbreitung in der Praxis. Damit eng verbunden ist die Entwicklung und Verbreitung von Werkzeugen, die UML/OCL unterstützen. Das bedeutet, daß immer mehr Anwendungen zu erwarten sind, die UML/OCL-Modelle verarbeiten und dabei eine Abbildung von UML/OCL in irgendeine geeignete Zielsprache verwenden. Insofern wäre es wünschenswert, bereits bestehende Abbildungen direkt wiederverwenden oder leicht anpassen zu können. Im allgemeinen wird das nicht funktionieren, da die einzelnen Anwendungen zu verschieden sein werden, innerhalb bestimmter Anwendungsgruppen hingegen kann das gut funktionieren: Man denke beispielsweise an die Übersetzung von OCL-Ausdrücken in eine imperative Programmiersprache (Java, C++, Eiffel, Pascal etc.), um OCL-Ausdrücke zur Laufzeit einer Implementierung auszuwerten, oder aber die Übersetzung von UML/OCL in beliebige Logiksprache, um eine deduktive Verarbeitung der Modelle zu ermöglichen.

Wir wollen uns im folgenden auf eine *spezielle* Anwendung konzentrieren und die oben genannten Kriterien für diese Anwendung *konkretisieren* und *verfeinern*:

Diese Arbeit entstand Rahmen des KeY-Projekts, das wir in Abschnitt 1.2 bereits kurz vorgestellt haben. Das Ziel des Projektes liegt in der Integration von formalen Methoden zur Software-Verifikation in den industriellen Software-Prozeß. Das bedeutet insbesondere, daß eine zu verifizierende Implementierung eines Software-Systems durch ein UML/OCL-Modell spezifiziert wird. Um das UML/OCL-Modell in der Verifikation deduktiv verarbeiten zu können, benötigt man eine Übersetzung von UML/OCL in eine Logiksprache. In unserem Falle entspricht die Zielsprache der Logiksprache **DL** [Bec01], einer speziellen dynamische Logik für JAVA-CARD, die zur Verifikation von JAVA-CARD-Programmen besonders geeignet ist.

Die während der Übersetzung generierten Formeln werden mithilfe eines Beweissystems für den Nachweis von sogenannten Beweisverpflichtungen benutzt, die z.B. besagen, daß das gegebene Modell in sich konsistent ist, oder daß die betrachtete Implementierung ihrer Spezifikation genügt.

Die im Zuge der Software-Verifikation anfallenden Beweisaufgaben sind im allgemeinen so komplex und hart, daß eine rein automatische Behandlung aussichtslos ist. Zum Beweisen wird deshalb ein *interaktives* (und teilautomatisiertes) Beweissystem eingesetzt, bei dem ein Mensch in Zusammenarbeit mit dem System versucht, die Aufgabe zu lösen. Der Mensch liefert dabei die wesentlichen Kernideen des Beweises, wohingegen das Beweissystem sicherstellt, daß alle Schritte detailliert und korrekt ausgeführt sind, und viele einfache Beweisaufgaben, die als Teilaufgaben bei der Verifikation anfallen, ohne Hilfe des Menschen automatisch löst.

Wir betrachten daher insbesondere die Lesbarkeit bzw. Verständlichkeit der entstehenden Formeln für den menschlichen Beweiser als einen kritischen Einflußfaktor für den Nutzen einer solchen Abbildung im Rahmen des *interaktiven* Beweisens und damit des KeY-Projekts.

Man könnte an dieser Stelle einwenden, daß die Lesbarkeit nicht als entscheidender Faktor für das Qualitätsmaß einer solche *Abbildung* herangezogen werden soll-

te, da man in einem teilautomatisierten Beweissystem *nach* Übersetzung und *bevor* der menschliche Beweiser erstmalig zum Zuge kommt, die generierten Formeln mit Rewriting-Methoden behandeln und so in eine lesbare Form bringen könnte. Wir sind jedoch der Meinung, daß der Einsatz von Rewriting-Methoden in seiner Leistung beschränkter ist, als eine mit Hinblick auf Lesbarkeit entwickelte Abbildung, da es nach der Übersetzung – auf der Ebene von Termen und Formeln, die für Rewriting-Regeln nichts mehr mit dem ursprünglichen UML/OCL-Modell zu tun haben – möglicherweise ,,zu spät" für grundlegende Veränderungen der generierten Ausdrücke ist, wohingegen zum Zeitpunkt der Übersetzung – auf der Ebene von UML/OCL – alle verfügbaren Informationen und Verknüpfungen vorhanden sind. Außerdem gewinnt man einen zusätzlichen Freiheitsgrad für Optimierungen, wenn man die Lesbarkeit der entstehenden Formeln in das Gütekriterium mitaufnimmt, da man anschließend immer noch Rewriting-Methoden anwenden kann. Inwiefern noch weitere Einflußfaktoren für das KeY-Projekt eine wichtige Rolle spielen, ist momentan unklar und muß in zukünftigen Fallstudien untersucht werden.

Die Erfüllung des **Gütekriteriums** aus dem obigen Anforderungskatalog läßt sich also im Rahmen des KeY-Projektes folgendermaßen verfeinern:

- **Hohe Ähnlichkeit zwischen einem OCL-Ausdruck und seiner Übersetzung.**
  Die entstehenden Formeln und Terme sollen – wie oben beschrieben – als Eingabe für ein interaktives Beweissystem dienen. Da OCL-Ausdrücke erfahrungsgemäß relativ leicht zu lesen sind[2] und man davon ausgehen kann, daß der menschlichen Beweiser mit den abzubildenden OCL-Ausdrücken vertraut ist, würde eine Transformation, deren Ergebnisformeln den Ausgangsausdrücken ähneln, dem Menschen das Arbeiten mit den generierten Formeln beim Beweisen erleichtern.

- **Entstehende Formeln sollten möglichst einfach sein.**
  Je einfacher die generierten Formeln sind, desto leichter sind sie für den Menschen zu verstehen; dieses Verständnis der Formeln erleichtert gerade die Beweissituation für den menschlichen Beweiser und hat damit unter Umständen entscheidenden Einfluß auf die Akzeptanz eines Systems wie KeY.

  Genauer gesagt, sollte die Übersetzung eines OCL-Ausdrucks nicht wesentlich komplizierter sein, als der ursprüngliche OCL-Ausdruck selbst. Im Idealfall sollte sie sogar einfacher sein.

Auch die Anforderung der **Anpaßbarkeit** läßt sich an dieser Stelle präzisieren:
KeY befaßt sich mit UML/OCL im Kontext der Software-Verifikation, weshalb als Zielsprache die spezielle Logiksprache **DL** gewählt wurde. Es wäre sicherlich wünschenswert und naheliegend eine Abbildung zu konstruieren, die sich leicht auf andere Logiksprachen anpassen läßt, so daß die entwickelte Übersetzung in gewisser Weise als Ausgangspunkt für verschiedenste Anwendungen dienen kann, die mit dem logischen Schließen über UML/OCL-Modellen zu tun haben. Die Ausgangssituation dafür scheint günstig zu sein, denn dynamische Logiken sind Erweiterungen von klassischen Prädikatenlogiken erster Stufe, die wiederum als Kern für viele in der Praxis angewendete, ausdrucksstarke Logiksprachen bzw. formale Methoden dient.

[2] Dies war ja gerade eines der Design-Ziele bei der Entwicklung von OCL. Siehe [WK99].

Der Frage der **Korrektheit** der Abbildung wenden wir uns in detaillierter Form in Abschnitt 2.2 zu. Dort versuchen wir das Korrektheitskriterium in Form des Begriffs der *Informationserhaltung* zu formalisieren.

Man beachte, daß auch unsere Zielsprache kein explizites Äquivalent zu undefinierten Ausdrücken in OCL bietet. Wir müssen die undefinierten Werte also entsprechend auflösen. Wir versuchen dabei insbesondere eine Technik zu verwenden, die auch auf andere Logiksprachen übertragbar ist.

Es ist klar, daß sich die Ziele nicht orthogonal zueinander verhalten, sie stehen vielmehr in Konkurrenz. Wir müssen deshalb einen geeigneten Kompromiß bei der Verfolgung der einzelnen, oben genannten Ziele eingehen.

2.2 Ein Korrektheitskriterium für die Abbildung – Informationserhaltung

Jede Invariante und jede Vor- bzw. Nachbedingung, die mit OCL formuliert (und somit formalisiert) wurden, repräsentiert eine Gesetzmäßigkeit der modellierten Miniwelt und kapselt daher eine bestimmte semantische Information. Diese Information darf bei der Abbildung von OCL-Ausdrücken für Invarianten und Vor- bzw. Nachbedingungen *nicht* verfälscht werden, da ansonsten jeder formale Beweis einer Aussage über Gesetzmäßigkeiten der Miniwelt, der die generierten Formeln benutzt, schlichtweg unbrauchbar ist. Wir erwarten also, daß eine Übersetzung in dem Sinne korrekt sein sollte, daß sie die im zu übersetzenden Modell enthaltenen semantischen Informationen in die erzeugte Beschreibung in der Zielsprache unverfälscht überträgt und somit diese Informationen erhält.

Eine Klärung und Präzisierung des intuitiven Begriffs *Informationserhaltung* für eine solche Abbildung von OCL-Constraints in Formeln einer Logiksprache ist daher von außerordentlicher Bedeutung. Wir wollen uns damit in Abschnitt 2.2.1 befassen.

Haben wir schließlich eine klare Vorstellung darüber gewonnen, was man unter *Informationserhaltung* versteht, so können wir über eine mathematische Formalisierung des Begriffs die Möglichkeit schaffen, über die Korrektheit einer Übersetzung von OCL in eine Logiksprache bzw. formale Methode nachzudenken und sogar mit formalen Mitteln unzweifelhaft die Korrektheit nachzuweisen. In Abschnitt 2.2.2 wollen wir eine solche Formalisierung angeben, die allgemein genug ist, um für *beliebige* Ziellogiken verwendet werden zu können.

Verschiedene Logiksprachen können sich jedoch erheblich in ihrer Ausdrucksstärke unterscheiden und das oben angegebene Kriterium könnte im Hinblick auf eine konkrete Logiksprache und deren Ausdrucksmöglichkeiten angepaßt und vereinfacht werden, indem einige Eigenschaften, die in unserer allgemeinen Formalisierung noch semantisch auf einer Metaebene beschrieben sind, in der Logiksprache selbst – d.h. syntaktisch durch Formeln der Zielsprache – darzustellen. Wir wollen eine solche Anpassung für das KeY-Projekt und damit die Logik **DL** in Abschnitt 2.2.3 beispielhaft durchführen.

2.2.1 Klärung des Begriffs

Wir wollen in diesem Abschnitt auf abstrakter und eher informeller Ebene diskutieren und klären, welche semantische Information überhaupt durch ein UML/OCL-Modell dargestellt wird, um anschließend unsere intuitive Vorstellung von *Informationserhaltung* konkretisieren zu können.

Gegeben sein ein UML/OCL-Modell $\mathsf{M} = \mathcal{D}, C_1, \ldots, C_n$, das aus dem UML-Modell $\mathcal{D}$ und den OCL-Constraints $C_1, \ldots, C_n$ besteht. Die einzelnen Constaints $C_1, \ldots, C_n$ stellen jeweils eine Invariante I oder eine Methodenspezifikation P dar, d.h wir betrachten genauer eine Menge von Invarianten $I_1, \ldots, I_k$, sowie eine Menge von Methodenspezifikationen $P_1, \ldots, P_j$ $(n = k + j)$.

Ein UML-Modell $\mathcal{D}$ beschreibt eine Menge von Systemen $\mathsf{Systems}_{\mathcal{D}}$. Ein solches System $\sigma \in \mathsf{Systems}_{\mathcal{D}}$ wird gekennzeichnet durch seine statische Struktur und das dynamische Verhalten des Systems über dieser statischen Struktur.

Die statische Struktur legt fest, welche Zustände das System σ überhaupt einnehmen kann. Ein solcher Zustand legt fest, welche Objekte gerade im System existieren, beschreibt die konkreten Attributbelegungen aller Objekte und definiert, in welcher Weise die Objekte in dieser Momentaufnahme des Systems assoziiert sind. Die statische Struktur des Systems wird damit durch eine Teilmenge aller Instanziierungen D von $\mathcal{D}$ dargestellt

$$\mathsf{States}_{\sigma} \subseteq \mathsf{Instantiations}_{\mathcal{D}} := \{\, D \mid D \; \textit{Instanziierung von } \mathcal{D} \,\}$$

Das dynamische Verhalten des Systems σ hingegen beschreibt die Zustandsübergänge, die im System möglich sind. Zustandsübergänge werden dabei ausschließlich durch Ausführungen von Methoden m in $\mathcal{D}$ ermöglicht.

Eine Ausführung wird durch einen Methodenaufruf in einem Zustand D des Systems angestossen, der insbesondere von der Belegung β der Methodenparameter und des Kontextelements, d.h. dem Objekt auf dem die Methode aufgefrufen wird, abhängt. Ein Methodenaufruf überführt dann eine Paar (D, β) aus einer Instanziierung D und einer Variablenbelegung β in D, in ein[3] neues solches Paar (D', β'). D' entspricht dann der Instanziierung, die nach der Methodenausführung erreicht wird, und β' ist gerade die Variablenbelegung in D', die die Werte der Methodenparameter, des Kontextelements, des möglichen Rückgabewertes der Methode oder sonstiger Variablen am Ende der Methodenausführung dargestellt. Wir wollen im Zusammenhang von Methodenaufrufen, diese Paare ebenfalls als *Zustände* bezeichnen, insbesondere heißt (D, β) der *Vorzustand* des Methodenaufrufs und (D', β') dessen *Nachzustand.*

Die Zustandsübergänge die durch eine Methode m in $\mathcal{D}$ möglich sind, lassen sich damit durch Paare aus Vor- und Nachzuständen beschreiben, also einer Relation ρ_m mit

$$\mathsf{StatePairs}_{\mathcal{D}} := \{\, ((D, \beta), (D', \beta')) \mid \begin{array}{l} D, D' \in \mathsf{Instantiations}_{\sigma}, \\ \beta \; \textit{Variablenbelegung in } D, \\ \beta' \; \textit{Variablenbelegung in } D' \,\} \end{array}$$

[3] Wir betrachten hier ausschließlich deterministische Systeme.

$$\rho_m \subseteq \{\, ((D,\beta),(D',\beta')) \mid \begin{array}{l} D, D' \in \mathsf{States}_\sigma, \\ \beta\ Variablenbelegung\ in\ D, \\ \beta'\ Variablenbelegung\ in\ D' \,\} \subseteq StatePairs_{\mathcal{D}} \end{array}$$

Wir nennen eine solche Relation ρ_m *(abstrakte) Implementierung* (einer Methode m) *im System* σ bzw. eine beliebige Teilmenge $\rho \subseteq \mathsf{StatePairs}_{\mathcal{D}}$ eine *abstrakte Implementierung* über $\mathcal{D}$.

Das dynamische Verhalten des Systems σ wird somit durch Menge von Relationen ρ_m festgelegt:

$$\mathsf{StateTransitions}_\sigma := \{\, \rho_m \mid \begin{array}{l} m\ ist\ eine\ Methode\ in\ \mathcal{D}\ und \\ \rho_m\ ist\ eine\ Implementierung\ von\ m \,\} \end{array}$$

Das UML-Modell $\mathcal{D}$ und die Constraints $C_1, \ldots C_n$ beschreiben nun gemeinsam die gewünschten Anforderungen ein System, und zeichnen somit eine bestimmte Menge von Systemen aus: Die Menge der Systeme, die den Anforderungen der UML/OCL-Modellierung M genügen, d.h. die Menge

$$\mathsf{ValidSystems}_\mathsf{M} := \{\, \sigma \in \mathsf{Systems}_\mathcal{D} \mid \sigma\ genuegt\ \mathsf{M} \,\} \subseteq \mathsf{Systems}_\mathcal{D}$$

Wir wollen die Systeme in dieser Menge als *gültige* Systeme (bzgl. des UML/OCL-Modells M) bezeichnen und schreiben für diese Systeme kurz $\sigma \models \mathsf{M}$.

Das Ziel bei der Modellierung mit UML/OCL besteht nun gerade darin, die Menge der für eine konkrete Anwendung zulässigen Systeme exakt durch die Menge der gültigen Systeme (bzgl. der UML/OCL-Modellierung) zu beschreiben.

Die uns interessierende semantische Information, welche durch ein *UML/OCL-Modell* beschrieben wird und durch eine Übersetzung erhalten werden soll, besteht nun gerade in der Menge der bezüglich der Modellierung gültigen Systeme $\mathsf{ValidSystems}_\mathsf{M}$.

Was verstehen wir nun genau darunter, daß ein System σ einem UML/OCL-Modell $\mathsf{M} = \mathcal{D}, C_1, \ldots, C_n$ genügt? Informell betrachten wir ein System σ aus $\mathsf{Systems}_\mathcal{D}$ als gültig bezüglich M, falls es alle Anforderungen erfüllt, die in M angegeben sind: Das UML-Modell $\mathcal{D}$ (mit all seinen graphischen Notationen), sowie alle Constraints $C_1, \ldots, C_n$ gestellt werden.

Es sind dabei *zwei* Klassen von Anforderungen zu unterscheiden: Anforderungen statische Struktur des Systems, also die möglichen Systemzustände States_σ in σ, sowie Anforderungen an das dynamische Verhalten des Systems, d.h. die Zustandsübergänge $\mathsf{StateTransitions}_\sigma$ in σ.

Wir betrachten zunächst die Anforderungen an die möglichen Systemzustände des modellierten Systems, welche durch M dargestellt werden:

Das Modell $\mathcal{D}$ und die Invarianten $I_1, \ldots, I_k$ charakterisieren gemeinsam alle *Zustände*, die das modellierte System annehmen darf. Ein solcher Zustand legt – wie bereits erwähnt – fest, welche Objekte gerade im System existieren, beschreibt die konkreten Attributbelegungen aller Objekte und definiert, in welcher Weise die Objekte in dieser Momentaufnahme des Systems assoziiert sind. Eine solche Momentaufnahme des modellierten Systems kann in UML selbst durch sogenannte *Objektdiagramme* dargestellt werden.

Wir wollen im folgenden zwischen Instanziierungen, gültigen Instanziierungen und Snapshots eines UML/OCL-Modells $\mathcal{D}$ unterscheiden:

Definition 1 (Instanziierung eines UML-Modells)
Sei $\mathcal{D}$ ein UML-Modell, das um die OCL-Constraints $C_1, \ldots, C_n$ angereichert wurde.

Eine **Instanziierung** *D des UML-Modells $\mathcal{D}$ entspricht einem Objektdiagramm von $\mathcal{D}$, welches alle* semantischen *Informationen, die durch graphische Notationen – wie Multiplizitätsangaben oder Stereotypen – in $\mathcal{D}$ festgelegt sind, verletzen kann.*

Wir bezeichnen eine Instanziierung D eines UML-Modells $\mathcal{D}$ als **gültige Instanziierung** *von $\mathcal{D}$, wenn sie allen graphisch spezifizierten Einschränkungen genügt. Wir schreiben in diesem Fall $D \models \mathcal{D}$.*

Erfüllt eine gültige Instanziierung D außerdem alle Invarianten $I_1, \ldots, I_k$, so nennen wir D einen **Snapshot** *(bzw. Zustand) des durch $\mathcal{D}, I_1, \ldots, I_k$ modellierten Systems und notieren $D \models \mathcal{D}, I_1, \ldots, I_k$* ◁

Invarianten werden nun bezüglich einer Instanziierung D des Modells $\mathcal{D}$ ausgewertet und erhalten so eine Bedeutung:

Sie können über dem betrachteten D erfüllt, nicht erfüllt oder undefiniert sein. Betrachtet man verschiedene Instanziierungen des UML-Modells $\mathcal{D}$, so wird man im allgemeinen gültige Instanziierungen von $\mathcal{D}$ finden, welche die Invarianten $I_1, \ldots, I_k$ erfüllen, aber auch solche, die für das Modell $\mathcal{D}$ gültig sind, aber gegen mindestens eine der durch die Invarianten formulierten zusätzlichen Einschränkungen verstoßen – mit anderen Worten, die Invarianten $I_1, \ldots, I_k$ schränken in der Regel die Menge der gültigen Instanzen eines UML-Modells weiter ein. Diese Einschränkung kann in den meisten Fällen nicht unter alleiniger Nutzung der graphischen Notationen der UML charakterisiert werden.

Ein System $\sigma \in \mathsf{Systems}_{\mathcal{D}}$ erfüllt somit die statischen Anforderungen in M, falls jeder Zustand des Systems σ sowohl dem UML-Diagrammen genügt, als auch den einzelnen Invarianten, also

$$\text{Für alle } D \in \mathsf{States}_\sigma \text{ gilt: } D \models \mathcal{D}, I_1, \ldots, I_k$$

Wir notieren in diesem Fall kurz $\sigma \models \mathcal{D}, I_1, \ldots, I_k$.

Wir wenden uns schließlich den durch M dargestellten Anforderungen an das dynamische Verhalten des modellierten Systems zu:

Das dynamische Verhalten des modellierten Systems wird durch die Methodenspezifikationen $P_1, \ldots, P_j$ aus dem UML/OCL-Modell M eingeschränkt, indem jeder Constraint P zu einer Methode m eine Anforderung an die zugehörige abstrakte Implementierung ρ_m aus $\mathsf{StateTransitions}_\sigma$ stellt:

Sei P eine Methodenspezifikation $P = (V, N)$ mit der Vorbedinung V und der Nachbedingung N[4] zu der Methode $m(p_1{:}T_1, \ldots, p_n{:}T_n) : T$ in der Klasse C und $c{:}C$ das

[4] Die Annahme von jeweils genau einer Vor- bzw. Nachbedingung stellte keine Einschränkung der Allgemeinheit dar: Im Falle von mehreren Vor- bzw. Nachbedingungen verwenden wir jeweils die Konjunktion der einzelnen Vor- bzw. Nachbedingungen; wurde gar keine Vor- bzw. Nachbedingung angegeben, so verwenden wir `true`.

Kontextelement der Spezifikation, also der Platzhalter für das Objekt, für welches die Methode aufgerufen wird.

Da V und N boolesche OCL-Ausdrücke sind, können diese Ausdrücke gerade das Kontextelement c des Contraints (meist `self`), den Rückgabewert *result* sowie die Methodenparameter $p_1, \ldots, p_n$ als freie Variablen beeinhalten.

Der Erfüllbarkeitsbegriff für die booleschen OCL-Ausdrücke E umfaßt daher neben einer Instanziierung D auch eine Variablenbelegung β für die freien Variablen in diesen Ausdrücken: $D, \beta \models E$ besagt, daß der boolesche OCL-Ausdruck E unter der gegebenen Instanziierung D und der Variablenbelegung β zu wahr ausgewertet wird. In Methodenspezifikationen kann in Nachbedingungen außerdem auf die Auswertung eines Operators bezüglich des Zustands *vor* der Methodenausführung bestanden werden, weshalb der Erfüllbarkeitsbegriff für boolesche OCL-Ausdrücke E_{post} in Nachbedingungen zwei Zustände benötigt: $(D, \beta), (D', \beta') \models E_{post}$ besagt, daß der boolesche OCL-Ausdruck E_{post} unter der gegebenen Instanziierung D' und der Variablenbelegung β' zu wahr ausgewertet wird, wenn alle mit `@pre` gekennzeichneten Eigenschaften im Zustand (D, β) ausgewertet werden.

Eine Methodenspezifikation P einer Methode m aus einem UML-Modell $\mathcal{D}$ wird nun von einer abstrakten Implementierung ρ_m bzgl. einer Zustandsmenge $M \subseteq \mathsf{Instantiations}_{\mathcal{D}}$ erfüllt, falls gilt:

> Für alle Instanziierungen $D \in M$ von $\mathcal{D}$ und
> alle Variablenbelegungen β in D gilt:
> Wenn $(D, \beta) \models V$, dann muß gelten:
> Es gibt ein Zustandspaar $((D, \beta), (D', \beta')) \in \rho_m$ mit
> $D' \in M$ und $(D, \beta), (D', \beta') \models N$

Wir schreiben dann kurz: $M, \rho_m \models P$.

Eine Methodenspezifikation P einer Methode m aus einem UML-Modell $\mathcal{D}$ wird nun erfüllt von einer abstrakten Implementierung ρ_m eines Systems $\sigma \in \mathsf{Systems}_{\mathcal{D}}$, falls gilt: $\mathsf{States}_\sigma, \rho_m \models P$.

Wir schreiben dann kurz: $\sigma, \rho_m \models P$.

Man beachte, daß D' gerade die Instanziierung wiederspiegelt, die das System durch die Methodenausführung erreicht, und β' eine Belegung in D' ist, die gerade der Belegung der freien Variablen im Nachzustand entspricht.

Eine Methodenspezifikation P von m wird nun erfüllt vom System σ (kurz $\sigma \models P$), falls gilt:

> Für die zu m gehörende abstrakte Implementierung
> $\rho_m \in \mathsf{StateTransitions}_\sigma$ in σ gilt: $\sigma, \rho_m \models P$

Das bedeutet, daß ein System $\sigma \in \mathsf{Systems}_{\mathcal{D}}$ die dynamischen Anforderungen des UML/OCL-Modells M erfüllt, falls das System jede Methodenspezifikation $P_1, \ldots, P_j$ in M erfüllt, also

> Für alle Methodenspezifikationen P_i aus M, $i \in \{1, \ldots n\}$ gilt: $\sigma \models P_i$

Wir schreiben unter diesem Umständen einfach $\sigma \models P_1, \ldots, P_j$.

Schließlich können wir unserem ursprüngliches Anliegen jetzt nachkommen und eine präzise Definition für die Aussage, daß ein System σ einem UML/OCL-Modell M genügt, angeben:

Ein System σ genügt einem UML/OCL-Modell $\mathsf{M} = \mathcal{D}, C_1, \ldots, C_n$, falls σ den statischen und dynamischen Anforderungen aus M genügt, d.h.

> Seien $I_1, \ldots I_k$ die Invarianten und
> $P_1, \ldots P_j$ die Methodenspezifikationen in M.
> Dann muß gelten: $\sigma \models \mathcal{D}, I_1, \ldots, I_k$ und $\sigma \models P_1, \ldots, P_j$

Wir schreiben dann, wie vereinbart $\sigma \models \mathsf{M}$.

Bemerkung (Eigenschaften solcher Systeme). Man beachte insbesondere, daß für ein System σ, welches $\sigma \models \mathsf{M}$ erfüllt, somit unter anderem gilt, daß die Menge des Systemzustände eine Teilmenge der *gültigen* Instanziierungen von $\mathcal{D}$ bildet, und die abstrakten Implementierungen der Methoden in $\mathcal{D}$ jeweils *gültige* Systemzustände wieder in *gültige* Systemzustände überführen. Das entspricht sehr schön der Vorstellung, daß das System σ eine *korrekte* Realisierung des Modells M ist. □

Damit haben wir die Erfüllbarkeit einer UML/OCL-Modellierung M über einem System σ zurückgeführt auf Erfüllbarkeitsbegiffe, die wir in naheliegender Weise *jeder* der Zielsprache L durch die logische Erfüllbarkeit von Formeln nachempfinden können.

Indem wir die einzelnen in den Definitionen auftretenden sematischen Größen aus UML/OCL (Instanziierungen, Variablenbelegungen, Zustände, Instanziierungsmengen, Relationen über Zustandsmengen und Systeme) durch korrepsondierende semantische Objekte in der Zielsprache (Strukturen, Variablenbelegungen etc.) nachempfinden, können wir versuchen, die Situation, unter der ein UML/OCL-Modell M ausgewertet wird, exakt in der Logik nachzubilden. Sollte das möglich sein, so können wir in natürlicher Weise auch davon sprechen, daß das Modell M unter den *gleichen* Umständen die Anforderungen der Übersetzung $trans(\mathsf{M})$ erfüllt, was es uns schließlich ermöglichen würde, einen Korrektheitsbegriff formal angeben zu können!

Wir wollen daher die Zielsprachen der Übersetzung und die Formalisierung $trans(\mathsf{M})$ eines UML/OCL-Modells M in diesen Zielsprachen genauer betrachten:

Zielseitig befassen wir uns mit Logiksprachen, die einer gewissen Mindestanforderung genügen sollten: Sie müssen in irgend einer Weise eine Erweiterung einer klassischen Prädikatenlogik sein, d.h. wir können Berechnungen in Form von Termen beschreiben und die üblichen universellen und existenziellen Quantifizierungen vornehmen.

In der Ziellogik verwenden wir nun Symbole aus einer Signatur Σ^*, um bestimmte Größen aus dem Modell – wie die Typen aus OCL oder dem Modell, die Attribute der Klassen, Assoziationen oder Operatoren aus OCL – in der Logik zu repräsentieren. Unter Verwendung dieser Signatur, die insbesondere eine aus der Formalisierung des UML-Modells $\mathcal{D}$ (sowie aus Signaturen von eventuell verwendeten abstrakten Datentypen) stammende Basissignatur $\Sigma_{\mathcal{D}}$ enthält, sowie Symbole, welche Operatoren aus OCL darstellen, generiert eine Übersetzung schließlich Formeln, die zum einen die semantischen Informationen, welche durch die graphischen Notationen in UML-Modell

$\mathcal{D}$ selbst beschrieben sind, formal fassen und zum anderen die Aussagen der Constraints $C_1, \ldots, C_n$ entsprechend der Semantik von UML/OCL in der Logiksprache formalisieren.

Wir betrachten daher das Ergebnis der Übersetzung $\mathsf{M} = \mathcal{D}, C_1, \ldots, C_n$ eines UML/OCL-Modells abstrakt als ein Tupel aus der Übersetzung des UML-Modells $\mathcal{D}$, sowie den Übersetzungen der einzelnen Constraints, d.h. wir erhalten durch die Übersetzung $trans(\mathsf{M}) = (trans(\mathcal{D}), trans(C_1), \ldots, trans(C_n))$. Da die einzelnen Elemente des Tupels sind nun Formeln aus unserer Zielsprache, da die jeweils übersetzen Größen Anforderungen – also Aussagen – in UML/OCL beschreiben. Das UML-Modell und die einzelnen Constraints sind jeweils atomare und unabhängige Anforderungen an ein System (im Modell M). Eine Übersetzung wird insofern diese Größen unabhängig voneinander behandeln, weshalb diese Modellierung an dieser Stelle sinnvoll erscheint.

Diese Formeln werden nun bezüglich einer (semantischen) *Struktur* $\mathcal{S}$ (über der Signatur Σ^*) ausgewertet, die diesen Formeln eine konkrete Bedeutung verleiht. Auf der semantischen Ebene haben also Instanziierungen D und Strukturen $\mathcal{S}$ eine äquivalente Funktion und können korrespondieren, d.h. genauer:

Strukturen $\mathcal{S}$ über der aus dem UML-Diagramm $\mathcal{D}$ (und möglicherweise verwendeten ADTs) gewonnenen Basissignatur $\Sigma_{\mathcal{D}}$, welche die Symbole für die in OCL eingebauten Typen geeignet interpretieren – beispielsweise sollten entsprechende Standard-Axiomatisierungen für Domänen wie die ganzen Zahlen oder Mengen erfüllt sein – , beschreiben jeweils eine Instanziierung D des Modells $\mathcal{D}$. Wir schreiben in einem solchen Fall $\mathcal{S} \simeq D$.

Eine Struktur $\mathcal{S}^*$ über der erweiterten Signatur Σ^* beschreibt somit genau dann eine Instanziierung D von $\mathcal{D}$, falls sie eine Σ^*-Erweiterung einer $\Sigma_{\mathcal{D}}$-Struktur $\mathcal{S}$ ist, mit $\mathcal{S} \simeq D$, bzw. für die $\Sigma_{\mathcal{D}}$-Einschränkung $\mathcal{S}^*|_{\Sigma_{\mathcal{D}}}$ der Struktur $\mathcal{S}^*$ gilt $\mathcal{S}^*|_{\Sigma_{\mathcal{D}}} \simeq D$. In diesem Fall wollen wir von einer *Korrespondenz* zwischen der Struktur $\mathcal{S}^*$ und der Instanziierung D sprechen.

Nun kann jedem Snapshot D (mindestens) eine entsprechende Σ^*-Struktur $\mathcal{S}_D$ zugeordnet werden, wohingegen im allgemeinen nicht jede Σ^*-Struktur $\mathcal{S}$ zu einem Snapshot D beschreibt, da beispielweise die angegebenen Multiplizitäten in $\mathcal{D}$ durch $\mathcal{S}$ verletzt werden.

Für die Auswertung einer Formel über eine Σ^*-Struktur $\mathcal{S}_D$ benötigen wir im allgemeinen außerdem Werte für die freien Variablen aus der Formel, als eine Variablenbelegungen β_L über der Struktur $\mathcal{S}_D$. Variablenbelegungen β in UML/OCL über einer Instanziierung D und Varablenbelegungen β_L über einer korrespondierenden Σ^*-Struktur haben also eine entsprechende Funktion. Aus einer solchen Belegung β ergibt sich für eine feste korrepsondierende Struktur $\mathcal{S}_D$ genau eine korrespondierende Variablenbelegung β_L in $\mathcal{S}_D$. Wir schreiben dann $\beta_L \simeq \beta$.

Da wir nun Instanziierungen D durch die korrespondierende Strukturen $\mathcal{S}_{\mathcal{D}}$ und Variablenbelegungen β durch korrespondierende Variablenbelegungen β_L in jeder hier betrachteten Logik L nachbilden bzw. simulieren können, und alle anderen für uns relevanten semantischen Größen (Zustände, Zustandspaare, Zustandsmengen, Zustandspaarmengen) aus UML/OCL aus diesen atomaren Bestandteilen aufgebaut sind, ist es uns nun auch möglich, auch diese (komplexen) Größen durch *korrpesondierende* Objekte (Paare aus Strukturen und Variablenbelegungen, Mengen solcher Paare, Mengen

von Paaren solcher Paare), welche den ursprünglichen Größen aus UML/OCL genau entsprechen, in der Zielsprache *auf einer Metaebene*[5] nachzubilden:

Seien D bzw. D' beliebige Instanziierungen des UML-Modells $\mathcal{D}$ und β bzw. β' beliebige Variablenbelegungen in D bzw. D'. Seien $\mathcal{S}_D$ bzw. $\mathcal{S}_{D'}$ beliebige Σ^*-Strukturen und β_L bzw. β'_L beliebige Variablenbelegungen in $\mathcal{S}_D$ bzw. $\mathcal{S}_{D'}$. Sei M eine beliebige Menge von Zuständen und M' eine beliebige Menge von Zustandspaaren. Sein M_L eine beliebige Menge von Paaren $(\mathcal{S}_D, \beta_L)$ und M'_L eine beliebige Menge von Paaren $((\mathcal{S}_D, \beta_L), (\mathcal{S}_{D'}, \beta'_L))$

Wir wollen der vereinfachenden Sprechweise wegen ein Paar $(\mathcal{S}_D, \beta_L)$ als *Zustand* (in L), ein Paar $((\mathcal{S}_D, \beta_L), ((\mathcal{S}_{D'}, \beta'_L))$ als *Zustandspaar* (in L), eine Menge M_L als *Zustandsmenge* (in L) und eine Menge M'_L als *Zustandspaarmenge* (in L) bezeichnen.

Dann definieren wir als *Korrespondenz*-Relation $\simeq$ für die einzelnen aus den elementaren Begriffen *Instanziierung* und *Variablenbelegung* aufgebauten (komplexen) semantischen Größen aus UML/OCL:

- *Zustände*

  $(\mathcal{S}_D, \beta_L) \simeq (D, \beta)$ **gdw.** $\mathcal{S}_D \simeq D$ und $\beta_L \simeq \beta$

  Wir bezeichnen dann $(\mathcal{S}_D, \beta_L)$ als einen zu (D, β) *korrespondierenden Zustand* (in L).

- *Zustandspaare*

  $((\mathcal{S}_D, \beta_L), (\mathcal{S}_{D'}, \beta'_L)) \simeq ((D, \beta), (D', \beta'))$ **gdw.**
  $(\mathcal{S}_D, \beta_L) \simeq (D, \beta)$ und $(\mathcal{S}_{D'}, \beta'_L) \simeq (D', \beta')$

  Wir bezeichnen dann $((\mathcal{S}_D, \beta_L), (\mathcal{S}_{D'}, \beta'_L))$ als ein zu $((D, \beta), (D', \beta'))$ *korrespondierendes Zustandspaar* (in L).

- *Mengen von Zuständen oder Zustandspaaren*

  $M_L \simeq M$ **gdw.**

  Für alle Zustände z_L (in L) gilt:

  $z_L \in M_L$ gdw. (es gibt ein $z \in M$ mit $z_L \simeq z$)

  Das bedeutet also, daß die Elemente in M_L genau die zu z korrespondierenden Zustände z_L (in L) sind.

  Wir bezeichnen dann M_L als eine zu M *korrespondierende Zustandsmenge* (in L).

  Über eine analoge Definition, die Zustandspaare anstelle von Zuständen benutzt, erhalten wir den Begriff der zu M' *korrespondierenden Zustandsmenge* M'_L

Man beachte, daß insbesondere mit dem letztgenannten Punkt es für uns möglich ist, alle für uns relevanten Aspekte eines Systems σ, also die statische Struktur States_σ und das dynamische Verhalten $\mathsf{StateTransitions}_\sigma$ des Systems, in der Logik nachzubilden.

[5] Alle hier betrachteten Logiksprachen stellen eine Erweiterung einer klassischen Prädikatenlogik dar. Wir können daher für eine solche Logik L in der allgemeinen Betrachtung lediglich annehmen, daß sie die elementaren semantischen Begriffe *Struktur* und *Variablenbelegung explizit* unterstützt. Betrachten wir hingegen eine *bestimmte* Logiksprache, so lassen sich dann eventuell auch komplexere semantische Begriffe aus UML/OCL *direkt* in der Logik darstellen, beispielsweise können wir in der Logik **DL** Zustände (aus einem System) *direkt* durch Zustände (in einer Kripke-Struktur) repräsentieren!

Bemerkung (Korrespondenz im Falle eines Systems). Identifizieren wir außerdem ein System σ mit der Beschreibung seiner statischen Strukur und der Beschreibung seines dynamischen Verhaltens – lassen wir also alle anderen Aspekte eines abstrakten Systems außen vor (was für unsere Zwecke hier problemlos möglich ist) –, also $\sigma = (\mathsf{States}_\sigma, \mathsf{StateTransitions}_\sigma)$, so können wir auch für ein *System* ein korresponierendes semantisches Objekt σ_L in der Logik L angeben, welches *genau* dieses System σ in der Logik repräsentiert!

Dieses Objekt σ_L ist ein ein Paares aus einer Menge von Zuständen (in L) und einer Menge abstrakter Implementierungen (in L) – also Relationen auf der Zustandsmenge (in L). Es entspricht damit einer *prädikatenlogischen Kripke-Struktur* $\mathcal{K}_\sigma$. Wir simulieren also im Prinzip (auf einer Metaebene) in der Logik L ein *System* durch eine korrespondierende *Kripke-Struktur* (bzw. ein System σ induziert eine solche korrepsondierende Kripke-Struktur).

Dieser Sachverhalt legt daher in schöner Weise dar, daß *modale Prädikatenlogiken* der *natürlichste* mathematische Formalismus zur Darstellung von Systemen σ, die durch UML/OCL-Modelle beschrieben werden, und somit *modale prädikatenlogische Sprachen* besonders angemessen zur *vollständigen* Beschreibung – d.h. der statischen Struktur *und* des dynamischen Verhaltens – solcher Systeme (durch eine formale Sprache) sind.

Unterstützt nun eine Logiksprache L Kripke-Strukturen explizit als semantisches Konzept (und bietet daher *syntaktische* Mittel an, um über Zustandsübergänge Aussagen zu treffen) so ist zu erwarten, das einige der folgenden Definitionen einfacher zu formulieren sein werden. Wir werden diesen Punkt in Abschnitt 2.2.3 nocheinmal aufgreifen. □

Damit ist es also grob gesprochen möglich, die Situation (exakter das System σ), bezüglich der das UML/OCL-Modell M ausgewertet wird, genau die Logik zu übertragen bzw. dort nachzubilden. Wir werden jedoch aus Platzgründen die folgenden Ausführungen und Definitionen nicht unter dieser allgemeinen Betrachtung weiterführen – obwohl das sicherlich interessant wäre und möglichweise zusätzliche Einsichten liefern würde – und nutzen lediglich den Korrespondenzbegriff für die elementarsten semantischen Größen in den hier betrachteten Logiksprachen L – d.h. Strukturen und Variablenbelegungen – aus.

Nach diesen Betrachtungen können wir nun in natürlicher Weise auch davon sprechen, daß ein System σ die Übersetzung $trans(\mathsf{M})$ des UML/OCL-Modells M erfüllt und eine genaue Definition dafür entwickeln:

Definition 2 (Erfüllung der Übersetzung einer Methodenspezifikation)
Sei $\sigma = (\mathsf{States}_\sigma, \mathsf{StateTransitions}_\sigma)$ *ein System aus* $\mathsf{Systems}_\mathcal{D}$. *Sei* P *eine Methodenspezifikation einer Methode* m *aus* $\mathcal{D}$. *Sei* $trans(P)$ *die Übersetzung der Methodenspezifikation* P *und* $trans(V)$ *bzw.* $trans(N)$ *die Übersetzung der Vor- bzw. Nachbedingung von* P.

Eine abstrakte Implementierung ρ_m **erfüllt die Übersetzung der Methodenspezifikation** P **bzgl. der Zustandsmenge** $M \subseteq \mathsf{Instantiations}_\mathcal{D}$, *falls gilt:*

Für alle Instanziierungen $D \in M$ von $\mathcal{D}$,
alle Variablenbelegungen β in D,
alle zu D korrespondierenden Strukturen $\mathcal{S}_D$ und
alle zu β korrespondierenden Variablenbelegungen β_L in $\mathcal{S}_D$ gilt:
$(\mathcal{S}_D, \beta_L) \models_L trans(V)$ und
Wenn $(D, \beta) \models V$, dann muß gelten:
Es gibt ein Zustandspaar $((D, \beta), (D', \beta')) \in \rho_m$ mit $D' \in M$ und
$(\mathcal{S}_D, \beta_L), (\mathcal{S}_{D'}, \beta'_L) \models trans(N)$
für alle zu D' korrespondierenden Strukturen $\mathcal{S}_{D'}$ und
alle zu β' korrespondierenden Variablenbelegungen β'_L in $\mathcal{S}_{D'}$

Wir schreiben kurz: $M, \rho_m \models_L trans(P)$.
Die abstrakte Implementierung ρ_m in σ **erfüllt die Übersetzung der Methodenspezifikation** *P, falls gilt:*

$$\mathsf{States}_\sigma, \rho_m \models_L trans(P)$$

Wir schreiben kurz: $\sigma, \rho_m \models_L trans(P)$.

Das System σ **erfüllt die Übersetzung der Methodenspezifikation** *P, falls gilt:*

Die zu der spezifizierten Methode m gehörende
abstrakte Implementierung ρ_m in σ erfüllt: $\sigma, \rho_m \models_L trans(P)$.

Wir schreiben kurz: $\sigma \models_L trans(P)$. ◁

Bemerkung (Zielsprache und abstrakte Implementierungen). Die Definition der Relation $\rho_m \models_L trans(P)$ ist in der gegebenen Form für Zielsprachen L erforderlich, welche *nicht* genügend ausdruckstark sind, um *syntaktisch* über eine abstrakte Implementierung ρ_m zu sprechen.

Für Zielsprachen, die eine solche syntaktische Beschreibung zulassen, kann die Definition vereinfacht werden: Wir müssen dann nicht mehr über die Übersetzungen $trans(V)$ bzw. $trans(V)$ sprechen, sondern direkt die Übersetzung $trans(P)$ und ein zu ρ_m korrepsondierendes semantisches Objekt ρ_m^L verwenden.

Wir werden diesen Aspekt in Abschnitt 2.2.3 nocheinmal betrachten. □

Definition 3 (Erfüllung der Anforderungen aus $trans(\mathsf{M})$)
Sei $\mathsf{M} = \mathcal{D}, C_1, \ldots, C_n$ ein UML/OCL-Modell, $I_1, \ldots, I_k$ die Invarianten und $P_1, \ldots, P_j$ die Methodenspezifikationen aus M.
Sei $trans(\mathsf{M}) = (trans(\mathcal{D}), trans(C_1), \ldots, trans(C_n))$ die Übersetzung von M.
Sei $\sigma = (\mathsf{States}_\sigma, \mathsf{StateTransitions}_\sigma)$ ein System.

Das System σ **erfüllt die statischen Anforderungen der Übersetzung** *$trans(\mathsf{M})$, falls gilt:*

Für alle Instanziierungen $D \in \mathsf{States}_\sigma$ und
alle zu D korrespondierenden Strukturen $\mathcal{S}_D$ gilt:
$\mathcal{S}_D \models_L trans(\mathcal{D})$ und
$\mathcal{S}_D \models_L trans(I_l)$ für alle $l \in \{1, \ldots, k\}$

Wir schreiben kurz: $\sigma \models_L trans(\mathcal{D}), trans(I_1), \ldots, trans(I_k))$.

Das System σ **erfüllt die dynamischen Anforderungen der Übersetzung** $trans(\mathsf{M})$, *falls gilt:*

$$\sigma \models_L trans(P_l) \text{ für alle } l \in \{1, \ldots, j\}$$

Wir schreiben kurz: $\sigma \models_L trans(P_1), \ldots, trans(P_j))$.

◁

Wir können nun definieren, wann ein System σ bezüglich der Übersetzung $trans(\mathsf{M})$ des Modells M gültig ist: σ erfüllt sowohl die statischen als auch die dynamischen Anforderungen, die in $trans(\mathsf{M})$ formalisiert wurden.

Definition 4 (Gültigkeit eines Systems σ **bezüglich** $trans(\mathsf{M})$**)**
Sei $\mathsf{M} = \mathcal{D}, C_1, \ldots, C_n$ *ein UML/OCL-Modell,* $I_1, \ldots, I_k$ *die Invarianten und* $P_1, \ldots, P_j$ *die Methodenspezifikationen aus* M.
Sei $trans(\mathsf{M}) = (trans(\mathcal{D}), trans(C_1), \ldots, trans(C_n))$ *die Übersetzung von* M.

Ein System $\sigma \in \mathsf{Systems}_{\mathcal{D}}$ *ist* **gültig bezüglich der Übersetzung** $trans(\mathsf{M})$ *des Modells* M, *falls gilt:*

$$\begin{array}{l} \sigma \models_L trans(\mathcal{D}), trans(I_1), \ldots, trans(I_k) \text{ und} \\ \sigma \models_L trans(P_1), \ldots, trans(P_j) \end{array}$$

Wir schreiben unter diesen Umständen kurz: $\sigma \models_L trans(\mathsf{M})$. ◁

Wie wir am Anfang dargelegt haben, zeichnet ein UML/OCL-Modell M durch die in ihm enthaltenen Anforderungen eine feste Teilmenge der Menge $\mathsf{Systems}_{\mathcal{D}}$ aller möglichen Systeme über $\mathcal{D}$ besonders aus: Die Menge $\mathsf{ValidSystems}_{\mathcal{D}}$ der gültigen Systeme für das Modell M. Wir haben dort außerdem erkannt, daß diese Menge gerade die semantische Information verkörpert, die in dem Modell M steckt und durch eine Übersetzung erhalten werden muß.

Eine Übersetzung des UML/OCL-Modells M erzeugt aus den einzelnen Bestandteilen des Modelles eine neue Beschreibung in der Zielsprache L: $trans(\mathsf{M})$, welche die Anforderungen aus M in Form von Formeln der Zielsprache L wiedergibt.

Diese Formalisierung des UML/OCL-Modells M in der Zielsprache zeichnet nun ihrerseits eine Menge von Systemen besonders aus, nämlich die Systeme $\sigma \in \mathsf{Systems}_{\mathcal{D}}$, welche die durch die erzeugte Formalisierung beschriebenen Anforderungen genügen, d.h. also (entsprechend der vorangehenden Diskussion) die Menge

$$\mathsf{ValidSystems}_{trans(\mathsf{M})} := \{\, \sigma \in \mathsf{Systems}_{\mathcal{D}} \mid \sigma \models_L trans(\mathsf{M}) \,\} \subseteq \mathsf{Systems}_{\mathcal{D}}$$

Eine Übersetzung eines UML/OCL-Modells M kann nun als *korrekt* angesehen werden, wenn die durch die Übersetzung $trans(\mathsf{M})$ beschriebene Menge von Systemen genau mit der Menge der bzgl. des UML/OCL-Modells gültigen Systeme übereinstimmt, wenn also gilt

$$\mathsf{ValidSystems}_{\mathsf{M}} = \mathsf{ValidSystems}_{trans(\mathsf{M})}$$

Definition 5 (Informationserhaltung bezüglich UML/OCL-Modellen)
Eine Übersetzung trans von UML/OCL-Modellen in eine der hier betrachteten Logiksprachen L heißt **informationserhaltend bzgl. UML/OCL-Modellen**, *falls gilt:*

> *Für alle UML-Modelle $\mathcal{D}$, alle UML/OCL-Modelle M über $\mathcal{D}$ und alle Systeme $\sigma \in \mathsf{Systems}_{\mathcal{D}}$ gilt:*
> $$\sigma \models \mathsf{M} \textbf{ gdw. } \sigma \models_L trans(\mathsf{M})$$

◁

Wir haben nun ausführlich einen Korrektheitsbegriff für das *Gesamtverfahren* hergeleitet, wenn man das Verfahren in Bezug auf *UML/OCL-Modelle* betrachtet.

Bei der Entwicklung solchen Übersetzung *trans* wird man im allgemeinen jeweils *einzelne Verfahren* $trans_{UML}$, $trans_I$ und $trans_P$ zur Behandlung des UML-Modells $\mathcal{D}$, der Invarianten und der Methodenspezifikationen erstellen, da diese Größen die elementaren Bestandteile eines UML/OCL-Modells M und die zugehörigen inhaltlichen Aussagen prinzipiell unabhängig voneinander sind. Das Gesamtverfahren kombiniert dann die einzelnen Verfahren zur Übersetzung von UML-Modellen $\mathcal{D}$, Invarianten I und Methodenspezifikationen P.

Aus den obigen Definitionen läßt sich nun ableiten, welchen Anforderung diese einzelnen Verfahren genügen müssen, damit das Gesamtverfahren im obigen Sinne korrekt ist. Die Klasse der Übersetzungen, die diese Eigenschaft aufweisen, nennen wir $\mathsf{IPTranslations}_{Model}$.

Damit sind wir mit unseren Betrachtungen jedoch noch nicht am Ende. Wir werden nun erläutern, daß bei einer Betrachtung der Übersetzung auf einer *feineren* Ebene als der Ebene der UML/OCL-Modelle sich ein stärkerer Korrektheitsbegriff für Übersetzung bezüglich dieser feineren Ebene angeben läßt.

Sei im folgenden $\sigma \in \mathsf{Systems}_{\mathcal{D}}$ ein beliebiges, aber festes System, welches durch ein UML/OCL-Modell M über $\mathcal{D}$ beschrieben werden soll.

Wir wollen nun die Ebene des *UML-Modells und der einzelnen OCL-Constraints* genauer untersuchen und beginnen mit den Invarianten und dem UML-Modell.

Die statischen Anforderungen aus dem UML/OCL-Modell M – das UML-Modell $\mathcal{D}$ und die Invarianten $I_1, \ldots, I_k$ – beschreiben gemeinsam die erlaubten Systemzustände – die Snapshots von M – für alle Systeme σ, welche als korrekte Realisierung des UML/OCL-Modells angesehen werden können, also die Menge

$$\mathsf{Snapshots}_{\mathsf{M}} := \{D \mid D \models \mathcal{D}, I_1, \ldots, I_k\}$$

Die einzelnen statischen Anforderungen lassen sich damit nicht nur – wie bisher – als Anforderungen an ein *System* ansehen, sondern auch feinkörniger als Anforderungen an einzelne *Zustände*.

Betrachtet man eine Invariante I als Anforderung an einzelne *Zustände* eines modellierten Systems. so zeichnet sie eine bestimmte Teilmenge der gültigen Instanziierungen D des Modells $\mathcal{D}$ besonders aus, nämlich die Menge der gültigen Instanziierungen, über denen die Invariante I erfüllt ist:

$$\mathsf{ValidStates}_I = \{D \mid D \models \mathcal{D} \text{ und } D \models I\ \}$$

In ähnlicher Weise beschreibt die zugehörige Übersetzung $trans(I)$ der Invariante I eine bestimmte Teilmenge der gültigen Instanziierungen D des Modells $\mathcal{D}$:

$$\begin{aligned}\mathsf{ValidStates}_{trans(I)} = \{D \mid\ & D \models \mathcal{D} \text{ und}\\ & \text{für alle zu } D \text{ korrespondierenden}\\ & \Sigma^*\text{-Strukturen } \mathcal{S}_D \text{ gilt: } \mathcal{S}_D \models_L trans(I)\ \}\end{aligned}$$

Die durch *eine* Invariante I ausgezeichnete Menge gültiger Instanziierungen kann man wiederum als die *semantische* Information auffassen, die in einer *einzelnen Invariante* steckt und nicht durch das UML-Modell alleine dargestellt wird.

Wir könnten also von einer Übersetzung $trans(I)$ einer Invariante I fordern, daß sie die gleiche Menge von gültigen Instanziierungen D des Modells $\mathcal{D}$ auszeichnet, wie der Constraint selbst, d.h. daß die jeweils ausgezeichneten Instanziierungsmengen $\mathsf{ValidStates}_I$ und $\mathsf{ValidStates}_{trans(I)}$ übereinstimmen.

Wir nennen schließlich eine Übersetzung von UML/OCL in eine Logiksprache *informationserhaltend bzgl. Invarianten*, wenn die Abbildung für jede beliebige Invariante I, ein Ergebnis $trans(I)$ generiert, welches die gleiche Menge gültiger Instanziierungen von $\mathcal{D}$ beschreibt.

Damit fordern wir also für eine informationserhaltende Übertragung von *Invarianten* I, daß für jede gültige Instanziierung D von $\mathcal{D}$ gilt: D erfüllt die Invariante I genau dann wenn die generierte Formel $trans(I)$ über allen korrespondierenden Σ^*-Strukturen $\mathcal{S}_D$ erfüllt ist, also

$$\begin{gathered}D \models I\\ \textbf{gdw.}\\ \mathcal{S}_D \models_L trans(I) \text{ für alle zu } D \text{ korrespondierenden } \Sigma^*\text{-Strukturen } \mathcal{S}_D\end{gathered}$$

Daher gilt für eine bzgl. Invarianten informationserhaltende Übersetzung, daß die Menge $\mathsf{Snapshots}_\mathsf{M}$ der Snapshots des UML/OCL-Modells M gerade der Menge gültiger Instanziierungen entspricht, die durch die Konjunktion der einzelnen Übersetzungen $trans(I_1), \ldots, trans(I_k)$ beschrieben wird:

$$\begin{aligned}\mathsf{Snapshots}_\mathsf{M} &= \{D \mid D \models \mathcal{D}, I_1, \ldots, I_k\}\\ &= \bigcap_{i=1,\ldots,k} \mathsf{ValidStates}_{I_i}\\ &= \bigcap_{i=1,\ldots,k} \mathsf{ValidStates}_{trans(I_i)}\\ &= \{D \mid\ D \models \mathcal{D}\ und\\ &\qquad fuer\ alle\ zu\ D\ korrespondierenden\ \Sigma^*{-}Strukturen\ \mathcal{S}_D\\ &\qquad gilt: \mathcal{S}_D \models_L trans(I_1) \wedge \ldots \wedge trans(I_k)\}\end{aligned}$$

Erzeugt die Übersetzung außerdem noch eine formale Beschreibung $trans(\mathcal{D})$ der „Gültigkeit" einer Instanziierung D von $\mathcal{D}$ – also der Informationen in $\mathcal{D}$, die durch graphische Notationen festgehalten sind – dann läßt sich die Menge $\mathsf{Snapshots}_\mathsf{M}$ der Snapshots des UML/OCL-Modells sogar rein formal durch Formeln in der Zielsprache charakterisieren:

Die Übersetzung $trans(\mathcal{D})$ des UML-Modells $\mathcal{D}$ spiegelt die semantischen Informationen, die im UML-Modell $\mathcal{D}$ durch rein graphische Notationen dargestellt wurden, adäquat wieder, falls die durch $trans(\mathcal{D})$ charakterisierten Instanziierungen D von $\mathcal{D}$ gerade die gültigen Instanziierungen sind, also sofern gilt:

Für alle Instanziierungen D von $\mathcal{D}$:
$$D \models \mathcal{D}$$
gdw.
für alle zu D korrespondierenden Σ^*-Strukturen $\mathcal{S}_D$ gilt:
$$\mathcal{S}_D \models_L trans(\mathcal{D})$$

Wir nennen schließlich eine Übersetzung von Diagrammen $\mathcal{D}$ in eine Logiksprache *informationserhaltend bzgl. Diagrammen*, falls die Abbildung für jedes Diagramm $\mathcal{D}$ eine formale Beschreibung $trans(\mathcal{D})$ generiert, welche genau die Menge der gültigen Instanziierungen von $\mathcal{D}$ beschreibt.

Für eine bzgl. Invarianten und Diagrammen informationserhaltende Abbildung gilt somit:

$$\begin{aligned}\mathsf{Snapshots}_{\mathsf{M}} = \{D \mid\ & fuer\ alle\ zu\ D\ korrespondierenden\ \Sigma^* - Strukturen\ \mathcal{S}_D \\ & gilt : \mathcal{S}_D \models_L trans(\mathcal{D}) \wedge trans(I_1) \wedge \ldots \wedge trans(I_k)\}\end{aligned}$$

bzw.

Für jede Instanziierung D von $\mathcal{D}$ gilt:
$$D \models \mathcal{D}, I_1, \ldots I_k$$
gdw.
für alle zu D korrespondierenden Σ^*-Strukturen $\mathcal{S}_D$ gilt:
$$\mathcal{S}_D \models_L trans(\mathcal{D}) \wedge trans(I_1) \wedge \ldots \wedge trans(I_k)$$

Und das ist gerade die Eigenschaft, die garantiert, daß wir durch eine Übersetzung die statischen Anforderungen an die erlaubten *Zustände* aus einem UML/OCL-Modells exakt in der Zielsprache L wiedergeben und damit die Informationen aus der Beschreibung der statischen Anforderungen für *Zustände* in der Quellsprache nicht verfälscht wird.

Bemerkung (Betrachtung von gültigen Instanziierungen). Man beachte, daß eine Übersetzung entsprechend dem gerade angegebenen Kriterium im Falle von *Invarianten* die Erfüllbarkeit nur für *gültige* Instanziierungen D erhalten muß, was eine schwächere Anfordung darstellt, als die Erhaltung der Erfüllbarkeit für *beliebige* Instanziierungen.

Diese schwächere Formulierung reicht jedoch für unsere Zwecke aus, was auf auf der Tatsache beruht, daß es uns bei der feinkörnigeren Betrachtung um die Erhaltung der *Snapshot*-Eigenschaft eines Zustands durch eine Übersetzung geht, und die Menge der Snapshots eine Teilmenge der gültigen Zustände bilden. Insofern muß die Abbildung nur für gültige Instanziierungen diese Eigenschaft erhalten, da alle anderen Instanziierungen in jedem Fall *keine* Snapshots darstellen, auch wenn diese Instanziierungen einen Constraint C erfüllen mögen. Die stärkere Formulierung stellt für unsere Zwecke keine erforderliche Einschränkung dar. □

Wir wenden uns nun in der gleichen Weise den Anforderungen an das dynamische Verhalten eines Systems zu:

Die Methodenspezifikationen $P_1, \ldots, P_j$ aus dem UML/OCL-Modell stellen gemeinsam die Anforderungen an das dynamische Verhalten des modellierten Systems

dar und beschreiben die Menge $\mathsf{ValidImplementations}_{\mathsf{M}} \subseteq \mathsf{StatePairs}_{\mathcal{D}}$ der erlaubten abstrakten Implementierungen des UML/OCL-Modells

$$\mathsf{ValidStatePairs}_{\mathcal{D}} := \{\, ((D,\beta),(D',\beta')) \mid ((D,\beta),(D',\beta')) \in \mathsf{StatePairs}_{\mathcal{D}} \textit{ und } D \models \mathcal{D} \textit{ und } D' \models \mathcal{D} \,\} \subseteq \mathsf{StatePairs}_{\mathcal{D}}$$

$$\begin{aligned}\mathsf{ValidImplementations}_{\mathsf{M}} := \{\rho_m \mid\ & \rho_m \subseteq \mathsf{ValidStatePairs}_{\mathcal{D}} \text{ und}\\ & \text{für alle } P_i, i \in \{1,\ldots,j\} \text{ gilt:}\\ & \quad \text{Wenn } \rho_m \text{ die durch } P_i \text{ spezifizierte}\\ & \quad \text{Methode } m \text{ implementiert,}\\ & \quad \text{dann ist } \mathsf{ValidStatePairs}_{\mathcal{D}}, \rho_m \models P \text{ erfüllt.}\}\end{aligned}$$

Interpretiert man eine Methodenspezifikation P als eine über das UML-Modell $\mathcal{D}$ hinausgehende Einschränkung an die abstrakten Implementierungen, die gültige Zustände des Systems wieder in gültige Zustände überführen, dann gelten nun die selben Überlegungen wie im Falle von Invarianten mit Hinblick auf abstrakte Implementierungen $\rho \subseteq \mathsf{StatePairs}_{\mathcal{D}}$:

$$\begin{aligned}\mathsf{ValidImplementations}_{P} = \{\rho \mid\ & \rho \subseteq \mathsf{ValidStatePairs}_{\mathcal{D}} \text{ und}\\ & \mathsf{ValidStatePairs}_{\mathcal{D}}, \rho \models P \,\}\end{aligned}$$

ist die Menge abstrakter Implementierungen, die durch die Methodenspezifikation P erlaubt wird. Ihre Übersetzung läßt hingegen die folgenden abstrakten Implementierungen zu:

$$\begin{aligned}\mathsf{ValidImplementations}_{trans(P)} = \{\rho \mid\ & \rho \subseteq \mathsf{ValidStatePairs}_{\mathcal{D}} \text{ und}\\ & \mathsf{ValidStatePairs}_{\mathcal{D}}, \rho \models_L trans(P) \,\}\end{aligned}$$

Erfüllt eine Übersetzung die Forderung nach der Gleichheit der beiden durch P und $trans(P)$ ausgezeichneten Mengen abstrakter Implementierungen, also

$$\begin{aligned}&\text{Für alle abstrakten Implementierungen } \rho \subseteq \mathsf{ValidStatePairs}_{\mathcal{D}} \text{ über } \mathcal{D} \text{ gilt:}\\ &\mathsf{ValidStatePairs}_{\mathcal{D}}, \rho \models P \textbf{ gdw. } \mathsf{ValidStatePairs}_{\mathcal{D}}, \rho \models_L trans(P)\end{aligned}$$

für beliebige Methodenspezifikationen, dann nennen wir die Abbildung *informationserhaltend bzgl. Methodenspezifikationen.*

Eine Übersetzung von UML/OCL-Modellen, die jeweils informationserhaltend bzgl. Invarianten, Methodenspezifikationen und UML-Diagrammen ist, nennen wir kurz *informationserhaltend bzgl. Constraints.* Die Klasse der bzgl. Constraints informationserhaltenden Übersetzungen bezeichnen wir mit $\mathsf{IPTranslations}_{Const}$.

Ohne formalen Beweis weisen wir darauf hin, daß der folgende Zusammenhang zwischen den beiden für die verschiedenen Ebenen entwickelten Korrektheitsbegriffen besteht:

$$\mathsf{IPTranslations}_{Const} \subset \mathsf{IPTranslations}_{Model}$$

Daß die neue Klasse von Übersetzungen eine *echte* Teilmenge der zuerst eingeführten Klasse von Übersetzungen bildet, liegt im wesentlichen an der *feineren* bzw. *lokalen* Betrachtung eines Systems[6].

[6] Man findet ein ähnliches Phänomen beispielsweise bei der Entwicklung einer Konsequenzrelation in Modallogiken, wo man eine globale und lokale Relation angeben kann, wobei die lokale Relation eine echte Verstärkung des globalen Konsequenzbegriffs darstellt.

Wir erhalten also einen stärkeren Korrektheitsbegriff, wenn wir UML/OCL als eine Beschreibungssprache auf der Ebene von Zuständen und abstrakte Implementierungen auffassen und somit Erhaltung für die semantische Information aus den einzelnen Constraints fordern, anstatt UML/OCL grober als Beschreibungssprache für Systeme aufzufassen und daher auf der Informationserhaltung auf der Ebene der UML/OCL-Modelle zu verlangen.

Wir können unsere Sicht auf Systeme für den Fall von Methodenspezifikationen noch etwas weiter verfeinern: Wir fassen dazu eine Methodenspezifikationen noch detaillierter als sprachliches Mittel zur Beschreibung von Anforderungen an *Zustandspaare* (anstelle von abstrakten Implementierungen) auf. Durch eine Methodenspezifikation P aus dem Modell M mit der Vorbedingung V und der Nachedingung N wird somit die Menge

$$\begin{aligned}\mathsf{ValidStatePairs}_P := \{((D,\beta),(D',\beta')) \mid\ & ((D,\beta),(D',\beta')) \in \mathsf{ValidStatePairs}_{\mathcal{D}} \\ & \text{und } D,\beta \models V \\ & \text{und } (D,\beta),(D',\beta') \models N\end{aligned}$$

festgelegt.

Die Übersetzung $trans(P)$ einer Methodenspezifikation bildet die Vor- und Nachbedingung durch die Formeln $trans(V)$ und $trans(N)$ nach, und zeichnet ihrerseits eine bestimmte Menge gültiger Zustandspaare aus:

$$\begin{aligned}\mathsf{ValidStatePairs}_{trans(P)} := \{\ & ((D,\beta),(D',\beta')) \mid \\ & ((D,\beta),(D',\beta')) \in \mathsf{ValidStatePairs}_{\mathcal{D}} \\ & \text{und alle } ((\mathcal{S}_D,\beta_L),(\mathcal{S}_{D'},\beta'_L)) \simeq ((D,\beta),(D',\beta')) \\ & \text{erfüllen: } (\mathcal{S}_D,\beta_L) \models_L trans(V) \text{ und} \\ & (\mathcal{S}_D,\beta_L),(\mathcal{S}_{D'},\beta'_L) \models_L trans(N)\ \ \}\end{aligned}$$

Bemerkung (Erfüllbarkeit boolescher Ausdrücke). Der Erfüllbarkeitsbegriff für die booleschen OCL-Ausdrücke B in Nachedingungen umfaßt wie bereits angemerkt, ein Paar $((D,\beta),(D',\beta'))$ aus Zuständen und wir schreiben $(D,\beta),(D',\beta') \models B$, falls der boolesche OCL-Ausdruck B unter der gegebenen Instanziierung D' und der Variablenbelegung β' zu wahr ausgewertet wird, wenn alle mit `@pre` gekennzeichneten Eigenschaften im Zustand (D,β) ausgewertet werden. Analog notieren wir in der Ziellogik L für die korrespondierenden Strukturen, Variablenbelegungen und die Übersetzung des Ausdrucks $(\mathcal{S}_D,\beta_L),(\mathcal{S}_{D'},\beta'_L) \models trans(B)$ mit der gleichen Intention bezüglich der Interpretation der Funktionssymbole, die zu den mit `@pre` versehenen Eigenschaften in B gehören. □

Es ergibt sich also die folgende Anforderung an eine Übersetzung, die die in der Menge $\mathsf{ValidStatePairs}_P$ enthaltene Information erhält:

Für beliebige Vorbedingungen V und Nachbedingungen N über $\mathcal{D}$ und
alle gültigen Instanziierungen D, D' von $\mathcal{D}$ und
alle Variablenbelegungen β in D und β' in D' gilt:

$$(D, \beta) \models V$$
gdw.
$(\mathcal{S}_D, \beta_L) \models_L Th_V$
für alle zu D korrspondierenden Σ^*-Strukturen $\mathcal{S}_D$ und
alle zu β korrespondierenden Varablenbelegungen β_L in $\mathcal{S}_D$

und

$$(D, \beta), (D', \beta') \models N$$
gdw.
$(\mathcal{S}_D, \beta_L), (\mathcal{S}_{D'}, \beta'_L) \models_L Th_N$
für alle zu D korrspondierenden Σ^*-Strukturen $\mathcal{S}_D$ und
alle zu D' korrspondierenden Σ^*-Strukturen $\mathcal{S}_{D'}$ und
alle zu β korrespondierenden Varablenbelegungen β_L in $\mathcal{S}_D$ und
alle zu β' korrespondierenden Varablenbelegungen β'_L in $\mathcal{S}_{D'}$

Wir nennen eine Übersetzung die diese Forderung erfüllt *informationserhaltend für Vor-/Nachbedingungen.*

Eine Übersetzung von UML/OCL-Modellen, die jeweils informationserhaltend bzgl. Invarianten, Methodenspezifikationen sowie UML-Diagrammen ist, und außerdem informationserhaltend bzgl. Vor-/Nachbedingungen ist, nennen wir *informationserhaltend bzgl. Constraints und Vor-/Nachbedingungen.* Die Klasse der bzgl. Constraints und Vor-/Nachbedingungen informationserhaltenden Übersetzungen bezeichnen wir mit $\mathsf{IPTranslations}_{Cons,PrePost}$.

Ohne formalen Beweis weisen wir darauf hin, daß der folgende Zusammenhang zu den anderen Klassen von Übersetzungen besteht:

$$\mathsf{IPTranslations}_{Cons,PrePost} \subset \mathsf{IPTranslations}_{Cons}$$

Die Inklusion ist *echt*, d.h wir erhalten also eine weitere echte Verschärfung des Korrektheitsbegriff, wenn wir für eine Übersetzung neben der Informationserhaltung bzgl. Constraints, außerdem fordern, daß die Informationen auf der Ebene der Zustandspaare übertragen werden.

Eine weitere Verfeinerung der Betrachtung eines UML/OCL-Modells M ist nicht möglich, da Instanziierungen D und Zustände (D, β) die atomaren Bestandteile des UML/OCL-Modells darstellen; wir haben Informationserhaltungsbegriffe für alle relevanten Detaillierungsgrade der Betrachtung eines UML/OCL-Modells definiert.

Damit haben wir also den Begriff der Korrektheit einer Übersetzung beliebiger UML/OCL-Modellen in beliebige Logiksprachen prinzipiell durch Informationserhaltungsbegriffe auf verschiedenen Abstraktionsebenen formuliert und in einer allgemeinen Art und Weise ausführlich analysiert.

Wir haben insbesondere eine *strenge* Hierarchie von Informationserhaltungsbegriffen angegeben, die entlang dem Detaillierungsgrad verläuft, mit dem man die semantischen Objekte betrachtet, die durch die Sprache UML/OCL beschrieben werden.

Abschließend stellt sich somit die Frage, welcher Informationserhaltungsbegriff die *Korrektheit* einer Übersetzung darstellt, also in gewissem Sinne der „richtige" Begriff ist?

Die Antwort darauf ist recht naheliegend: Das kommt prinzipiell auf die jeweilige Anwendung an, die eine Übersetzung verwendet!

Benutzt man eine Übersetzung von UML/OCL-Modellen, um ausschließlich Modelle als unteilbares Ganzes zu betrachten, so ist die Klasse $\mathsf{IPTranslations}_{Model}$ prinzipiell ausreichend, um einen Korrektheitsbegriff festzulegen. Eine solch grobe Sichtweise auf ein UML/OCL-Modell wird jedoch für die meisten Anwendungen ungeeignet sein.

Möchte eine Anwendung jedoch mit den einzelnen Constraints – wiederum als unteilbares Ganzes – arbeiten, so ist der schärfere Informationserhaltungsbegriff, der durch die Klasse $\mathsf{IPTranslations}_{Cons}$ dargestellt wird, anzuwenden.

Arbeitet eine Anwendung außerdem auch mit den einzelnen booleschen Ausdrücken, die die eigentliche logische Aussage hinter einem Constraint verkörpern[7], so muß man auf die restriktivste Klasse $\mathsf{IPTranslations}_{Cons,PrePost}$ zurückgreifen.

Im allgemeinen werden die meisten Anwendungen wahrscheinlich an einer Abbildung aus der Klasse $\mathsf{IPTranslations}_{Cons,PrePost}$ interessiert sein. Wir verwenden damit in Abschnitt 2.2.2 diese Klasse zur Definition der Korrektheit einer Übersetzung.

2.2.2 Formalisierung für beliebige logische Zielsprachen

Wir wollen nun in diesem Abschnitt eine formalere Definition der *Korrektheit* für Übersetzungen angeben, die UML/OCL-Modelle in beliebige Logiksprachen abbilden, welche Erweiterungen einer klassischen prädikatenlogischen Sprache erster Stufe sind. Die Definition soll allgemein genug sein, um allen diese Sprachen gerecht zu werden.

Wir knüpfen hier an die Ausführungen in Abschnitt 2.2.1 an und konkretisieren diese mit Hinblick auf eine formale Semantik von UML/OCL:

Als Ausgangspunkt dient die formale Semantik von OCL wie sie in [Sch01b] entwickelt wurde: Dort werden insbesondere Instanziierungen D des Modells $\mathcal{D}$ durch eine mehrsortige Algebra $\mathcal{M}_D = (M_D, I_D)$ über einer Signatur $\Sigma_\mathcal{D}$ verkörpert, wobei M_D ein entsprechendes Universum ist, welches die Extensionen der einzelnen Klassen umfaßt, und I_D eine geeignete Interpretation der Signatur $\Sigma_\mathcal{D}$ darstellt, die bewirkt, daß die Algebra genau die Instanziierung D wiederspiegelt. Anschließend wird eine solche Algebra $\mathcal{M}_D$ zu einer OCL-Algebra $\mathcal{M}_D^{OCL} = (M_D^{OCL}, I^{OCL})$ über der Signatur $\Sigma_\mathcal{D}^{OCL} \supseteq \Sigma_\mathcal{D}$ erweitert, die dann zur eigentlichen Auswertung von OCL-Constraints herangezogen wird. Die Erweiterungen des Algebra $\mathcal{M}_D$ beziehen sich dabei auf die in OCL eingebauten Typen und ihre Operatoren.

[7] Das ist zum Bespiel der Fall, wenn die Übersetzungen der Vor- und Nachbedinungen zweier Methodenspezifikationen zu einer Formel kombiniert werden, die eine bestimmte für die Anwendung relevante Beweisverpflichtung darstellt.

Diese OCL-Algebren $\mathcal{M}_D$ treten in den Formulierungen aus Abschnitt 2.2.1 nun an die Stelle der zuvor abstrakt betrachteten Instanziierungen D.

Eine Übersetzung von OCL muß notwendigerweise das UML-Modell $\mathcal{D}$ formalisieren, an welches die zu übersetzenden OCL-Constraints gebunden sind und das als Basis für die Formulierung der Constraints selbst dient.

Dieses Modell umfaßt prinzipiell zwei Arten von Informationen: Eine Basissignatur, auf der OCL-Ausdrücke aufbauen, sowie bestimmte semantische Einschränkungen an das modellierte System, die durch graphische Annotationen dargestellt werden.

In der Logik entstehen somit aus dem Modell $\mathcal{D}$ zum einen eine Basissignatur $\Sigma_{\mathcal{D}}$, sowie eine Menge Th_D von Formeln (über der Signatur $\Sigma_{\mathcal{D}}$), welche die semantischen Informationen aus den graphischen Darstellungen in $\mathcal{D}$ einfangen.

Die Basissignatur $\Sigma_{\mathcal{D}}$ enthält im allgemeinen eine Signatur Σ_{ADT} von abstrakten Datentypen, die von der Übersetzung verwendet werden. Entsprechend enthält die Menge Th_D dann alle Formeln Th_{ADT} die zur Axiomatisierung der verwendeten ADTs dienen, beispielsweise eine Standardaxiomatisierung von Mengen oder der ganzen Zahlen.

Die Basissignatur $\Sigma_{\mathcal{D}}$ wird nun um Symbole erweitert, die die Operatoren aus OCL in irgend einer Weise verkörpern. Somit besteht das Ergebnis Th_C der Übersetzung eines Constraints C schließlich aus Formeln über der erweiterten Signatur $\Sigma^* \supseteq \Sigma_{\mathcal{D}}$.

Entsprechend dem Vorgehen im Abschnitt 2.2.1 wollen wir zunächst festlegen, was wir unter Informationserhaltung im Falle von Invarianten verstehen:

Sei $OCLConst_{\mathcal{D}}$ die Menge aller OCL-Constraints über dem Modell $\mathcal{D}$ und For_{Σ}^{L} die Menge der Formeln der Logiksprache L über der Signatur Σ.

Definition 6 (Informationserhaltung für Invarianten)
Sei $\mathcal{D}$ ein beliebiges UML-Modell und $\Sigma_{\mathcal{D}}$ die zugehörige Signatur aus der Formalisierung des Modells. Sei $\Sigma^* \supseteq \Sigma_{\mathcal{D}}$ die Signatur, die von der Übersetzung insgesamt verwendet wird.

Eine Abbildung trans:$OCLConst_{\mathcal{D}} \to For_{\Sigma^*}^{L}$ heißt **informationserhaltend für Invarianten**, *falls gilt:*
Für jede Invariante I über dem Modell $\mathcal{D}$ und jede OCL-Algebra $\mathcal{M}_D^{OCL}$ zu einer gültigen Instanziierung D des Modells $\mathcal{D}$ und alle zu D korrespondierenden $\Sigma_{\mathcal{D}}$-Strukturen $\mathcal{S}_D$ gilt:

$$I \text{ ist wahr in der OCL-Algebra } \mathcal{M}_D^{OCL}$$
$$\textbf{gdw.}$$
$$\mathcal{S}_D^* \models trans(I) \text{ für alle } \Sigma^*\text{-Erweiterungen } \mathcal{S}_D^* \text{ von } \mathcal{S}_D$$

◁

In ähnlicher Weise gehen wir für Methodenspezifikationen vor, wobei wir zunächst eine Definition der Informationserhaltung bezüglich Methodenspezifikationen angeben, die als Rahmen für eine geeignete Definition des Begriffs für alle Zielsprachen L dient.

Definition 7 (Informationserhaltung für Methodenspezifikationen)
Sei $\mathcal{D}$ ein beliebiges UML-Modell.

Eine Abbildung trans:$OCLConst_{\mathcal{D}} \to For_{\Sigma^*}^{L}$ heißt **informationserhaltend für Methodenspezifikationen**, *falls gilt:*

Für jede Methodenspezifikation P mit der Vorbedingung V und der Nachbedingung N über dem Modell $\mathcal{D}$, jede abstrakte Implementierung $\rho \subseteq \mathsf{ValidStatePairs}_{\mathcal{D}}$:

(Für alle OCL-Algebren $\mathcal{M}_D^{OCL}$ zu einer gültigen Instanziierung D und alle Variablenbelegungen β in $\mathcal{M}_D^{OCL}$ gilt:
Wenn $I_{D,\beta}^{OCL}(V) = \mathbf{true}$, dann muß gelten:
Es gibt ein Zustandspaar $((D,\beta),(D',\beta')) \in \rho$, so daß für alle OCL-Algebren $\mathcal{M}_{D'}^{OCL}$ zu D' und alle Variablenbelegungen β' in $\mathcal{M}_{D'}^{OCL}$ gilt:
$(\mathcal{M}_D^{OCL},\beta),(\mathcal{M}_{D'}^{OCL},\beta') \models N$)

gdw.

$$\mathsf{ValidStatePairs}_{\mathcal{D}}, \rho \models_L trans(P)$$

◁

Man beachte, generische Schablone für eine *bestimmte* Zielsprache L auf einfachste Weise angepaßt werden kann, indem man die Relation $\mathsf{ValidStatePairs}_{\mathcal{D}}, \rho \models_L trans(P)$ für diese Zielsprache geeignet definiert. Man erhält somit sehr elegant den erwünschten Informationserhaltungsbegiff für die betrachtete Zielsprache.

Nicht jede Logiksprachen L, die eine Erweiterung einer klassischen Prädikatenlogik verkörpert, ermöglicht es, *syntaktisch* in Formeln über Zustandsübergänge und somit abstrakte Implementierungen zu sprechen. In solchen Sprachen ist es daher *nicht* möglich die Semantik einer Methodenspezifikation *vollständig* durch eine Formel anzugeben. Der Einsatz solcher Sprachen – beispielsweise klassischer Prädikatenlogik erster Stufe – zur Formalisierung von UML/OCL-Modellen kann trotzdem für eine bestimmte Anwendung ausreichend sein.

Wir möchten daher für diese Zielsprachen eine geeignete Formulierung des Begriffs *Informationserhaltung bzgl. Methodenspezifikationen* angeben.

Durch eine einfache Festlegung der Relation $\mathsf{ValidStatePairs}_{\mathcal{D}}, \rho \models_L trans(P)$ ist es auch in diesem Fall möglich, die Formulierung aus Definition 7 auf unsere Bedürfnisse anzupassen:

In einer solchen Logiksprache L läßt sich die Bedeutung einer Methodenspezifikation P zwar nicht vollständig formulieren, zumindest lassen sich – wie restlichen Teil des Kapitels noch deutlich gemacht wird – die Bedeutung der Vor- und Nachbedingung – bei einer geeigneten Interpretation – formalisieren.

Wir fassen die Übersetzung $trans(P)$ für diese Sprachen daher nicht als eine Formel auf, sondern als ein Formelpaar $trans(P) = (trans(V), trans(N))$ aus den jeweiligen Übersetzungen der Vor- und Nachbedingung.

Die Definition der Relation $\mathsf{ValidStatePairs}_{\mathcal{D}}, \rho \models_L trans(P)$ lautet für diese Sprachen dann:

Definition 8 (Erfüllung von $trans(P)$ durch ρ auf gültigen Zuständen)
Sei $\mathcal{D}$ ein beliebiges UML-Modell und $\Sigma_{\mathcal{D}}$ die zugehörige Signatur aus der Formalisierung des Modells. Sei $\Sigma^* \supseteq \Sigma_{\mathcal{D}}$ die Signatur, die von der Übersetzung insgesamt verwendet wird. Sei P eine beliebige Methodenspezifikation über $\mathcal{D}$ mit der Vorbedingung V und der Nachbedingung N. Die Übersetzung $trans(P) = (trans(V), trans(N))$ von P sein das Paar aus den jeweiligen Übersetzungen von V und N. Sei $\rho \subseteq \mathsf{StatePairs}_{\mathcal{D}}$ eine beliebige abstrakte Implementierung.

Die abstrakte Implementierung ρ **erfüllt die Übersetzung der Methodenspezifikation** P **bzgl. der gültigen Zustandspaare von** $\mathcal{D}$, *falls gilt:*

Für alle gültigen Instanziierungen D *von* $\mathcal{D}$,
alle Variablenbelegungen β *in* D,
alle zu D *korrespondierenden* Σ^**-Strukturen* $\mathcal{S}^*_D$ *und*
alle zu β *korrespondierenden Variablenbelegungen* β_L *in* $\mathcal{S}^*_D$ *gilt:*
$(\mathcal{S}^*_D, \beta_L) \models_L trans(V)$ *und*
Wenn $(D, \beta) \models V$, *dann muß gelten:*
Es gibt ein Zustandspaar $((D, \beta), (D', \beta')) \in \rho$ *mit*
D' *ist eine gültig Instanziierung und*
$(\mathcal{S}^*_D, \beta_L), (\mathcal{S}^*_{D'}, \beta'_L) \models trans(N)$
für alle zu D' *korrespondierenden* Σ^**-Strukturen* $\mathcal{S}^*_{D'}$ *und*
alle zu β' *korrespondierenden Variablenbelegungen* β'_L *in* $\mathcal{S}^*_{D'}$

Wir schreiben in diesem Fall: $\mathsf{ValidStatePairs}_{\mathcal{D}}, \rho \models_L trans(P)$ ◁

Wir werden in Anschnitt 2.2.3 zeigen, wie man für die sehr ausdruckstarke Logik **DL** durch eine andere Festlegung der Definition 8 den Begriff der *Informationserhaltung bzgl. Methodenspezifikationen* für die **DL** anpaßt.

Bemerkung (Mehrere Vor- oder Nachbedingungen). In der obigen Definition wird von genau einer Vor- und einer Nachbedingung im Constraint P ausgegangen. Dies ist keine Einschränkung der Allgemeinheit, da im Falle von mehreren Vorbedingungen der OCL-Ausdruck für die konjunktive Verknüpfung der einzelnen Vorbedingungen betrachtet werden kann; ist gar keine Vorbedingung vorhanden, so kann der OCL-Ausdruck `true` verwendet werden. Das selbe gilt für mehrere oder fehlende Nachbedingungen im Constraint P. □

Definition 9 (Informationserhaltung bzgl. Vor-/Nachbedingungen)
Sei $\mathcal{D}$ *ein beliebiges UML-Modell und* $\Sigma_{\mathcal{D}}$ *die zugehörige Signatur aus der Formalisierung des Modells. Sei* $\Sigma^* \supseteq \Sigma_{\mathcal{D}}$ *die Signatur, die von der Übersetzung insgesamt verwendet wird.*

Eine Abbildung $trans{:}OCLConst_{\mathcal{D}} \to For^{L}_{\Sigma^*}$ *heißt* **informationserhaltend für Vor/Nachbedingungen**, *falls gilt:*
Für jede Methodenspezifikation P *über dem Modell* $\mathcal{D}$ *gilt: Wenn* V *die Vor- bzw.* N *die Nachbedingung in* P *und* Th_V *bzw.* Th_N *die zugehörigen Formeln aus der Übersetzung* $trans(P)$ *des Constraints sind, so sind, dann gilt für diese Größen:*

Für alle OCL-Algebren $\mathcal{M}_D^{OCL}, \mathcal{M}_{D'}^{OCL}$ zu gültigen Instanziierungen D, D' von $\mathcal{D}$ und alle Variablenbelegungen β in $\mathcal{M}_D^{OCL}$ und β' in $\mathcal{M}_{D'}^{OCL}$ gilt:

$$\mathcal{M}_{D'}^{OCL}, \beta' \models V$$
gdw.
$(\mathcal{S}_D, \beta_L) \models_L Th_V$
für alle zu D korrspondierenden Σ^*-Strukturen $\mathcal{S}_D$ und alle zu β korrespondierenden Varablenbelegungen β_L in $\mathcal{S}_D$

und

$$(\mathcal{M}_D^{OCL}, \beta), (\mathcal{M}_{D'}^{OCL}, \beta') \models N$$
gdw.
$(\mathcal{S}_D, \beta_L), (\mathcal{S}_{D'}, \beta'_L) \models_L Th_N$
für alle zu D korrspondierenden Σ^*-Strukturen $\mathcal{S}_D$ und alle zu D' korrspondierenden Σ^*-Strukturen $\mathcal{S}_{D'}$ und alle zu β korrespondierenden Varablenbelegungen β_L in $\mathcal{S}_D$ und alle zu β' korrespondierenden Varablenbelegungen β'_L in $\mathcal{S}_{D'}$

◁

Eine Übersetzung von UML/OCL-Modellen muß außerdem die UML-Diagramme formalisieren. Wir wünschen uns für eine solche Formalisierung, daß sie die ,,Gültigkeit" einer Instanziierung des Modells angemessen beschreibt:

Definition 10 (Informationserhaltung für Diagramme)
Sei $Diagrams^{UML}$ die Menge aller UML-Klassendiagramme. Sei Σ eine feste, geeignete Signatur, die von der Abbildung insgesamt verwendet werden kann.

Eine Abbildung trans:$Diagrams^{UML} \to For_\Sigma^L$ heißt **informationserhaltend für Diagramme**, *falls gilt:*
Für jedes Diagramm $\mathcal{D}$ und jede OCL-Algebra $\mathcal{M}_D^{OCL}$ zu einer beliebigen Instanziierung D des Diagramms $\mathcal{D}$ gilt:

Die OCL-Algebra $\mathcal{M}_D^{OCL}$ entspricht einer gültgen Instanziierung von $\mathcal{D}$
gdw.
$\mathcal{S}_D \models trans(\mathcal{D})$ für alle zu D korrespondierenden $\Sigma_\mathcal{D}$-Strukturen $\mathcal{S}_D$

◁

Nun wollen wir schließlich ausdrücken, was wir unter einer korrekten Übersetzung von UML/OCL-Modellen verstehen.

Definition 11 (Korrekte Übersetzung)
Unter einer **Übersetzung** *$T = (trans_\mathcal{D}^I, trans_\mathcal{D}^P, trans_{UML})$ von OCL in eine Logiksprache L verstehen wir ein Verfahren, das für beliebige UML/OCL-Modelle $\mathcal{D}$ eine Abbildung $trans_\mathcal{D}^I$ zur Formalisierung von Invarianten (über $\mathcal{D}$), eine Abbildung $trans_\mathcal{D}^P$ zur Formalisierung von Methodenspezifikationen (über $\mathcal{D}$) und eine Abbildung $trans_{UML}$ zur Formalisierung von UML-Modellen realisiert.*

Eine solche Übersetzung T nennen wir **korrekt**, *falls die Abbildung $trans_{UML}$ informationserhaltend für Diagramme ist und für beliebige UML/OCL-Modelle $\mathcal{D}$ die Abbildung $trans_\mathcal{D}^I$ informationserhaltend für Invarianten und die Abbildung $trans_\mathcal{D}^P$ infor-*

mationserhaltend sowohl für Methodenspezifikationen, als auch für Vor-/Nachbedingungen ist, wenn die bei der Formalisierung des UML-Modells tatsächlich benutzte Signatur $\Sigma_{\mathcal{D}}$ betrachtet wird. ◁

Wir werden im weiteren Verlauf der Arbeit unter anderem ein Übersetzungsverfahren von UML/OCL-Modellen angeben, welches allgemein genug ist, um für alle hier betrachteten Logiksprachen L anwendbar zu sein, und das korrekt[8] ist, sofern wir die Übersetzung einer Methodenspezifikation abstrakt als das Paar aus der Übersetzung der Vor- und der Nachbedingung ansehen. Für eine konkrete Anwendung, die das erzeugte Formelpaar in einer (bzgl. der zugehörigen Zielsprache L) geeigneten Weise kombiniert und verändert, kann so auf einfache Weise ein *korrektes* Übersetzungsverfahren abgeleitet werden.

2.2.3 Ein Korrektheitskriterium für das KeY-System

Wir werden in diesem Abschnitt demonstrieren, wie sich die allgemeinen Definitionen aus Anschnitt 2.2.2 für eine bestimmte Anwendung anpassen lassen: Dem KeY-Projekt.

Im KeY-System wird als Zielsprache die Logiksprache **DL** verwendet, die eine dynamische Prädikatenlogik verkörpert. Damit ist die Sprache **DL** eine sehr ausdrucksstarke Logiksprache, die es unter anderem erlaubt, den Zusammenhang zwischen den Instanziierung D für den Vorzustand und D' für den Nachzustand des Methodenaufrufs in der obigen Definition der Informationserhaltung bzgl. Methodenspezifikationen, sowie beispielsweise die Auswertung eines Funktionssymbols in verschiedenen Zuständen oder der Rückgabewert eines Methode schon in der Logiksprache selbst zu beschreiben.

Wir können in dieser Logiksprache daher die Semantik einer Methodenspezifikation P direkt durch eine Formel $trans(P)$ beschreiben, anstatt nur auf die Übersetzung der Vor- und der Nachbedingung zurückgreifen zu müssen.

Wir wollen nun ein Korrektheitskriterium angeben, welches das allgemeine Kriterium aus Abschnitt 2.2.2 für das KeY-Projekt anpaßt. Wir beschreiben in Form dieses Kriteriums insbesondere die Eigenschaft, die das im weiteren Verlauf der Arbeit zu entwicklende spezielle Übersetzungsverfahren für das KeY-Projekt aufweisen soll.

Sei $\mathcal{D}$ ein beliebiges, aber festes UML-Modell. Sei $\Sigma_{\mathcal{D}}$ wieder die Menge der Symbole, die für die Formalisierung des UML-Modells $\mathcal{D}$ benötigt werden, und Σ^* die Signatur, die insgesamt von der Übersetzung verwendet wird.

An die Stelle der allgemeinen semantischen Strukturen $\mathcal{S}$ in den Definitionen bzw. Erläuterungen in den Abschnitten 2.2.2 bzw. 2.2.1 treten nun Zustände (bzw. Welten) in *speziellen* semantischen Strukturen, über denen **DL**-Formeln ausgewertet werden: Kripke-Strukturen $\mathcal{K}$, wie sie in [Bec01] beschrieben werden.

Für diese Strukturen gilt unter anderem, daß die Menge der möglichen Welten gerade aus den Zuständen des Systems besteht und die Übergangsrelationen zwischen den einzelnen Zuständen jeweils durch ein syntaktisch korrektes JAVA-CARD-Programm, die zugehörige Java-Semantik und die Interpretation der Methodenaufrufe (*atomare Programme*) festgelegt wird. Die einzelnen Zustände w werden prinzipiell als

[8] Die Korrektheit wurde nicht formal nachgewiesen!

prädikatenlogische Σ^*-Strukturen $\mathcal{S}_w = (M, I_w)$ über einem festen, für alle Zustände gleichen Universum M angesehen, deren Interpretation I_w alle Funktions- und Relationssymbole, die nicht benutzterdefiniert sind, entsprechend ihrer Intention immer in der gleichen Weise interpretiert und durch die Interpretation der Symbole zu Attributen, statischen Attributen und Assoziationen eine bestimmten Systemzustand festlegt. Die Menge der möglichen Welten aus der Kripke-Struktur ist außerdem abgeschlossen bezgl. der Übergangsrelation. Ein Zustand definiert jedoch neben den existierenden Objekten und ihren Attributbelegungen außerdem die Belegung aller Programmvariablen (insbesondere `this`).

Insofern ergibt sich aus einem Zustand w in einer Kripke-Struktur $\mathcal{K}$ eine entsprechende Instanziierung D des modellierten Systems (die nicht notwendigerweise gültig sein muß), desweitern erhält man aber aus der Interpretation der Programmvariablen auch eine korrespondierende Variablenbelegung β. Das heißt also, ein Zustand einer solchen Kripke-Struktur korrespondiert zu genau einem Paar (D, β) aus einer Instanziierung des modellierten Systems $\mathcal{D}$ und einer Variablenbelegung, wohingegen eine Instanziierung D zu mehreren Zuständen w der Kripke-Struktur korrespondiert; für alle diese Zustände gilt: Die $\Sigma_{\mathcal{D}}$-Einschränkungen der zugehörigen Σ^*-Strukturen $\mathcal{S}_w$ stimmen alle überein und beschreiben gerade die Instanziierung D.

Etwas genauer gesagt, betrachten wir einen Zustand w *nur dann als korrespondierend* zu einem Paar (D, β) aus einer Instanziierung D des UML-Modells $\mathcal{D}$ und einer Variablenbelegung β in D, falls gilt: Die Menge der in D existierenden Objekte einer Klasse `C` wird durch die Objekte der entsprechenden Kontextklasse C dargestellt, für die das boolsche Attribute `created` in der zu w gehörenden Struktur $\mathcal{S}_w$ durch `true` interpretiert wird.

Die Attributbelegungen der Objekte, sowie die Verknüpfung der einzelnen Objekte durch Links in D entsprechen der Interpretation der Symbole für diese Attribute und Assozationen über der Struktur $\mathcal{S}_w$. Insbesondere wollen wir fordern, daß für Funktionssymbole r_i, welche `0..1`-Assoziationen r in der Signatur $\Sigma_{\mathcal{D}}$ zu einem UML-Modell $\mathcal{D}$ darstellen, im 0-Fall – also wenn kein Objekt mit einem betrachteten Objekt o in D assoziiert ist – die Interpretation des Funktionssymbol für das betrachtete Objekt gerade den *null*-Wert zurückliefert, also $I_w(r)(o) = null$. *null* dient in diesem Fall zur eindeutigen Auszeichnung des 0-Falls und ist insofern unproblematisch, da UML/OCL kein *null*-Objekt kennt (*null* also *kein* Objekt ist, welches sonst bei der Beschreibung einer Instanziierung D eines UML-Modells verwendet werden müßte) und *null* ein Element *jeder* Sorte zu einem Modelltypen darstellt. Es eignet sich also in gewisser Weise als ,,Fehlerelement" bzw. ,,kein Objekt assoziiert"-Element[9].

Schließlich muß die Belegung der Programmvariablen[10] in D gerade der Variablenbelegung β entsprechen.

Wir wollen im folgenden von einem *gültigen* Zustand w sprechen, wenn w zu einer *gültigen* Instanziierung korresponiert.

[9] Man mache sich klar, daß die Interpretation von Funktionssymbolen in der **DL** – wie in den meisten anderen Logiksprachen L gleichermaßen – durch *totale* Funktionen geschieht. `0..1`-Assoziationen sind hingegen in gewisser Weise *partielle* Funktionen, die wir durch die Anwendung eines ,,Fehlerelement" (in der **DL**:*null*) simulieren können.

[10] In der **DL** werden logische Variablen und Programmvariablen unterschieden. Unser Verfahren zur Übersetzung in die **DL** wird jedoch für die freien Variablen, die in den OCL-Constraints vorkommen können, jeweils Programmvariable erzeugen. Insofern brauchen wir uns an dieser Stelle nicht um logische Variablen zu sorgen, da diese niemals frei in der erzeugten Formeln auftreten.

Eine Krikpe-Struktur $\mathcal{K}_\sigma$ entspricht also genau dann einem System $\sigma \in \mathsf{Systems}_\mathcal{D}$, wenn die Menge der möglichen Welten der Kripke-Struktur für jede Instanziierung $D \in \mathsf{States}_\sigma$ alle korrespondierenden Welt w_D (mit zugeordneter Σ^*-Struktur $\mathcal{S}_{w_D}$) enthält, jedes atomare Programm über $\mathcal{D}$ (Methodenaufruf einer Methode m) durch eine Interpretation in Form der zugehörigen abstrakten Implementierung $\rho_m \in \mathsf{StateTransitions}_\sigma$ über $\mathcal{D}$ behandelt, und den Anforderungen aus [Bec01] genügt. In diesem Fall wollen wir $\mathcal{K}_\sigma$ als eine zu dem System σ korrespondierende Kripke-Struktur (über $\mathcal{D}$) bezeichnen.

Wir passen nun die angegebenen Definitionen der Informationserhaltung für Invarianten und Methodenspezifikationen für das KeY-Projekt entsprechend an:

Definition 12 (Informationserhaltung für Invarianten)
Sei $\mathcal{D}$ ein beliebiges UML-Modell. Sei $\Sigma_\mathcal{D}$ die zugehörige Signatur aus der Formalisierung des Modells und $\Sigma^* \supseteq \Sigma_\mathcal{D}$ die Signatur, die von der Übersetzung insgesamt verwendet wird.

Eine Abbildung trans:$OCLConst_\mathcal{D} \to For^{\mathsf{DL}}_{\Sigma^*}$ heißt **informationserhaltend für Invarianten**, *falls gilt:*
Für eine beliebige Kripke-Struktur $\mathcal{K}_\mathcal{D}$ zu einem System σ über $\mathcal{D}$ mit $\mathsf{States}_\sigma = \mathsf{ValidStates}_\mathcal{D}$, jede Invariante I über dem Modell $\mathcal{D}$ und jede OCL-Algebra $\mathcal{M}^{OCL}_D$ zu einer gültigen Instanziierung D des Modells $\mathcal{D}$ gilt:

$$I \text{ ist wahr in der OCL-Algebra } \mathcal{M}^{OCL}_D$$
$$\textbf{gdw.}$$
$$\mathcal{K}_\mathcal{D}, w_D \models_{\mathsf{DL}} trans(I) \text{ für alle zu } D \text{ korrespondierenden Welten } w_D \text{ in } \mathcal{K}_\mathcal{D}$$

◁

Für die Definition der Informationserhaltung bzgl. Methodenspezifikationen lautet, brauchen wir nun lediglich die Definition 8 im Kontext von KeY neu definieren:

Definition 13 (Erfüllung von $trans(P)$ durch ρ auf gültigen Zuständen)
Sei $\mathcal{D}$ ein beliebiges UML-Modell, P eine beliebige Methodenspezifikation einer Methode m über $\mathcal{D}$ mit der Übersetzung $trans(P)$ und $\rho \subseteq \mathsf{StatePairs}_\mathcal{D}$ eine beliebige abstrakte Implementierung.

Die abstrakte Implementierung ρ **erfüllt die Übersetzung Methodenspezifikation P bzgl. der gültigen Zustandspaare von $\mathcal{D}$,** *falls gilt:*
Für eine beliebige Kripke-Struktur $\mathcal{K}_\mathcal{D}$ zu einem System σ über $\mathcal{D}$ mit $\mathsf{States}_\sigma = \mathsf{ValidStates}_\mathcal{D}$, die das atomare Programm zu einem Methodenaufruf von m durch die Einschränkung $\rho|_{\mathsf{ValidStates}_\mathcal{D}} = \rho \cap \mathsf{ValidStatePairs}_\mathcal{D}$ der abstrakte Implementierung ρ auf die Menge der gültige Zustandstandspaare interpretiert, jede OCL-Algebra $\mathcal{M}^{OCL}_D$ zu einer gültigen Instanziierung D des Modells $\mathcal{D}$ und alle zu D korrespondierenden Welten w_D in $\mathcal{K}_\mathcal{D}$ gilt:

$$\mathcal{K}_\mathcal{D}, w_D \models_{\mathsf{DL}} trans(P)$$

Wir schreiben in diesem Fall: $\mathsf{ValidStatePairs}_\mathcal{D}, \rho \models_L trans(P)$ ◁

Im Kontext des KeY-Projekts erachten wir Informationserhaltung bzgl. Vor/Nachbedingungen als relevant und wichtig. Wir definieren daher:

Definition 14 (Informationserhaltung bzgl. Vor-/Nachbedingungen)
Sei $\mathcal{D}$ ein beliebiges UML-Modell und $\Sigma_{\mathcal{D}}$ die zugehörige Signatur aus der Formalisierung des Modells. Sei $\Sigma^* \supseteq \Sigma_{\mathcal{D}}$ die Signatur, die von der Übersetzung insgesamt verwendet wird.

Eine Abbildung trans:$OCLConst_{\mathcal{D}} \to For^{\mathsf{DL}}_{\Sigma^*}$ heißt **informationserhaltend für Vor/Nachbedingungen**, *falls gilt:*
Für eine beliebige Kripke-Struktur $\mathcal{K}_{\mathcal{D}}$ zu einem System σ über $\mathcal{D}$ mit $\mathsf{States}_\sigma = \mathsf{ValidStates}_{\mathcal{D}}$ und jede Methodenspezifikation P über dem Modell $\mathcal{D}$ gilt: Wenn V die Vor- bzw. N die Nachbedingung in P und Th_V bzw. Th_N die zugehörigen Formeln aus der Übersetzung $trans(P)$ des Constraints sind, so sind, dann gilt für diese Größen:

Für alle OCL-Algebren $\mathcal{M}^{OCL}_D, \mathcal{M}^{OCL}_{D'}$ zu gültigen Instanziierungen D, D' von $\mathcal{D}$ und alle Variablenbelegungen β in $\mathcal{M}^{OCL}_D$ und β' in $\mathcal{M}^{OCL}_{D'}$ gilt:

$$\mathcal{M}^{OCL}_{D'}, \beta' \models V$$
gdw.
$$\mathcal{K}_{\mathcal{D}}, w_D \models_{\mathsf{DL}} Th_V \text{ für alle zu } D \text{ korrespondierenden Welten } w_D \text{ in } \mathcal{K}_{\mathcal{D}}$$

und

$$(\mathcal{M}^{OCL}_D, \beta), (\mathcal{M}^{OCL}_{D'}, \beta') \models N$$
gdw.
$$(\mathcal{K}_{\mathcal{D}}, w_D), (\mathcal{K}_{\mathcal{D}}, w_{D'}) \models_{\mathsf{DL}} Th_N \text{ für alle zu } D \text{ bzw. } D' \text{ korrespondierenden Welten } w_D \text{ bzw. } w_{D'} \text{ in } \mathcal{K}_{\mathcal{D}}$$

◁

Die Informationserhaltung für Diagramme wird nun festgelegt durch:

Definition 15 (Informationserhaltung für Diagramme)
Sei $Diagrams^{UML}$ die Menge aller UML-Klassendiagramme. Sei Σ eine feste, geeignete Signatur, die von der Abbildung insgesamt verwendet werden kann.

Eine Abbildung trans:$Diagrams^{UML} \to For^{\mathsf{DL}}_{\Sigma}$ heißt **informationserhaltend für Diagramme**, *falls gilt:*
Für jedes Diagramm $\mathcal{D}$, jede Kripke-Struktur $\mathcal{K}_{\mathcal{D}}$ zu einem System σ über $\mathcal{D}$, jede OCL-Algebra $\mathcal{M}^{OCL}_D$ zu einer beliebigen Instanziierung D des Diagramms $\mathcal{D}$ gilt:

Die OCL-Algebra $\mathcal{M}^{OCL}_D$ entspricht einer gültgen Instanziierung von $\mathcal{D}$
gdw.
$\mathcal{K}_{\mathcal{D}}, w_D \models trans(\mathcal{D})$ *für alle zu D korrespondierenden Welten w_D in $\mathcal{K}_{\mathcal{D}}$*

◁

Damit haben wir den allgemeinen Rahmen des zu entwickelnden Verfahrens abgesteckt und können uns den Details einer Übersetzung von OCL in eine der hier betrachteten Logiksprachen L – insbesondere **DL** – zuwenden.

2.3 Eine Basisabbildung von UML/OCL-Modellen in DL-Formeln.

Ein OCL-Constraint besteht im wesentlichen aus einem OCL-Ausdruck des Typs Boolean und einem Deklarationsteil, welcher diesen OCL-Ausdruck an eine Klasse im UML-Modell – dem sogenannten Kontextelement – bindet. Im Falle von Vor- und Nachbedingungen wird der Constraint sogar etwas spezifischer an eine Methode des Kontextelements gebunden.

Deshalb kann eine Abbildung von UML/OCL in eine Zielsprache die OCL-Constraints nicht als *isolierte* Größen betrachten, sondern muß auch das UML-Modell, über dem die Constraints formuliert wurden, in die Übersetzung miteinbeziehen.

Desweiteren wird deutlich, daß eine solche Übersetzung mit zwei unterschiedlichen syntaktischen Kategorien umgehen muß: OCL-Constraints auf der einen Seite und OCL-Ausdrücken auf der anderen.

Die folgenden Abschnitte werden diesen Anforderungen gerecht: Zunächst stellen wir die Sorten in der DL vor, die wir für die Darstellung der OCL-Typen benötigen, und erläutern kurz, wie die Sortenabbildung geschieht. Anschließend wenden wir uns dem gegebenen UML-Model zu und klären, wie das UML-Modell in der Logik formalisiert werden kann. Danach sind alle benötigten Grundlagen vorhanden, um die eigentliche Abbildung von OCL-Ausdrücken zu beschreiben, die den Hauptteil der zu entwickelnden Abbildung verkörpert. Diese Abbildung wird schließlich in dem darauffolgenden Abschnitt zu einer Abbildung für OCL-Constraints erweitert.

Restriktionen. Die hier vorgestellte Übersetzung unterliegt den folgenden Restriktionen:

- **Undefinierte OCL-Ausdrücke.** Wir betrachten nicht den Fall von *undefinierten* OCL-Ausdrücken, d.h. wir übersetzen einen gegebenen OCL-Ausdruck unter der Prämisse, daß dieser Ausdruck nicht zu undefiniert ausgewertet werden kann. Diese Einschränkung ist sicherlich gewichtig, vereinfacht aber die Aufgabe stark. Unsere Erfahrung in der Modellierung mit UML/OCL zeigt, daß die Undefiniertheit eines OCL-Constraints C nur in Randsituationen auftritt und meist durch eine sorgfältigere Formulierung des Constraints vermieden werden könnte, d.h. man einen anderen OCL-Constraint C' angegeben kann, der genau dann erfüllt ist (bzgl. eines Snapshots D des Klassendiagramms), wenn der ursprünglich OCL-Constraint C erfüllt ist, wobei C' selbst niemals zu undefiniert ausgewertet werden kann. Trotzdem kann man von einem Modellierer im allgemeinen nicht erwarten, daß er bei der Formulierung einer UML/OCL-Spezifikation die OCL-Constraints mit einer solchen Sorgfalt entwickelt. D.h. diese Einschränkung muß irgendwann aufgehoben werden, was aber den Rahmen dieser Arbeit sprengen würde. Wir haben dafür trotzdem ein klares Konzept, daß am Ende in Abschnitt 4.3.1 kurz erläutert wird.
- **Nicht unterstützte OCL-Typen.** Die folgenden OCL-Typen werden nicht unterstützt: Enumeration, OclState, OclExpression; das Weglassen dieser Typen stellt, wie in Abschnitt 2.3.1 dargelegt wird, keine besondere Einschränkung dar. Auch OclType wird – mit Ausnahme des `allInstances`-Features – nicht

unterstützt. Wir werden in diesem Fall aber in Abschnitt 4.3.2 auf eine mögliche Behandlung hinweisen.

Für Illustrationen und Beispiele verwenden wir im folgenden das UML-Modell aus Abbildung 2.1

2.3.1 Abbildung der Sorten

Bevor wir beginnen können, beliebige OCL-Ausdrücke in Formeln oder Term der **DL** zu transformieren, müssen wir die Abbildung der Sorten aus OCL in Sorten der **DL** angeben. Hierzu gehen wir die einzelnen Typen aus OCL schrittweise durch, einen Überblick findet man in Tabelle 2.1.

Benutzerdefinierte Typen aus dem UML-Diagramm $\mathcal{D}$. Jede Klasse C aus $\mathcal{D}$ wird auf eine Klasse C im Kontext der **DL** abgebildet. Für die Kontextklasse C wird dabei der gleiche Klassenbezeichner wie für C verwendet. Eine genauere Darstellung der Formalisierung des UML-Modells findet sich in Anschnitt 2.3.2.

OCL-Basistypen. Die Basistypen werden entsprechend er Semantik der Basistypen in OCL durch abstrakte Datentypen in der **DL** dargestellt.

Integer. In der **DL** gibt es keine direkte Entsprechung zum OCL-Typ Integer, da der OCL-Typ Integer das mathematische Konzept der ganzen Zahlen repräsentiert. Wir führen deshalb einen benutzerdefinierten abstrakten Datentyp INTEGER ein, der durch eine geeignete Standardaxiomatisierung (z.B. Peano-Axiome) spezifiziert wird. Die Signatur des ADTs umfaßt alle benötigten Eigenschaften des OCL-Typs Integer.

Real. Auch hier führen wir einen abstrakten Datentypen REAL ein, der durch geeignete Axiome beschrieben wird. Die Signatur des ADTs umfaßt alle benötigten Eigenschaften des OCL-Typs Real.

String. Wir benutzen für die Darstellung von Zeichenketten in der **DL** den ADT STRING, der alle erforderlichen Eigenschaften des OCL-Typs String durch entsprechende Symbole in seiner Signatur zur Verfügung stellt. Die Semantik dieser Symbole wird durch einen geeigneten Satz von Axiomen festgelegt.

Boolean. Da wir den Fall von undefinierten OCL-Ausdrücken gesondert behandeln und an dieser Stelle ausschließen, gibt es prinzipiell zwei Möglichkeiten, boolesche OCL-Ausdrücke zu übersetzen: Man kann eine logische Formeln verwenden, oder aber einen Term über der **DL**-Sorte `boolean`. Wir werden im folgenden die erste Variante vorziehen, in bestimmten Situation[11], aber auf die zweite Alternative ausweichen. Beide Darstellungsvarianten sind – wie später noch gezeigt wird – zu jedem Zeitpunkt ineinander überführbar.

[11] Eine solche Darstellung ist nur dann erforderlich, wenn wir durch einen Term über Kollektionen von booleschen Werten sprechen wollen

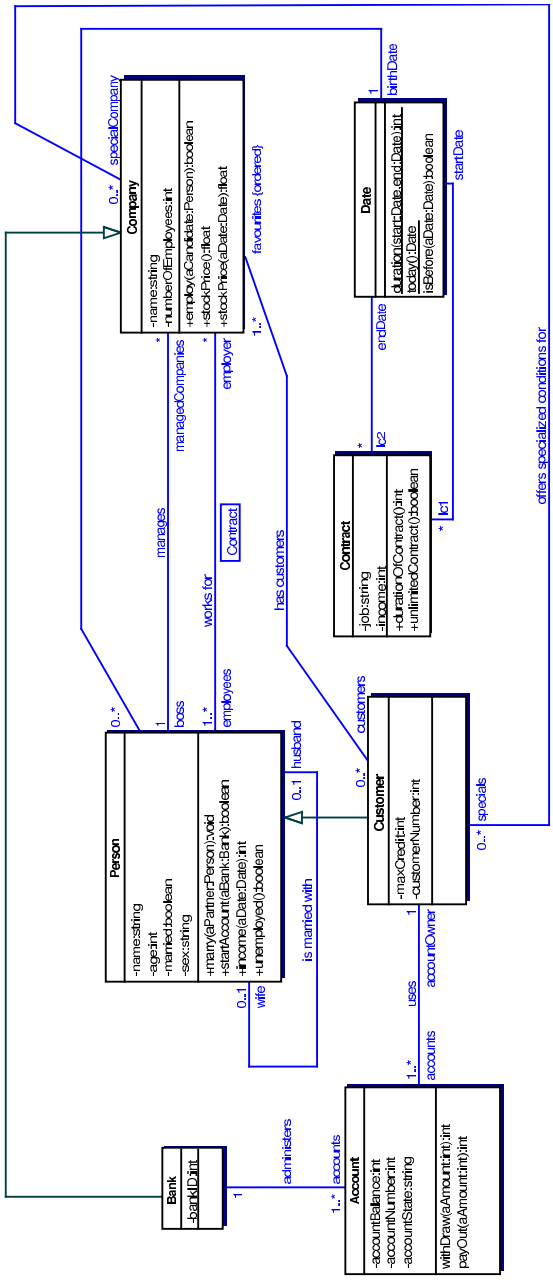


Abbildung 2.1: SimpleModel — Ein einfaches Beispiel eines Klassendiagramms.

Typen für Kollektionen. In OCL finden sich als Typen für Kollektionen Set(T), Bag(T), Sequence(T) und Collection(T). Diese Typen werden in OCL als abstrakte Datentypen betrachtet. Wir benutzen deshalb für die Übersetzung der drei konkreten Ausprägungen der Kollektionstypen und jede Elementsorte T jeweils einen entsprechenden ADT in der **DL**: Set_T, Bag_T und $Sequence_T$.

Zwar hat der OCL-Typ Collection(T) keine praktische Bedeutung, da er als abstrakter Obertyp der anderen Collection-Typen in OCL festgelegt ist und es somit keine Instanzen dieses Typs gibt, die nicht auch einem spezielleren Kollektionstypen angehören, doch wegen der Typhierarchie in OCL in Verbindung mit dem `if`-Konstrukt brauchen wir eine entsprechende Sorte in unserer Logik; wir verwenden zu diesem Zwecke einen ADT $Collection_T$ für jede Elementsorte T.

An dieser Stelle sei kurz erwähnt, daß eine Abbildung der Kollektionentypen auf **DL**-Arraytypen `T[]` deshalb ausscheidet, weil die Indexmenge der Arraytypen in der **DL** auf `byte` festgelegt ist und damit eine maximale Anzahl von 256 Elementen in den Kollektionen verbunden wäre, was der Semantik der OCL-Kollektionstypen als beliebig große Kollektionen keine Rechnung trägt. Zudem würde eine solche Darstellung nicht einem abstrakten Datentypen entsprechen, da ein solches Array in seinem Zustand veränderlich wäre.

Andere OCL-Typen. Der OCL-Typ Enumeration wird in Zukunft aus OCL wegfallen[12] und wurde deshalb in [Sch01b] nicht behandelt. Wir betrachten hier diesen Typen deshalb nicht.

Den OCL-Typen OclState werden wir ebenfalls nicht weiter behandeln, da die Unterstützung, die OCL bezüglich dieses Typs bietet, äußerst rudimentär ist. In [Sch01b] wird ebenfalls nicht genauer auf diesen Typ eingegangen.

Auch der Typ OclExpression wird hier nicht weiter behandelt, da er keinerlei praktische Bedeutung bei der Spezifikation von Softwaresystemen mittels UML/OCL hat. Dieser Typ wurde in OCL – wie es scheint – nur zum Zwecke der semiformalen Definition von OCL selbst aufgenommen.

Der OCL-Typ OclType bleibt im wesentlichen auch unbehandelt, da viele Eigenschaften dieses Typs in der Praxis nicht oft benötigt werden. In der zukünftigen UML-Version (2.0) scheint dieser Typ in OCL sogar nicht mehr vorgesehen zu sein. Deshalb gibt es keine entsprechende Sorte in der **DL**. Als einzige Ausnahme betrachten wir das `allInstances`-Feature; wir behandeln diese Eigenschaft ohne eine Sorte *OclType* in der Logik einführen zu müssen.

Abschließend bleibt noch der Typ OclAny, der als gemeinsamer Obertyp für alle OCL-Basistypen dient. Wir führen daher ein Sorte *Any* ein, die dementsprechend die Obersorte für alle Sorten in der **DL** darstellt. Man beachte, daß dieser OCL-Typ ebenfalls abstrakt ist und zur Definition von Eigenschaften dient, die allen Basistypen in OCL gemein sind. Instanzen dieses Typs haben immer auch einen spezielleren Typ. Die Sorte wird beispielsweise aufgrund des `if`-Konstrukts trotzdem benötigt.

2.3.2 Formalisierung eines UML-Modells $\mathcal{D}$

UML bietet eine Fülle von verschiedenen Diagrammtypen, um bestimmte Aspekte eines Systems zu beschreiben: Klassendiagramme, Zustandsübergangsdiagramme, Sequenzendiagramme, Interaktionsdiagramme, Verteilungsdiagramme und einige mehr.

[12] Dies belegt ein bereits akzeptierter *Issue* mit der Kennung #3134 bei der UML RTF (Revision Task Force). Demnach wird der Aufzählungstyp *Enumeration* durch UML-Klassen mit dem Stereotyp `<<enumeration>>` ersetzt.

OCL-Typ		zugeordneter **DL**-Typ
Modelltypen aus $\mathcal{D}$	Klasse C	**DL**-Kontextklasse C
OCL-Basistypen	Integer	ADT INTEGER
	Real	ADT REAL
	String	ADT STRING
	Boolean	**DL**-Formeln oder `boolean`
Typen für Kollektionen	Set(T)	ADT Set_T
	Bag(T)	ADT Bag_T
	Sequence(T)	ADT $Sequence_T$
	Collection(T)	ADT $Collection_T$
Andere OCL-Typen	Enumeration	*nicht behandelt*
	OclState	*nicht behandelt*
	OclExpression	*nicht behandelt*
	OclType	*nicht behandelt*
	OclAny	*Any*

Tabelle 2.1: Abbildung von OCL-Typen auf Sorten der **DL**

Aus diesem Potpourri verschiedener Darstellungstechniken kann der Modellierer diejenigen wählen, die für seine Zwecke notwendig sind.

Dabei sind Klassendiagramme sicherlich die bedeutendste und am weitesten verbreitete Darstellungstechnik. Sie beschreiben die statische Struktur des modellierten Systems mit all seinen möglichen Ausprägungen.

OCL, betrachtet als eine formale Sprache zur detaillierteren Modellierung von Eigenschaften eines Systems, die nicht durch graphische Notationen im Modell eingefangen werden können, unterstützt derzeit im wesentlichen nur Klassendiagramme[13]. Insofern beschränken wir uns im folgenden bei der Formalisierung von UML-Modellen auf *Klassendiagramme*.

Wenn man schließlich versucht ein Klassendiagramm zu formalisieren, dann entstehen erwartungsgemäß zwei Dinge: Zum einen Symbole, die es erlauben, über die Modellelemente im Klassendiagramm (Klassen, Attribute, Assoziationen usw.) zu sprechen, und zum anderen Formeln (oder andere Darstellungen semantischer Informationen), die die Interpretation dieser Symbole gemäß der Semantik von UML einschränken bzw. festlegen. Wir werden zunächst auf die entstehende Signatur $\Sigma_{\mathcal{D}}$ eingehen und anschließend Formeln generieren, die die Intention der Symbole in dieser Signatur auf der logischen Ebene einfangen.

2.3.2.1 Gewinnung einer Signatur aus einem Klassendiagramm

Klassen und Interfaces. Jede Klasse im Klassendiagramm beschreibt einen Typen, sowie eine Menge von Attributen und Methoden, die für diesen Typen definiert sind. Das gleiche gilt für Interfaces und andere Varianten von Klassen (beispielsweise abstrakte Klassen, Aufzählungsklassen etc.), da diese lediglich „besondere" Klassen

[13] Das einzige Konstrukt in OCL, daß eine andere Diagrammart betrifft, ist `oclInState` des OCL-Typs OclAny, das dem Modellierer erlaubt, auszudrücken, ein betrachtetes Objekt einer Klasse befinde sich in einem bestimmten Zustand der Zustandsmaschine, die der Klasse dieses Objekts zugeordnet ist. Diese Unterstützung von Zustandsdiagrammen betrachten wir als zu rudimentär, um nützlich zu sein.

im UML-Modell sind, denen eine spezielle Anwendungsemantik zufällt[14]. Wir behandeln deshalb alle verschiedenen Ausprägungen von Klassen prinzipiell auf die selbe Art und Weise: Sei C eine Klasse im Klassendiagramm $\mathcal{D}$. Dann erzeugen wir eine zugehörige Sorte C durch eine entsprechende Klassendefinition im Kontext der **DL**, die den gleichen Namen wie C besitzt. Jedes Attribut `a` (bzw. Methode `m`) wird auf ein gleichnamiges Attribut a (bzw. Methode m) abgebildet, wobei die Typen in der Signatur aus dem UML-Diagramm durch die entsprechenden **DL**-Sorten zu ersetzen sind. Man beachte, daß diese Attribute und Methode wie (nichtrigide) Funktionssymbole in der **DL** aufzufassen sind und lediglich anders notiert werden.

Beispiel 1
Betrachten wir die Klasse Person in unserem Beispielmodell.

Dann entsteht eine Kontextklasse $Person$ als Sorte in der **DL**.

Das Attribut `name:String` wird auf das Attribut $name{:}String$ der Kontextklasse $Person$ abgebildet.

Aus der Methode `income(aDate:Date):Integer` entsteht eine Methode $income(aDate{:}Date){:}Integer$ in der Kontextklasse $Person$.

Bei Aufzählungsklassen E – das bedeutet Klassen, die mit `<<enumeration>>` als Stereotypen versehen sind – besteht jedoch eine Besonderheit: die „Attribute" einer solchen Klasse stellen keine herkömmlichen Attribute dar, sondern lediglich Bezeichner für alle Literale, die der Aufzählungstyp zur Verfügung stellt. Insbesondere besitzen diese Literale keine explizite Typeangabe, da sie als Werte des entsprechenden Aufzählungstyp interpretiert werden.

Bei der Übersetzung gehen wir deshalb folgendermaßen vor: Wir erzeugen zu jedem Literal l ein gleichnamiges Attribut l in E, welches als Rückgabewert den Typ E selbst besitzt. Der speziellen Semantik der Literale wenden wir uns später in Abschnitt 2.3.2.2 zu.

Assoziationen. Wir beschränken uns bei der Behandlung von Assoziationen auf *binäre* Assoziationen, da diese am häufigsten in der praktischen Anwendung vorkommen.

Mehrstellige Assoziationen sind in ihrer Semantik komplexer und schwerer zu lesen – insbesondere die Multiplizitäten an den Enden der Assoziation – was ein Grund dafür sein mag, daß sie weitaus weniger oft bei der Modellierung benutzt werden. Unsere Einschränkung rechtfertig sich aber noch aus einem anderen Grund: Das CASE-Werkzeug, welches wir um Funktionalität zur formalen Beweisführung über UML/OCL-Modellen erweitern, unterstützt lediglich binäre Assoziationen. Das liegt hauptsächlich daran, daß dieses CASE-Werkzeug eine partielle, automatische Codeerzeugung aus einem UML-Modell anbietet und sogar die Synchronisation zwischen dem Modell und seiner Implementierung durchführt. Da es in keiner herkömmlichen Programmiersprache ein Konstrukt gibt, daß eine direkte Darstellung von mehrstelligen Assoziationen erlaubt, unterstützt dieses Werkzeug diese Form von Assoziationen nicht[15]. Die gleiche

[14] UML stellt hierzu den Mechanismus von *Stereotypen* zur Verfügung, die dazu dienen Modellelementen eine spezielle Anwendungssemantik zu verleihen. Ein Beispiel soll das verdeutlichen: Der Stereotyp `<<abstract>>` kennzeichnet eine *abstrakte* Klasse und verleiht der Klasse die spezielle semantische Eigenschaft, keine konkreten Ausprägungen zu besitzen, die diese Klasse als ihren speziellsten Typen aufweisen.

[15] Das ist eigentlich auch kein Problem, da mehrstellige Assoziationen vornehmlich in *Analysemodellen* eingesetzt werden. Diese Analysemodelle lassen sich dann Schrittweise hin zu einem *Implementierungsmodell* verfeinern, daß auf einfache Weise in eine Implementierungsprache übertragen werden

Situation findet sich sicher bei vielen CASE-Tools auf dem Markt, die gleichzeitig UML-Modellierungen und ihre Implementierung ermöglichen.

Ähnliches, gilt für *qualifizierte* Assoziationen, deren Behandlung wir deshalb auch nur skizzieren werden.

Sei nun r eine zweistellige Assoziation in $\mathcal{D}$ mit den Assoziationsenden e_0 und e_1, die die Klassen C_0 und C_1 verbindet. Jedem Assoziationsende e_i sei der Rollenbezeichner r_i zugeordnet $(i = 0, 1)$; wenn ein solcher Rollenname r_i nicht explizit angegeben wurde, so wird der Bezeichner der zugehörigen Klasse C_i – beginnend mit einem Kleinbuchstaben – verwendet.

Für jedes der beiden Assoziationsenden e_i wird nun ein Funktionssymbol in $\Sigma_{\mathcal{D}}$ eingeführt, welches den gleichen Namen wie der zugehörige Rollenbezeichner r_i trägt und zur Navigation über r in die Richtung des Assoziationsendes e_i dient.

Die Signatur hängt von der Multiplizität am Ende e_i ab: Ist diese höchstens eins, so besitzt die Funktion die Signatur $r_i : C_{1-i} \to C_i$, bei einer Multiplizität größer als eins erhalten wir $r_i : C_{1-i} \to Set_{C_i}$, es sei denn, daß e_i zusätzlich mit dem Stereotypen `<<ordered>>` markiert wurde und somit alle assoziierten Objekte unter einer gewissen Ordnung betrachtet werden. Wir benutzen in diesem Fall als Signatur $r_i : C_{1-i} \to Sequence_{C_i}$.

Sollte es sich bei r um eine rekursive Assoziation handeln, d.h $C_0 = C_1$, so legt die Namenskonvention von UML fest, daß die Rollennamen explizit angegeben und eindeutig sein müssen, um unterscheiden zu können, in welche Richting von einem betrachteten Objekt $o : C_i$ aus navigiert wird. Rekursive Assoziation können daher genauso behandelt werden, wie nicht-rekursive Assoziationen.

Beispiel 2
Betrachten wir die Beziehung *worksfor* zwischen den Klassen $Person$ und $Company$. Dann enthält $\Sigma_{\mathcal{D}}$ folgende zwei Funktionssymbole:

$$employer{:}Person \to Set_{Company}$$
$$employees{:}Company \to Set_{Person}$$

Aus der Assoziation *uses* zwischen den Klassen $Account$ und $Customer$ entstehen folgende Funktionssymbole in $\Sigma_{\mathcal{D}}$:

$$accounts{:}Customer \to Set_{Account}$$
$$accountOwner{:}Account \to Customer$$

Schließlich erzeugt die Assoziation *hasCustomers* zwischen den Klassen $Company$ und $Customer$ unter anderem folgendes Funktionssymbol in $\Sigma_{\mathcal{D}}$:

$$favourites{:}Customer \to Sequence_{Company}$$

kann. Auf dieser Ebene werden deshalb keine ,,komplexen" Konstrukte (die für Analysemodelle geeignet sind, bspw. mehrstellige Assoziationen oder Vererbungsbeziehungen zwischen Assoziationen) mehr vorkommen. Sie müssen im Zuge der Verfeinerung durch einfacherer Konstrukte (und zusätzlichen Informationen z.B. in OCL) aufgelöst werden. Das von uns erweiterte CASE-Tool spricht genau diese Ebene der Implementierungsmodelle an.

Bemerkung (Alternative Beschreibung von Assoziationen). Assoziationen verkörpern logische Beziehungen zwischen Objekten in der modellierten Miniwelt. Sie lassen sich daher in natürlicher Weise auch durch Relationssymbole darstellen, eine Sicht die an manchen Stellen möglicherweise verständlicher sein mag.

Die funktionale Übersetzung stattdessen beschreibt den Navigationscharakter von Assoziationen besser und kommt OCL in dem Sinne näher, als daß OCL im Grunde eine *funktionale* Sprache ist und eine Navigation somit als eine Funktionsanwendung auf ein betrachtetes Objekt betrachtet wird. Wir benutzen im folgenden nur die funktionale Darstellung von Assoziationen, eine Darstellung durch Relationen – oder aber eine gemischte Darstellung, die beide Alternativen benutzt – wäre jedoch leicht einzubauen.

Es sei trotzdem noch angemerkt, daß die alleinige Verwendung eines Relationssymbols für die Darstellung einer Assoziation r im allgemeinen *nicht* ausreicht: Wenn mindestens eines der Assoziationsenden e_i mit dem Stereotypen `<<ordered>>` markiert wurde, so wird die Ordnung der assoziierten Elemente, auf die der Stereotyp hinweist, nicht durch das Relationssymbol beschrieben. □

Beispiel 3
Betrachten wir wieder die Beziehung *worksfor* zwischen den Klassen $Person$ und $Company$. Dann könnte $\Sigma_{\mathcal{D}}$ auch folgendes Relationssymbol enthalten:

$$worksfor \subseteq Person \times Company$$

Bemerkung (n-stellige Assoziationen). Für n-stellige Assoziationen müßte man entsprechend ein Funktionsymbol zur Navigation von jeder beteiligten Klasse C_i zu den anderen beteiligten Klassen C_j $(i, j \in \{0, 1, \ldots, n\}$,
$i \neq j)$ in $\Sigma_{\mathcal{D}}$ einführen, d.h. $n(n-1)$ verschiedene Funktionssymbole.

Bei einer Repräsentation durch Relationssymbole würde dann ein n-stelliges Relationssymbole verwendet werden. □

Bemerkung (Qualifizierte Assoziationen). Der Fall von qualifizierten Assoziationen ist etwas komplizierter. In [JR99] wird die Semantik qualifizierter Assoziationen folgendermaßen beschrieben:

Während binäre Assoziationen ein Objekt auf eine Menge assoziierter Objekte abbilden, besteht bei qualifizierten Assoziationen zusätzlich die Möglichkeit, einen Wert (den *Qualifier*) anzugeben, der aus der Menge der assoziierten Objekte die Teilmenge auswählt, die durch diesen Wert „ausgezeichnet" sind. Man beachte, daß der Wert nicht notwendigerweise für die Navigation verwendet werden muß, und die angegebene Multiplizität sich auf den Fall bezieht, daß ein qualifizierender Wert angegeben wurde.

Im allgemeinen ist diese Größe als ein *Assoziationsattribut* zu betrachten, d.h. implizit der Assoziation zugeordnet.

Wir könnten also folgendermaßen vorgehen: Wir erzeugen zwei Funktionssymbole in $\Sigma_{\mathcal{D}}$, je eines für den qualifizierten bzw. den nichtqualifizierten Fall. Für den qualifizierten Fall enthält die Signatur des Symbols (gegenüber dem nichtqualifizierten Fall) einen zusätzlichen Parameter für den qualifizierenden Wert[16]. Die Multiplizitäten wir-

[16] Im Falle von mehreren Qualifizierenden Werten bedeutet das je einen zusätzlichen Parameter für jeden solchen Wert.

ken sich für den qualifizierten Fall genauso wie oben beschrieben auf die Ergebnissorte des zugehörigen Funktionssymbols aus, wohingegen im nichtqualifizierten Fall immer die Multiplizität $*$ angenommen wird.

Zusätzlich wird ein weiteres Funktionssymbol eingeführt, das zu einem Paar von assoziierten Objekten (das entspricht in etwa einem sogenannten *Link*) den zugehörigen qualifizierenden Wert[17] liefert. Dieses Symbol kann dann in Abschnitt 2.3.2.2 zur Beschreibung der Semantik der Funktionssymbole, die zur Navigation dienen, verwendet werden. □

Assoziationsklassen. Assoziationsklassen sind Klassen im Diagramm $\mathcal{D}$, denen eine spezielle Semantik zufällt: Sie beschreiben die konkrete Beziehung zwischen einem beliebigen Paar (allgemeiner: Tupel) von assoziierten Objekten genauer, indem sie spezielle Informationen kapseln, die eine konkrete *Beziehung* betreffen. Diese Informationen werden gewöhnlich durch Attribute der Assoziationsklasse dargestellt.

Betrachten wir nun eine Assoziation r unter den selben Bedingungen wie bei der Formalisierung von Assoziationen weiter oben. An diese Assoziation sei desweiteren eine Assoziationsklasse A geknüpft, die bei der Formalisierung der Klassen auf eine gleichnamige **DL**-Sorte A abgebildet wurde.

Neben der Möglichkeit der Navigation zwischen den beiden assoziierten Klassen C_0 und C_1, bietet UML/OCL in diesem Fall zusätzlich die Möglichkeit, von einem Objekt einer der Klassen C_i zu den zugehörigen Instanzen der Assoziationsklasse A zu navigieren, die die konkreten Beziehungen (Links) genauer beschreiben, an denen diese Objekt teilnimmt. Zusätzlich ist es möglich, von einer solchen Instanz der Assoziationsklasse A, zu jedem einzelnen der durch den zugehörigen Link verknüpften Objekte zu navigieren.

Für den ersten Fall führen wir für jede Klasse C_i ein einstelliges Funktionssymbol a in $\Sigma_{\mathcal{D}}$ ein, das als Bezeichner den Namen der Assoziationsklasse (beginnend mit einem kleinen Anfangsbuchstaben) und als Parametersorte C_i verwendet. Die Ergebnissorte des Funktionssymbols hängt in der gleichen Weise, wie das Funktionsymbols r_{1-i}, welches für die Navigation über r zur gegenüberliegenden Klasse C_{1-i} erzeugt wurde, von der Multiplizität des gegenüberliegenden Assoziationsendes e_{1-i} ab, d.h. also A im Falle einer Multiplizität von höchstens eins, Set_A bei einer Multiplizität größer eines, sofern das Assoziationsende nicht mit dem Stereotypen `<<ordered>>` versehen wurde, und $Sequence_A$ in allen anderen Fällen.

Für den zweiten Fall führen wir für jede Klasse C_i ein Funktionssymbol r_i in $\Sigma_{\mathcal{D}}$ ein, das als Bezeichner den zugehörigen Rollenname r_i und als Signatur $r_i : A \to C_i$ verwendet. Man beachte an dieser Stelle, daß eine Instanz der Assoziationsklasse einen Link (also eine konkrete Beziehung zwischen Objekten) beschreibt, und deshalb bei der Navigation von einer solchen Instanz zu einem der assoziierten Objekte niemals eine Menge entstehen kann.

Nun gilt es, noch eine Besonderheit zu beachten: Falls r eine rekursive Assoziation sein sollte (also $C_0 = C_1$), dann gibt es bei unserer Konstruktion in Fall 1 ein Problem: Es ist dann aus dem Namen und der Signatur der Symbole nicht mehr klar, in welche Richtung ein Funktionssymbol navigiert.

[17] Dieser existiert immer für ein solches Paar, da nach der Semantik einer qualifizierten Assoziation, jedes Element der Zielklasse für *genau einen* qualifizierenden Wert in der entsprechenden qualifizierten Menge assoziierter Objekte vorkommt. Die qualifizierenden Werte partitionieren somit die Menge der Instanzen der Zielklasse.

OCL selbst fordert an dieser Stelle, daß bei solchen Navigationen zu Assoziationsklassen über rekursive Beziehungen die Richtung explizit dadurch angegeben wird, daß dem Namen der Assoziationsklasse der Rollenname des Assoziationsendes (in eckigen Klammern) angefügt wird, in dessen Richtung wir navigieren. Aus diesem Grund verwenden wir dieselbe Bezeichnungstechnik für die beiden Funktionssymbole in Fall 1, d.h. die Bezeichner der Symbole r_i entsprechen dem Namen der Assoziationsklasse A (beginnend mit einem kleinen Anfangsbuchstaben), gefolgt von dem Rollenbezeichner r_{1-i} (in eckigen Klammern) des Assoziationsendes e_{1-i}, in dessen Richtung navigiert wird.

Beispiel 4
Betrachten wir die Beziehung *worksfor* zwischen den Klassen $Person$ und $Company$ und die zugehörige Assoziationsklasse $Contract$. Dann enthält $\Sigma_{\mathcal{D}}$ folgende zwei Funktionssymbole für die Navigation nach $Contract$:

$$contract{:}Person \rightarrow Set_{Contract}$$
$$contract{:}Company \rightarrow Set_{Contract}$$

Für die Navigation von der $Contract$ aus gibt es in $\Sigma_{\mathcal{D}}$ außerdem:

$$employer{:}Contract \rightarrow Company$$
$$employees{:}Contract \rightarrow Person$$

Bemerkung (Eindeutigkeit der Funktionssymbole). An dieser Stelle sei bemerkt, daß die Namenskonventionen, die UML verwendet, sicherstellt, daß die Definition der oben eingeführten Funktionsymbole wohldefiniert ist, d.h. es gibt keine zwei verschiedene Funktionssymbole mit gleichem Namen und gleicher Signatur. □

2.3.2.2 Extraktion von semantische Informationen aus einem Klassendiagramm

Wir versuchen in diesem Abschnitt, Informationen, die die Interpretation der in Abschnitt 2.3.2.1 generierten Symbole betreffen und mit der Semantik von UML zu tun haben, durch formale Mittel wie z.B. Formeln darzustellen, um sie für das Deduktionssystem verfügbar zu machen.

Als Ergebnis erhalten wir unter anderem eine Menge $Th_{\mathcal{D}}$ von Formeln, die festlegt, daß die erzeugten Symbole gemäß der UML Semantik interpretiert werden. Diese Formeln verkörpern *nichtlogische* Axiome dar und kapseln Wissen über bzw. Anforderungen an die $\Sigma_{\mathcal{D}}$-Strukturen $\mathcal{S}_D$, die zu beliebigen – nicht notwendigerweise gültigen – Instanziierungen D von $\mathcal{D}$ korrespondieren.

Klassen und Interfaces. Wir haben bisher aus einer Klasse C im Diagramm $\mathcal{D}$ eine Kontextklasse C als Sorte in der **DL** erzeugt, die prinzipiell alle Attribute und Operationen aus C enthält. Dies geschah durch Erzeugung einer entsprechenden Klassendeklaration für C im Kontext.

Nun gibt es bestimmte Attribute und Methoden mit einer speziellen Semantik, denen wir durch dieses Vorgehen noch nicht in ausreichendem Maße gerecht wurden: statische Attribute und Methoden sind nicht an ein bestimmtes Objekt gebunden sind, sondern an die zugehörige Klasse selbst.

Günstigerweise bietet Java gerade für diese Situation ein Schlüsselwort, welches wir für die Deklaration des entsprechenden Attributs bzw. der entsprechenden Methode verwenden wollen: `static`.

Durch dieses Vorgehen haben wir ebenfalls noch nicht die *spezielle* Semantik erfaßt, die mit den verschiedenen Varianten von Klassen einhergeht: Für Interfaces (Stereotyp `<<interface>>`) und abstrakte Klassen (Stereotyp `<<abstract>>`) heißt das beispielsweise, daß alle Instanzen dieser Klasse auch einen spezielleren Typen in der Klassenhierarchie aufweisen.

Da wir in der **DL** die Möglichkeiten der Klassendefinition aus Java ausnutzen können, gibt es für diese Fälle sogar ein syntaktisches Konstrukt in der Klassendefinition, mit dem diese Semantik festgehalten werden kann: Für Interfaces benutzen wir bei der Klassendefinition das Schlüsselwort `interface`, wohingegen bei abstrakten Klassen das Schlüsselwort `abstract class` zur Klassendeklaration zum Zuge kommt.

Für Aufzählungsklassen **E** gibt es kein angemessenes Schlüsselwort. Wir werden deshalb versuchen, über Formeln die Intention der eingeführten Attribute als Literale des Aufzählungstyps zu fassen:

Zunächst sind die Literale immer, d.h unabhängig von der Existenz irgend eines Objektes des Aufzählungstyps verfügbar. Sie sind an den Typen gebunden und somit statisch, was wir durch das Schlüsselwort `static` in den Deklarationen der entsprechenden Attribute für die einzelnen Literale ausdrücken.

Desweiteren handelt es sich um Literale, d.h. durch unterschiedliche Bezeichner werden auch unterschiedliche Größen beschrieben. Deshalb generieren wir für je zwei unterschiedliche (Literal)attribute $l_1, l_2{:}E$ der Sorte E folgende Formel in $Th_{\mathcal{D}}$, die deren Verschiedenheit ausdrückt:

$$\neg(E.l_1 \doteq E.l_2)$$

Zuguterletzt handelt es sich bei den angegebenen Literalen $l_1, \ldots, l_n$ um *alle* möglichen Werte dieses Typs E. Deshalb formulieren in $Th_{\mathcal{D}}$ wir zusätzlich:

$$\dot{\forall} e{:}E\ (e \doteq E.l_1 \vee \ldots \vee e \doteq E.l_n)$$

Bemerkung (Punktierte Quantoren). Wir werden im folgenden – wie auch in der gerade genannten Formel – häufig Gebrauch von *punktierten* Quantoren $\dot{\forall}$ bzw. $\dot{\exists}$ machen und deshalb die Intention, die hinter diesen Quantoren steht, kurz erläutern: Ein punktierter Quantor entspricht in seiner Semantik dem entsprechenden nichtpunktierten Quantor, wobei der Quantifizierungsbereich *eingeschränkt* ist, falls der entsprechende Bereich die Extension einer Kontextklasse C der **DL** darstellt, man also über Objekte quantifiziert. Genauer gesagt, entspricht der Quantifizierungsbereich eines punktierten Quantors über einem Objektuniversum allen in dem betrachteten Zustand *existierenden* Objekten mit der Ausnahme der Objekts *null*.

Anders gesprochen entspricht beispielsweise die Formel

$$\dot{\forall} e{:}C\ (\Phi)\ \textit{bzw.}\ \dot{\exists} e{:}C\ (\Phi)$$

(für eine Kontextklasse C) folgender Formel ohne punktierten Quantor:

$$\forall e{:}C\ (((e.\mathit{created} \doteq \texttt{true}) \wedge \neg(e \doteq \mathit{null})) \rightarrow \Phi)\ \textit{bzw.}$$
$$\exists e{:}C\ ((e.\mathit{created} \doteq \texttt{true}) \wedge \neg(e \doteq \mathit{null}) \wedge \Phi)$$

Für Universen, die keine Klassenextensionen verkörpern, entspricht die Semantik des punktierten Operators genau der Semantik des nicht-punktierten Operators.

Die Verwendung eines solchen *neuen* Quantors anstelle einer simplen Expansion der entsprechenden Abkürzung hat insbesondere folgende Vorteile:

Zum einen sind die entstehenden Formeln wesentlich kompakter und leichter zu lesen, was der Verständlichkeit der vorgestellten Abbildung zu gute kommt, zum anderen bleibt die Abbildung insgesamt allgemeiner und ist in gewisser Weise leichter auf andere Zielsprachen zu übertragen, da durch die Benutzung des punktierten Quantor keine syntaktischen Konstrukte verwendet werden, die ausschließlich in der **DL** vorhanden sind. Für eine gewünschte Zielsprache muß dann nach der Abbildung eines Contraints, die entstehende Formel nachbearbeitet und damit bereinigt werden, um eine mit der Zielsprache konforme Formel zu erhalten, welche die Semantik der punktierten Quantoren in geeigneter Weise in dieser Zielsprache auflöst.

Eine andere Alternative wäre beispielsweise die Erweiterung der Zielsprache um die punktierten Quantoren, wobei die formale Behandlung der Semantik dieser Quantoren dann auch in den Kalkülregeln eines verwendeten Deduktionssystems geschehen könnte. □

Bemerkung (Schlüsselworte vs. Formeln). Bei der gerade erläuterten Formalisierung semantischer Eigenschaften der generierten Symbole in $\Sigma_{\mathcal{D}}$ benutzen wir desöfteren Schlüsselworte der Programmiersprache in der **DL**, die es in den meisten anderen Logiksprachen sicher nicht gibt. Ein Formalisierung ist sicherlich ohne weiteres auch ohne Verwendung dieser Schlüsselworte möglich; das gewählte Vorgehen reduziert aber die Anzahl der Formeln in $Th_{\mathcal{D}}$ durch Verwendung der Semantik bestimmter Schlüsselworte der Programmiersprache in der **DL**.

Wir wollen einige Alternativen hier kurz skizzieren: Die Eigenschaft von Interfaces und abstrakten Klassen, nur Instanzen zu besitzen, die auch einem spezielleren Typen angehören, läßt sich folgendermaßen durch eine Formel beschreiben: Sei I ein Interface in $\mathcal{D}$ und C1,..., CN alle direkten Unterklassen des Interface I. Desweiteren seien $I, C_1, \ldots, C_n$ die entsprechenden Sorten in der **DL**. Dann lautet die gewünschte Formel:

$$\dot{\forall} i{:}I\,(\dot{\exists} o{:}C_1\,(o \doteq i) \vee \ldots \vee \dot{\exists} o{:}C_n\,(o \doteq i))$$

Die Verwendung des Schlüssewortes `static` für Attribute und Methoden einer Klasse C kann zum Beispiel durch die Verwendung von Funktionssymbolen (anstelle von Attributen und Methoden der Sorte C) ersetzt werden, wobei diese Symbole im nicht-statischen Falle immer einen zusätzlichen Parameter $o{:}C$ besitzen, der angibt, für welches Element das Attribut bzw. die Methode ausgewertet wird, und im statischen Fall, dieser Parameter gerade fehlt. Attribute würden also im nicht-statischen Fall als einstellige Funktionssymbole modelliert, wohingegen im statischen Fall eine Konstante verwendet würde.

Ähnliche Alternativen gibt es in den anderen Fällen ebenso, d.h. wir glauben, daß sich ein UML-Klassendiagramm $\mathcal{D}$ im wesentlichen durch eine Logiksprache erster Stufe beschreiben läßt. □

Vererbungsbeziehungen. Bisher haben wir im Abschnitt 2.3.2.1 lediglich die Klassen aus dem Diagramm einzeln übersetzt. Das Diagramm $\mathcal{D}$ definiert aber im allgemeinen eine Typhierarchie in Form von Vererbungsbeziehungen zwischen den einzelnen

Klassen. Diese Information formalsieren wir wieder direkt durch Verwendung entsprechender Schlüsselworte bei den einzelnen Klassendefinitionen:

Sei zwischen zwei beliebigen Klassen `C1` und `C2` eine Vererbungsbeziehung in $\mathcal{D}$ angegeben, wobei `C1` die Oberklasse sei. Dann wird in der zugehörigen Klasse C_2 bei der Deklaration eine entsprechende Vererbungsbeziehung über das Schlüsselwort `extends` festgehalten. Handelt es sich bei der Vererbungsbeziehung spezieller um eine Implementierungsbeziehung (Stereotyp `<<implements>>`), so verwenden wir stattdessen das Schlüsselwort `implements`.

Bemerkung (Mehrfachvererbung). Mehrfachvererbung wird in unserem Falle nicht betrachtet, da im KeY-Projekt ausschließlich UML-Modelle zu Software-Systemen, die in Java realisiert werden, behandelt werden, und Mehrfachvererbungen in Java jedoch nicht vorgesehen sind.

Man beachte, daß Mehrfachimplementierungen hingegen erlaubt sind und entsprechend der oben geschilderten Methode über eine längere Aufzählung von Schnittstellen $I_1, \ldots, I_k$ hinter dem Schlüsselwort `implements` in einer Klassendefinition behandelt werden können. □

Assoziationen. Bei der Formalisierung einer binären Assoziation r zwischen den Klassen C_i $(i = 0, 1)$ wurden je ein Funktionssymbol r_i für die Navigation in die entsprechende Richtung in $\Sigma_{\mathcal{D}}$ eingeführt. Diese zwei Symbole sind aber bisher formal nicht verknüpft, im Gegenteil, sie sind syntaktische Gebilde, die nichts miteinander zu tun haben. Diese Mißstand wollen wir durch eine entsprechende Formel beseitigen, die die Interpretation der beiden Funktionsymbole geeignet verknüpft, und damit im eigentlichen Sinne festlegt, daß die beiden Symbole r_i gemeinsam die Assoziation r wiederspiegeln.

Die Symbole r_i haben zunächst folgende Signatur: $r_i : C_{1-i} \to Set_{C_i}$ oder $r_i : C_{1-i} \to Sequence_{C_i}$. Dann lautet die notwendige Formel:

$$\dot{\forall} o_0{:}C_0 \dot{\forall} o_1{:}C_1\, (o_1 \in r_1(o_0) \leftrightarrow o_0 \in r_0(o_1))$$

Sollte eines der Symbole r_i als Ergebnistyp keinen Sorte zu einer Kollektion besitzen, dann wird die zugehörige $\in$-Beziehung in obiger Gleichung durch eine $\doteq$-Beziehung ersetzt.

Beispiel 5
Betrachten wir die Beziehung *worksfor* zwischen den Klassen $Person$ und $Company$. Dann enthält $Th_{\mathcal{D}}$ folgende Formel:

$$\dot{\forall} p{:}Person \dot{\forall} c{:}Company\, (p \in employees(c) \leftrightarrow c \in employer(p))$$

Aus der Assoziation *uses* zwischen den Klassen $Account$ und $Customer$ entsteht in $Th_{\mathcal{D}}$ andererseits:

$$\dot{\forall} a{:}Account \dot{\forall} c{:}Customer\, (a \in accounts(c) \leftrightarrow c \doteq accountOwner(a))$$

Würde an dieser Stelle zur Darstellung der Assoziation *worksfor* zwischen den Klassen $Person$ und $Company$ zusätzlich ein Relationssymbole $worksfor \subseteq Person \times Company$ in $\Sigma_{\mathcal{D}}$ benutzt, so mußte man außerdem noch folgende Formel einführen:

$$\dot{\forall} p{:}Person \dot{\forall} c{:}Company\, (p \in employees(c) \leftrightarrow worksfor(p, c))$$

Bemerkung (Qualifizierte Assoziationen). Bei qualifizierten Assoziationen ist an dieser Stelle zu beachten, daß für jeden qualifizierenden Wert, die Menge der ausgezeichneten assoziierten Objekte eine Teilmenge der Objekte bilden, die wir bei Navigation *ohne* Nutzung der Qualifikation erhalten würden.

Die Mengen, die für beliebige qualifizierende Werte entstehen, bilden sogar eine Partition der Menge der assoziierten Objekte (ohne Qualifikation), d.h. jedes assoziierte Objekt der Zielklasse kommt für genau einen Qualifikator in der entsprechenden qualifizierten Menge vor. □

Assoziationsklassen. Wir betrachten wieder die Situation von oben: Sei r eine Assoziation zwischen den Klassen C_i $(i = 0, 1)$, an die eine Assoziationsklasse A gebunden wurde.

Für die Formalisierung von Assoziationsklassen wurden syntaktisch insgesamt vier Funktionssymbole erzeugt: Für jede der assoziierten Klassen C_i beinhaltet $\Sigma_{\mathcal{D}}$ je ein Paar von Symbolen für das Navigieren von und zu der Assoziationklasse A.

Ähnlich wie für die zwei Funktionssymbole, die zur Formalisierung der Assoziation r selbst eingeführt wurden, müssen wir zunächst die Funktionssymbole, die zu einem dieser Paar gehören, verknüpfen: Ein solches Paar bestehe aus den Symbolen $a{:}C_i \to Set_A$ und $r_i{:}A \to C_i$. Wir wollen dann ausdrücken, daß es möglich ist, zwischen jedem Paar von assoziierten Objekten (der Assoziationsklasse A und der beteiligten Klasse C_i) zu hin und her zu navigieren:

$$\dot{\forall} a{:}A \dot{\forall} o{:}C_i \, (a \in a(o) \leftrightarrow o \doteq r_i(a))$$

Hat das gegenüberliegende Assoziationsende r_{1-i} eine Multiplizität nicht größer als eins, so ist in dieser Formel die $\in$-Beziehung durch eine $\doteq$-Beziehung zu ersetzen.

Nachdem wir jetzt die beiden Paare von Symbolen jeweils untereinander verknüpft haben, müssen wir sie schließlich noch miteinander bzw. mit dem Paar von Funktionssymbolen für Darstellung der Assoziation r selbst verbinden:

$$\dot{\forall} o_0{:}C_0 \dot{\forall} o_1{:}C_1 \, (o_1 \in r_1(o_0) \leftrightarrow \dot{\exists} a{:}A \, (a \in a(o_0) \wedge r_1(a) \doteq o_1))$$
$$\dot{\forall} o_0{:}C_0 \dot{\forall} o_1{:}C_1 \, (o_0 \in r_0(o_1) \leftrightarrow \dot{\exists} a{:}A \, (a \in a(o_1) \wedge r_0(a) \doteq o_0))$$

oder alternativ:

$$\dot{\forall} a{:}A \dot{\forall} o_0{:}C_0 \dot{\forall} o_1{:}C_1 \, ((o_0 \doteq r_0(a) \wedge o_1 \doteq r_1(a)) \to o_1 \in r_1(o_0))$$
$$\dot{\forall} o_0{:}C_0 \dot{\forall} o_1{:}C_1 \, (o_1 \in r_1(o_0) \to \dot{\exists} a{:}A \, (r_0(a) \doteq o_0 \wedge r_1(a) \doteq o_1))$$

Auch hier gilt: Hat eines der Assoziationsenden e_i eine Multiplizität nicht größer als eins, so ist in diesen Formeln die entsprechende $\in$-Beziehung durch eine $\doteq$-Beziehung zu ersetzen.

Zuguterletzt müssen wir noch modellieren, daß für jeden *Link* zwischen zwei Instanzen o_0, o_1 der Klassen C_0, C_1 genau eine entsprechende Instanz a der Assoziationsklasse existiert, die diesem Link zugeordnet ist. Da die obigen Formeln die Existenz einer solchen Instanz a der Assoziationsklasse im Falle eines Links zwischen den Instanzen o_0, o_1 ausdrücken, müssen wir lediglich noch sicherstellen, daß es nicht mehr als eine solche Instanz der Assoziationklasse gibt, die diesem Link zugeordnet ist, also

$$\begin{aligned}\dot{\forall} o_0{:}C_0 \dot{\forall} o_1{:}C_1 \, (o_1 \in r_1(o_0) \to \\ (\dot{\forall} a, a'{:}A \, ((r_0(a) \doteq o_0 \wedge r_1(a) \doteq o_1) \wedge \\ (r_0(a') \doteq o_0 \wedge r_1(a') \doteq o_1) \to a \doteq a'))\end{aligned}$$

Man beachte, daß die Formel

$$\dot{\forall} a, a' {:} A\,(((r_0(a) \doteq r_0(a')) \wedge (r_1(a) \doteq r_1(a'))) \rightarrow a \doteq a')$$

nicht die gleiche Aussage liefert, da die Eindeutigkeit des Assoziationsobjektes nur bzgl. eines *Links* erfüllt sein muß, nicht aber, wenn das Assoziationsobjekt an gar keinem Link teilnimmt!

Beispiel 6
In unserem Beispielmodell ist der Assoziation *worksfor* zwischen $Person$ und $Company$ eine Assoziationklasse $Contract$ zugeordnet. Für die formale Darstellung der Navigation von und zu der Assoziationsklasse wurde in $\Sigma_{\mathcal{D}}$ folgende Funktionssymbole erzeugt:

$$\begin{aligned} &contract{:}Person \rightarrow Set_{Contract} \\ &employees{:}Contract \rightarrow Person \\ &contract{:}Company \rightarrow Set_{Contract} \\ &employer{:}Contract \rightarrow Company \end{aligned}$$

Für die Darstellung der Assoziation r selbst zudem:

$$\begin{aligned} &employees{:}Company \rightarrow Set_{Person} \\ &employer{:}Person \rightarrow Set_{Company} \end{aligned}$$

Man beachte, daß der Name von Funktionssymbolen überladen sein kann. Die einzelnen Symbole sind dann jedoch aufgrund der unterschiedlichen Signatur unterscheidbar!

Diese werden nach dem geschilderten Verfahren verknüpft:

$$\begin{aligned} &\dot{\forall} ct{:}Contract \dot{\forall} p{:}Person\,(ct \in contract(p) \leftrightarrow p \doteq employees(ct)) \\ &\dot{\forall} ct{:}Contract \dot{\forall} c{:}Company\,(ct \in contract(c) \leftrightarrow c \doteq employer(ct)) \\ &\dot{\forall} p{:}Person \dot{\forall} c{:}Company\,(c \in employer(p) \leftrightarrow \\ &\qquad \dot{\exists} ct{:}Contract\,(ct \in contract(p) \wedge employer(ct) \doteq c)) \\ &\dot{\forall} p{:}Person \dot{\forall} c{:}Company\,(p \in employees(c) \leftrightarrow \\ &\qquad \dot{\exists} ct{:}Contract\,(ct \in contract(c) \wedge employees(c) \doteq p)) \\ &\dot{\forall} p{:}Person \dot{\forall} c{:}Company\,(p \in employees(c) \rightarrow \\ &\qquad (\dot{\forall} ct, ct'{:}Contract\,((employer(ct) \doteq c \wedge \\ &\qquad\qquad employees(ct) \doteq p) \wedge \\ &\qquad\qquad (employer(ct') \doteq c \wedge \\ &\qquad\qquad employees(ct') \doteq p) \rightarrow ct \doteq ct')) \end{aligned}$$

Multiplizitäten. Schließlich wenden wir uns den Multiplizitäten zu. Sie verkörpern ein Hilfsmittel, um die Anzahl der Objekte, die mit einem bestimmten Objekt in Beziehung stehen, genauer zu spezifizieren. Im allgemeinen werden sie durch eine natürliche Zahl `n` oder ein Intervall aus natürlichen Zahlen `n..m` notiert ($n, m \in \mathbb{N}_0 \cup \{*\}$, $n \leq m$). Im ersten Fall bedeutet die Multiplizität, daß zu dem betrachteten Objekt immer genau n Objekte der assoziierten Klasse in Beziehung stehen, wohingegen im zweiten Fall immer mindestens n, aber höchstens m Objekte der assoziierten Klasse mit betrachteten Objekt verbunden sind. Das Literal $*$ steht insbesondere für beliebig viele Objekte.

Sei nun r wieder eine Assoziation wie bisher. Zusätzlich sei das Assoziationsende e_i mit der Multiplizität $n_i..m_i$ versehen[18]. Wir erzeugen in $Th_{\mathcal{D}}$ dann folgende Formel[19]:

- Fall $n_i = m_i, 1 < m_i, m_i \neq *$:

$$\dot{\forall} o{:}C_{1-i}\,(size(r_i(o)) \doteq n_i)$$

- Fall $n_i = m_i = *$: Hier ist die Semantik der Kardinalität bereits durch die Signatur des Funktionssymbols r_i vollständig abgedeckt.

- Fall $n_i \neq m_i, 1 \leq n_i, m_i \neq *$:

$$\dot{\forall} o{:}C_{1-i}\,(n_i \leq size(r_i(o)) \wedge size(r_i(o)) \leq m_i)$$

- Fall $n_i \neq m_i, n_i = 0, m_i \neq *$:

$$\dot{\forall} o{:}C_{1-i}\,(size(r_i(o)) \leq m_i)$$

- Fall $n_i \neq m_i, 1 \leq n_i, m_i = *$:

$$\dot{\forall} o{:}C_{1-i}\,(n_i \leq size(r_i(o)))$$

- Fall $n_i = 0, m_i = *$: Hier ist die Semantik der Kardinalität bereits durch die Signatur des Funktionssymbols r_i vollständig abgedeckt.

- Fall $n_i = 0, m_i = 1$: Hier ist die Semantik der Kardinalität bereits durch die Signatur des Funktionssymbols r_i vollständig abgedeckt.

- Fall $n_i = 1, m_i = 1$: Hier kommt eine spezielle Eigenschaft der von uns betrachteten Strukturen $\mathcal{S}_D$ bzw. Welten w_D der Kripke-Struktur $\mathcal{K}_{\mathcal{D}}$ in der **DL** zum tragen, die zu einer Instanziierung D von $\mathcal{D}$ korrespondieren: Da wir in der **DL** nur *totale* Funktionen betrachten (also insbesondere für die Symbole für die Navigation über Assoziationen), im Falle von `0..1`-Assoziationen aber im 0-Fall im Prinzip eine *partielle* Funktionen benutzt werden müßte, simulieren wir die partiellen Funktionen durch totale Funktionen, die in ihrer Bildmenge ein „Fehlerelement" bzw. „kein assoziiertes Objekt vorhanden"-Element haben können. Da es sich in unserem speziellen Fall der `0..1`-Assoziationen bei den beteiligten Sorten stets um Kontextklassen handelt, können wir als Fehlerelement *null* verwenden. Die hier notwendige Eigenschaft der betrachteten Strukturen bzw. Welten besteht nun darin, daß ein entsprechendes Funktionssymbol für eine `0..1`-Assoziation im 0-Fall das Objekt *null* zurückliefert und somit anzeigt, daß zu dem betrachteten Objekt *kein* Objekt assoziiert ist.

  Wir können also bei der gegebenen Kardinalität ($n_i = 1, m_i = 1$) ausschließen, daß kein Objekt assoziiert ist, und benutzen deshalb[20]:

$$\dot{\forall} o_{1-i}{:}C_{1-i}\dot{\exists} o_i{:}C_i\,(r_i(o_{1-i}) \doteq o_i)$$

[18] Wird eine Multiplizität der Form `n` verwendet, so entspricht das einer Abkürzung von `n..n`

[19] Man beachte folgende notationelle Vereinfachung in den Formeln: Wir bezeichnen hier der Lesbarkeit wegen die Übersetzung von Literalen n_i, m_i natürlicher Zahlen genauso mit n_i, m_i, obwohl es sich um logisch unterschiedliche Sachverhalte handelt!

[20] Der punktierte Quantor $\dot{\exists}$ als Abkürzung für $\exists o : C((o.created \doteq \texttt{true} \wedge \neg o \doteq null) \wedge \Phi)$ in $\dot{\exists} o : C(\Phi)$ ist hier wesentlich!

Ist an die betrachtete Assoziation eine Assoziationsklasse A gebunden, so müssen prinzipiell die selben Formeln für die Funktionssymbole $a{:}C_i \to Set_A$ (bzw. $a{:}C_i \to Sequence_A$ oder $a{:}C_i \to A$) aus $\Sigma_{\mathcal{D}}$ generiert werden, die die Navigation von und zur Assoziationsklassen beschreiben. Dazu werden in dem entsprechenden Fall in der obigen Aufzählung die Klasse C_i durch A und das Funktionsymbol r_i durch das zugehörige Funktionssymbol a (für die Navigation zu A in Richtung des Assoziationsendes e_i) ersetzt. Für die Symbole r_i zur Navigation von der Assoziationklasse A in die beteiligten Klassen C_i wird außerdem festgehalten[21]:

$$\dot{\forall} a{:}A\,(\dot{\exists} o_{1-i}{:}C_{1-i}\,(a \doteq a(o_{1-i})) \to \dot{\exists} o_i{:}C_i(r_i(a) \doteq o_i))$$

Beispiel 7
Betrachten wir abermals die Beziehung *worksfor* im Beispielmodell, an die die Assoziationklasse C gebunden wurde.

$\Sigma_{\mathcal{D}}$ enthält also folgende Funktionssymbole für die formale Darstellung der Navigation von und zu der Assoziationsklasse:

$$\begin{array}{l} contract{:}Person \to Set_{Contract} \\ employees{:}Contract \to Person \\ contract{:}Company \to Set_{Contract} \\ employer{:}Contract \to Company \end{array}$$

, sowie für die Darstellung der Assoziation r selbst:

$$\begin{array}{l} employees{:}Company \to Set(Person) \\ employer{:}Person \to Set(Company) \end{array}$$

Für die Erfassung der Multiplizitäten generieren wir:

$$\begin{array}{l} \dot{\forall} c{:}Company\,(1 \leq size(employees(c))) \\ \dot{\forall} c{:}Company\,(1 \leq size(contract(c)) \\ \dot{\forall} ct{:}Contract \dot{\exists} p{:}Person\,(employees(ct) \doteq p) \\ \dot{\forall} ct{:}Contract \dot{\exists} c{:}Company\,(employer(ct) \doteq c) \end{array}$$

Betrachten wir andererseits die Assoziation *uses* zwischen $Account$ und $Customer$, dann enthält $Th_{\mathcal{D}}$ folgende Formeln:

$$\begin{array}{l} \dot{\forall} c{:}Customer\,(1 \leq size(accounts(c))) \\ \dot{\forall} a{:}Account \dot{\exists} c{:}Customer\,(accountOwner(a) \doteq c) \end{array}$$

Bemerkung (Assoziationen als Relationssymbole). Man beachte, daß die Formalisierung der Multiplizitäten im allgemeinen wesentlich komplizierter würde, wenn man zur Darstellung der Assoziationen in der Logik nur Relationssymbole in $\Sigma_{\mathcal{D}}$ heranziehen würde. Man denke beispielsweise an eine Multiplizität `4..32`. □

Bemerkung (Qualifizierte Assoziationen). Bei qualifizierten Assoziationen ist an dieser Stelle zu beachten, daß die angegebenen Multiplizitäten nur für den Fall gelten, daß ein qualifizierender Wert zur Navigation verwendet wird. Andernfalls wird

[21] Wieder ist die punktierte Form des Existenzquantors wesentlich!

die Multiplizität `0..*` verwendet. Deshalb müßte nach dem oben geschilderten Verfahren lediglich Formeln für das Funktionssymbol r_i generiert werden, welches den qualifizierten Fall der Navigation erfaßt. □

Bemerkung (Attributschreibweise für Funktionssymbole). Wir werden im folgenden für die eingeführten (nichtrigiden) Funktionssymbole jeweils eine Notation verwenden, wie wir sie für Attribute in Klassen kennen, also einen Funktionsterm $f(o, p_1, \ldots, p_n)$ in der Form $o.f(p_1, \ldots, p_n)$ schreiben. Diese andere Schreibweise soll ausschließlich bewirken, daß die die entstehenden Formeln und Terme später der Notation von OCL näherstehen und somit sich die Lesbarkeit dieser Formeln und Term für den Modellierer erhöht wird. □

Beispiel 8
Betrachten wir wieder die Beziehung *worksfor* zwischen den Klassen $Person$ und $Company$ und die zugehörige Assoziationsklasse $Contract$.

Anstatt der Formel

$$\dot{\forall} a{:}Account \dot{\forall} c{:}Customer\ (a \in accounts(c) \leftrightarrow c \doteq accountOwner(a))$$

schreiben wir dann nun

$$\dot{\forall} a{:}Account \dot{\forall} c{:}Customer\ (a \in c.accounts \leftrightarrow c \doteq a.accountOwner)$$

Bemerkung (Übertragbarkeit in andere Logiksprachen L). Die bisherigen Ausführungen sollten unter anderem klar machen, daß die vorgestellte Formalisierung von UML-Modellen $\mathcal{D}$ sich leicht in andere Logiken, z.B. Prädikatenlogiken erster Stufe oder ihre Erweiterungen, übertragen läßt, da wir im wesentlichen Gebrauch von Funktionssymbolen machen, und die notwendigen Eigenschaften der Symbole im Rahmen prädikatenlogischer Ausdrucksmittel festhalten konnten. □

Bemerkung (Weitere graphische Notationen in UML). UML bietet – gerade in Form von Stereotypen – eine Fülle von weiteren graphischen Notationen, die in einem Klassendiagramm angewendet werden können und die wir hier nicht weiter betrachten werden.

Wir haben jedoch die wichtigsten Modellelemente und Notationen für Klassendiagramme, die tatsächlich in der Praxis oft verwendet werden, behandelt. Wir sind der Meinung, daß auch weitere solcher semantischen Informationen mittels Formeln in unserem Ansatz einzubetten sind. Andererseits gibt es möglicherweise auch Notationen, deren Semantik sich einer Formalisierung entzieht. Ob diese Semantik dann für irgend eine Eigenschaft, die in der Praxis relevant ist und mittels eines Deduktionssystems überprüft werden soll, eine Rolle spielt, bleibt abzuwarten. □

2.3.3 Abbildung von OCL-Ausdrücken in die DL

Wir entwickeln in diesem Abbschnitt eine Basisversion einer Abbildung *beliebiger* OCL-Ausdrücke in Terme bzw. Formeln der **DL** und skizzieren zunächst kurz die entscheidenden Grundideen, die hinter dieser Übersetzung stehen.

Unser Verfahren benutzt prinzipiell *Terme* über geeigneten abstrakten Datentypen, um OCL-Ausdrücke darzustellen. Eine gewisse Sonderrolle nehmen dabei die OCL-Ausdrücke des OCL-Typs Boolean ein: Sie werden – sofern das möglich ist – in logische *Formeln* abgebildet. Man gewinnt durch diese Darstellung über Formeln in gewisser Weise ein notationelle Vereinfachung, auf der anderen Seite erhält man (im Vergleich zu Termen) Objekte einer anderen syntaktischen Kategorie, was in wenigen Randfällen nicht problemlos durchzuführen ist.

Die Abbildung arbeitet rekursiv auf der syntaktischen Struktur des zu übersetzenden OCL-Ausdrucks. Bei der Übersetzung bestimmter OCL-Features bzw. Konstrukte, wie sie weiter unten detailliert dargelegt wird, werden in manchen Fällen *neue* Funktions- bzw. Relationssymbole eingeführt, sowie geeignete Axiome, die die Interpretation dieser Symbole entsprechend der Semantik von UML/OCL einschränken. Wir erhalten dadurch bei der Übersetzung eines OCL-Ausdrucks e aus $OCLExp_{\mathcal{D}}$ einen (oder mehrere) Term(e) t_e (und/oder Formeln Φ_e) über einer Erweiterung Σ^* der Signatur $\Sigma_{\mathcal{D}}$, die durch die Formalisierung des UML-Modells $\mathcal{D}$ in Abschnitt 2.3.2.1 festgelegt wurde.

Insbesondere wenden wir die folgende *Benennungstechnik* desöfteren an: Um einen gewissen, durch einen OCL-Ausdruck `e` (explizit) beschriebenen Sachverhalt S der modellierten Miniwelt (beispielsweise eine bestimmte Menge von Objekten), der sich *nicht explizit* durch Aggregation von Termen aus Übersetzungen von Subausdrücken von `e` in Form eines (komplexeren) Terms in der **DL** darstellen läßt, zu repräsentieren, führen wir ein neues Symbol s_e in Σ^* ein, welches diesen Sachverhalt S verkörpert.

Genauer gesagt fungiert das neu eingeführte Symbol s_e als ein *Name* bzw. ein *Bezeichner* für den zu charakterisierenden Sachverhalt S der Miniwelt. Um sicherzustellen, daß dieser Bezeichner s_e in der erwünschten Weise interpretiert wird, generieren wir zusätzlich geeignete Formeln (*Axiome*), die dann *implizit* die semantischen Eigenschaften von S einfangen. Man beachte, daß der in OCL beschriebene Sachverhalt S von der Belegung *freier* Variablen (beispielsweise `self`) im OCL-Ausdruck `e` abhängen kann und damit der Term, welcher den selben Sachverhalt in der Logik verkörpern soll, in gleicher Weise von *Parametern* abhängen muß, d.h. s_e ist im allgemeinen keine Konstante, sondern ein mehrstelliges Funktions- oder Relationszeichen.

Wir verwenden zur übersichtlicheren und kompakteren Darstellung die folgenden notationellen Konventionen bei der weiteren Beschreibung der Abbildung:

Wie gerade erläutert, besteht die Übersetzung eines OCL-Ausdrucks `exp` im allgemeinen aus zwei Teilen: Einem Term oder einer Formel, die den durch `exp` verkörperten Sachverhalt in der Logik benennbar macht, sowie einer Menge von (nichtlogischen) Axiomen, die möglicherweise während der Übersetzung neu eingeführte Symbole in Σ^* semantisch definiert. Diesen ersten Teil bezeichnen wir durch $\lceil$`exp`$\rceil$, den zweiten durch $Ax_{\texttt{exp}}$. Die Menge $Ax_{\texttt{exp}}$ der Axiome, die bei der Übersetzung des OCL-Ausdrucks `exp` entstehen, besteht entsprechend der rekursiven Vorgehensweise des Verfahrens aus allen Axiomen, die aus der Übersetzung eines Subausdrucks $\texttt{e}_i$ von `exp` stammen, sowie den Axiomen, die für die Übersetzung der OCL-Eigenschaft notwendig sind, welche sich auf der obersten Ebene des zu `exp` gehörenden Syntaxbaumes befindet. Wir bezeichnen schließlich die **DL**-Sorte, die dem OCL-Typen T zugeordnet ist, mit T.

Abschließend bleibt noch eine Kleinigkeit anzumerken: Bei der Implementierung der Abbildung, die auf der Parserkomponente des OCL-Compilers [HDF00] der TU Dresden aufbaut, machen wir Gebrauch von der Möglichkeit der syntaktischen *Normalisierung* von OCL-Ausdrücken, *bevor* die eigentliche Übersetzungsprozedur angewendet wird. Diese Normalisierungen haben den Sinn, einen beliebigen OCL-Ausdruck e in einen semantisch äquivalenten Ausdruck e' zu wandeln, der bestimmte syntaktische Einschränkungen erfüllt und damit zu einem echten Teilfragment der Sprache $OCLExp_{\mathcal{D}}$ gehört. Wir können daher diese Einschränkungen bei der eigentlichen Übersetzung als gegeben annehmen und somit das Abbildungsverfahren vereinfachen.

Wir benutzen dabei von den mit der Parserkomponente mitgelieferten Normalisierungen die folgenden:

- `CollectShorthandExpansion` — Expandiert die in OCL erlaubte Abkürzung für das `collect`-Feature und macht die implizite Anwendung des `collect`-Operators explizit.
- `IteratorInsertion` — Fügt bei iterierenden Features in OCL eine Iteratordeklaration ein, sofern diese nicht schon vorhanden ist.
- `DefaultContextInsertion` — Fügt das Kontextelement, für das ein OCL-Ausdruck (z.B. bei iterierenden OCL-Features wie `select`) ausgewertet wird, explizit ein.
- `ConstraintNaming` — Führt eindeutige Namen für die Benennung von OCL-Constraints ein.
- `TypeInformationInsertion` — Fügt die Typinformation bei der Deklaration von Iterator-Variablen explizit ein.
- `VariableClarification` — Sorgt für die eindeutige Variablenbezeichner innerhalb der zu übersetzenden Constraints.

Zusätzlich haben wir folgende Normalisierungsschritte selbst entwickelt:

- `KeyMultipleIteratorSolving` — Sorgt für die Auflösung mehrfacher Iterator-Variablen bei der Verwendung von iterierenden OCL-Features. Die entsprechenden Ausdrücke werden durch geeignete geschachtelte Ausdrücke ersetzt, die jeweils nur noch eine Iterator-
  Variable verwenden.
- `InstanceAsSetNormalizer` — In OCL ist es möglich, eine einzelne Instanz (zum Beispiel ein Objekt) als eine Menge mit genau diesem Element zu benutzen. Dafür gibt es folgende spezielle Notation: Sei `o` ein OCL-Ausdruck der eine beliebige Instanz eines OCL-Typs `T` beschreibt und `setFeature` eine beliebige Eigenschaft des Typs `Set(T)` in OCL. Dann wird durch die Pfeilnotation in `o->setFeature` angedeutet, daß das betrachtete Objekt als Einermenge betrachtet wird (anstatt als das Objekt selbst!). Wir lösen diese *implizite* Wandlung durch eine *explizite* Mengendefinition auf, um unser rekursives Verfahren in einem einheitlichen Rahmen zu gestalten: `o->setFeature` wird ersetzt durch den OCL-Ausdruck `Set{o}->setFeature`.

Zwischen den einzelnen Normalisierungen gibt es Abhängigkeiten in der Form, daß die Anwendung eines bestimmten Normalisierungsschrittes n die Anwendung anderer

Normalisierungschritte n_i zuvor erforderlich machen kann. Diese Abhängigkeiten sind etwas detaillierter in [Fin00] beschrieben.

Nun sind alle Grundlagen gelegt und wir können uns schließlich der eigentlichen Abbildung an sich zuwenden.

2.3.3.1 Grundlegendes

Variable. Die Übersetzung einer OCL-Variablen `v` des Typs `T` entspricht einer logischen Variablen v (mit demselben Bezeichner) des Typs T, d.h. $\lceil \texttt{v} \rceil = v$.

Bemerkung (DL und Programmvariablen). In der **DL** wollen wir für die freien Variablen aus dem zu übersetzenden *Constraint*, also dem Kontextelement (meist `self`) und der Parametern `p1, ..., pN` einer Methode (im Falle von Methodenspezifikationen) prinzipiell eine *spezielle* Form von Variablen verwenden: *Programmvariable*. In der **DL** wird zwischen logischen und Programmvariablen unterschieden. Logische Variablen entsprechen den herkömmlichen Variablen aus klassischen Prädikatenlogiken, ihr Wert wird durch eine Variablenbelegung β festgelegt und ist insbesondere unabhängig von dem betrachteten Zustand w_D der Kripke-Struktur $\mathcal{K}_\mathcal{D}$.

Programmvariable verhalten sich dagegen wie modale Konstanten, können also verschiedene Wert in unterschiedlichen Zuständen annehmen. Über sie kann jedoch *nicht* quantifiziert werden.

Um die nachfolgende Darstellung einfacher, einheitlicher und leichter übertragbar in andere Zielsprachen L anzugeben, verzichten wir an dieser Stelle bei Übersetzung auf die Verwendung von Programmvariablen und verwenden somit immer logische Variablen.

Ein an die **DL** *angepaßtes* Verfahren kann man prinzipiell dann dadurch erhalten, daß in den generierten Formeln $trans(C)$ für einen Constraint C in einem auf die Übersetzung durch das hier dargestellte Verfahren *nachfolgenden* Bereinigungsschritt die logischen Variablen zum Kontextelement ($self$) und den Methodenparametern $(p_1, \ldots, p_n)$ jeweils durch gleichnamige und gleichsortige Programmvariablen ersetzt werden, und alle Quantoren, die eine dieser Variablen als Quantorvariable verwenden, wegfallen lassen. Außerdem benötigen wir bei allen eingeführten Funktionssymbolen, die eine solche Variable durch ihr freies Vorkommen in einem zugehörigen Teilausdruck als Parameter verwenden, diesen Parameter nicht mehr und lassen diesen ebenfalls wegfallen. Die generierten Formeln sind dann entsprechend anzupassen. Wie im folgenden deutlich wird, können diese Variablen nur für Allquantoren als Quantorvariable auftreten. Die *explizite* Allquantifikation über alle Werte wir in der **DL** *implizit* über die Belegung der Programmvariablen in allen zu einer Instanziierung D korrepsondierenden Welten w_D der Kripke-Struktur $\mathcal{K}_\mathcal{D}$ nachgebildet. □

Literale. Literale `l` eines OCL-Typs `T` werden im allgemeinen auf Terme t_l über einem entsprechenden ADT T abgebildet:

- Boolean — Die Literale `true` und `false` werden durch die atomaren Formeln *true* bzw. *false* in der Logik dargestellt. Falls in dem betrachteten Kontext für diese Literale die Formeldarstellung von booleschen OCL-Ausdrücken jedoch nicht zulässig sein sollte, so verwenden wir stattdessen die Konstanten `true` bzw. `false` der **DL**-Sorte `boolean`.

- Integer — Für die Darstellung ganzer Zahlen in Dezimaldarstellung bietet der ADT INTEGER ein entsprechendes (einstelliges) Funktionssymbol $z{:}Integer \to Integer$ für jede Ziffer $\mathtt{z} \in \{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}\}$, sowie ein Konstantensymbol #, das die ganze Zahl Null darstellt. Ziffernfolgen für Integer-Literale werden dann durch eine Funktionsschachtelung (beginnend mit dem Nullsymbol dargestellt).

  Beispiel 9
  Die Übersetzung des Literals `1234` ergibt sich zu

  $$\lceil \mathtt{1234} \rceil = 4(3(2(1(\#))))$$

  Diese Darstellung mag zunächst etwas befremdend erscheinen, doch eine entsprechende Implementierung der graphischen Schnittstelle kann diese spezielle Schreibweise für Literale ganzer Zahlen vor dem Benutzter verbergen.

  Eine solche Unterstützung einer Zifferndarstellung erlaubt es einem Deduktionssystem nun durch entsprechende Beweisregeln[22] mit Zifferndarstellungen ganzer Zahlen umzugehen und gewissermaßen mittels Beweisregeln symbolisch zu rechnen.

  Wir bezeichnen im folgenden der Übersichtlichkeit wegen die Übersetzung eines Integer-Literals `n` mit dem gleichen Bezeichner n in der Logik. Der Leser denke sich an den entsprechenden Stellen den entsprechenden Term expandiert.

- Real — Ein Literal `r` für eine reelle Zahl wird auf eine Konstante r des ADT REAL abgebildet, die den gleichen Bezeichner trägt.

  Diese Darstellung besitzt nicht die Vorteile einer Zifferndarstellung; man muß daher im Deduktionssystem geeignete Mechanismen entwickeln, um mit den generierten Konstanten umzugehen. Dies ist langfristig auch für die ganzen Zahlen notwendig, da man in einem Beweis, der mit solchen Literalen zu tun hat, nicht immer eine Rechnung detailliert (in Form von Beweisregelanwendungen) ausgeführt haben möchte, sondern eher eine Art „Taschenrechner" wünschenswert wäre. Ein Entscheidungsverfahren für (entscheidbare) Fragmente der Arithmetik (wie z.B. Presburger Arithmetik) wären denkbar. Diese Frage wird gerade im Rahmen einer Diplomarbeit an diesem Institut untersucht und ist nicht Teil dieser Arbeit.

  Die naive Darstellung mittels Konstanten hat aber einen großen Vorteil: Da die konkrete Darstellung und Behandlung der Literale eigentlich unabhängig von der Übersetzung an sich ist, kann durch einfaches Nachbearbeiten der entstehenden Formeln, im Nachhinein eine gewünschte Repräsentation festgelegt werden. Durch einfaches Ersetzten der Literal-Konstanten kann die Abbildung an die entsprechenden Bedürfnisse angepaßt werden.

- String — Entsprechend dem ADT INTEGER zur Darstellung der ganzen Zahlen benutzten wir auch für den ADT STRING zur Darstellung von Zeichenketten eine „Zifferndarstellung": Für jeden Buchstaben `c` gibt es eine entsprechende Konstante c über dem ADT STRING. Mit einer weiteren Konstanten *emptyString* für das leere Wort, sowie einer Funktion *append* zum Anhängen von Zeichen an eine Zeichenkette lassen sich alle möglichen Zeichenkettenliterale darstellen.

[22] Diese Regeln beschreiben das Rechnen mit der Dezimaldarstellung ganzer Zahlen, das jeder Schüler in der Grundschule lernt.

Beispiel 10
Die Übersetzung des Literals `Ocl` ergibt sich zu

$$\lceil \texttt{Ocl} \rceil = append(append(append(emptyString, O), c), l)$$

Auch hier bezeichnen wir der Übersichtlichkeit wegen die Übersetzung eines String-Literals `s` ebenfalls mit dem gleichen Bezeichner s in der Logik. Der Leser denke sich an den entsprechenden Stellen den entsprechenden Term expandiert.

- Kollektionsliterale — In OCL werden Kollektionsliterale normalerweise durch eine vollständige Aufzählung der Elemente der Kollektion angegeben. In einem solchen Fall generieren wir einen entsprechenden Term über einem geeigneten ADT Set_T (bzw. Bag_T oder $Sequence_T$):

  Seien $\texttt{e}_i$ $(i = 1, \ldots, n)$ die OCL-Ausdrücke, die die Elemente der durch das Literal bezeichneten Kollektion spezifizieren, und T_i die zugehörigen OCL-Typen. Sei zudem T der kleinste gemeinsame Obertyp[23] der OCL-Typen T_i. Dann stellen wir den entsprechenden Term über dem ADT Set_T (bzw. Bag_T oder $Sequence_T$) durch Funktions-Schachtelung mit einer Funktion *insert* zum Einfügen von Elementen in eine Kollektion und einer Konstanten $emptySet_T$ (bzw. $emptyBag_T$ oder $emptySequence_T$) für die entsprechende leere Kollektion dar, also für Mengen beispielsweise:

  $$\lceil \texttt{Set\{1,2,3\}} \rceil = insert(insert(insert(emptySet_{\mathsf{Integer}}, 1), 2), 3)$$

  Sollte eines der $\texttt{e}_i$ ein boolscher OCL-Ausdruck sein, der durch eine Formel $\Phi_{\texttt{e}_i}$ übersetzt wurde, so wandeln wir an dieser Stelle die Darstellung durch eine Formel in eine entsprechende Termdarstellung über der Sorte `boolean` – das notwendige Vorgehen wird in Abschnitt 2.3.3.2 genau erklärt. In dem oben angegebenen Funktionsterm verwenden wir dann für das Einfügen des Elements $\texttt{e}_i$ gerade einen Term $t_{\texttt{e}_i}(q_1, \ldots, q_k)$, wobei $q_1, \ldots, q_k$ genau die freien Variablen aus $\Phi_{\texttt{e}_i}$ seien.

  Es bleibt aber noch eine weitere Besonderheit zu beachten: OCL erlaubt keine geschachtelten Kollektionen, sondern verwendet stattdessen verschmolzene bzw. *geflattete* Kollektionen. Sollte deshalb einer der Elementausdrücke $\texttt{e}_i$ einen Kollektionstypen (mit dem Elementtyp T' haben), so verwenden wir für die Berechnung des Typs T der Ergebniskollektion als das entsprechende T_i gerade den Elementtyp T' der Kollektion und an der entsprechenden Stelle in der Funktionschachtelung das Symbol *union* für das Vereinigen der entsprechenden Kollektionen statt des Symbols *insert* für das Einfügen des durch $\texttt{e}_i$ beschriebenen Elements.

  OCL bietet zudem für Kollektionen aus Integer-Werten[24] die Möglichkeit, diese Kollektion mittels eines Kollektionsliterals unter Angabe eines Intervalls zu beschreiben.

  Seien also `e1`,`e2` beliebige OCL-Ausdrücke des Typs Integer.

  Dann definiert das Literal `CollType{e1..e2}` eine Kollektion der Art `CollType` $\in \{Set, Bag, Sequence\}$, die genau die Elemente in dem angegeben Intervall enthält.

[23] Man beachte die Bemerkung auf Seite 63.

[24] Für den Typ Real besteht diese Möglichkeit in OCL *nicht*, da ansonsten *unendliche* Kollektionen in OCL entstehen könnten und somit der `iterate`-Operator und damit alle durch `iterate` definierten Operator auf einer solchen Kollektion nicht terminieren würden, d.h. undefiniert wären.

Für Multimengen gilt genauer, daß die Elemente genau einmal in der Kollektion vorkommen, und für Sequenzen gilt außerdem, daß die Elemente in der Sequenz entsprechend der Ordnung $<$ (auf Integer-Werten) aufsteigend geordnet sind.

Wir wollen das nun formalisieren: Da die gegebenen Grenzen des Intervalls *beliebige* OCL-Ausdrücke des Typs Integer sein können, also insbesondere keine Literale sein müssen, können wir die Elemente der definierten Kollektion nicht explizit aufzählen (und nach obigem Verfahren vorgehen). Wir benutzen deshalb wieder unsere Benennungstechnik auf folgende Weise:

Seien $p_1, \ldots, p_n$ genau die freien Variablen aus den Übersetzungen der Intervallgrenzen $\lceil$e1$\rceil$, $\lceil$e2$\rceil$ und $i, j : T$ neue Variable, die nicht in $\lceil$e1$\rceil$ oder $\lceil$e2$\rceil$ vorkommen.

- $E =$ Set{e1..e2}
  Wir führen in Σ^* ein neues Symbol set_E ein und definieren als Übersetzung des Literals $\lceil$E$\rceil = set_E(p_1, \ldots, p_n)$. Die Eigenschaften dieses Symbols werden außerdem durch folgendes Axiom in Ax_E festgelegt:

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, \forall i{:}T \, (i \in set_E(p_1, \ldots, p_n) \leftrightarrow (\lceil \texttt{e1} \rceil \leq i \wedge i \leq \lceil \texttt{e2} \rceil))$$

- $E =$ Bag{e1..e2}
  Wir gehen genauso wie im Falle eines Set-Literals vor, benutzen jedoch als neues Symbol in Σ^* bag_E und generieren zusätzlich in Ax_E folgendes Axiom:

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, \forall i{:}T \, (count(bag_E(p_1, \ldots, p_n), i) \leq 1)$$

- $E =$ Sequence{e1..e2}
  Wir gehen genauso wie im Falle eines Bag-Literals vor, benutzen jedoch als neues Symbol in Σ^* seq_E und generieren zusätzlich in Ax_E folgendes Axiom:

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, \forall i, j{:}T \, (i \leq j \rightarrow at(seq_E(p_1, \ldots, p_n), i) \leq at(seq_E(p_1, \ldots, p_n), j))$$

Beispiel 11
Gegeben sei der Integer Ausdruck n, dessen Übersetzung $\lceil$n$\rceil$ die freie Variable $s : S$ enthalte, sowie das Literal $E =$ Sequence{4..n}. Als Übersetzung erhalten wir schließlich

$$\lceil \texttt{Sequence\{4..n\}} \rceil = seq_E(s)$$

sowie folgende Axiome aus Ax_E, die die Semantik des neuen Symbols festhalten:

$$\dot{\forall} s{:}S \, \forall i{:}Integer \, (i \in seq_E(s) \leftrightarrow (4 \leq i \wedge i \leq \lceil \texttt{n} \rceil))$$
$$\dot{\forall} s{:}S \, \forall i{:}Integer \, (count(seq_E(s), i) \leq 1)$$
$$\dot{\forall} s{:}S \, \forall i, j{:}Integer \, (i \leq j \rightarrow at(seq_E(s), i) \leq at(seq_E(s), j))$$

Bemerkung (Kleinster gemeinsamer Obertyp von Basistypen).
Seien $\mathtt{e}_1, \ldots, \mathtt{e}_n$ beliebige OCL-Ausdrücke eines Basistyps, und T_i $(i = 1, \ldots, n)$ die zugehörigen OCL-Typen.

Dann gilt, daß es immer einen gemeinsamen Obertyp gibt (Any), es im allgemeinen aber *mehrere* kleinste gemeinsame Obertypen gibt. Der Grund liegt in möglichen Mehrfachvererbungen und Mehrfachimplementierungen in der Klassenstruktur in $\mathcal{D}$.

Wir haben nun aber von *dem* kleinsten gemeinsamen Obertypen gebrauch gemacht, aber welchen meinen wir denn nun?

Für unsere Zwecke spielt es grundsätzlich *keine* Rolle, welchen Typen wir aus der Menge der kleinsten gemeinsamen Obertypen verwenden (wir benötigen diesen Typen lediglich zur Typisierung des neu eingeführten Symbols!), die Wahl muß lediglich deterministisch sein und in allen Fällen auf die selbe Weise getroffen werden. Wir nehmen deshalb OBdA. an, daß es eine beliebige, aber *feste* totale Ordnung $<_{\mathcal{D}}$ auf den Typen in OCL gibt, und im Falle mehrerer kleinster gemeinsamer Obertypen das bzgl. $<_{\mathcal{D}}$ minimale Element aus dieser Kandidatenmenge gewählt wird. Man könnte sich außerdem vorstellen, daß die Ordnung $<_{\mathcal{D}}$, gewissen zusätzlichen Anforderungen genügt, beispielweise alle Klassen C bzgl. $<_{\mathcal{D}}$ kleiner sind, als Interfaces I. □

2.3.3.2 Basistypen – Boolean

Wie oben dargelegt wurde, bilden wir für einen booleschen OCL-Ausdruck `b` in den meisten Fällen auf eine logische Formel $\Phi_b \in For^{DL}$ ab. Die booleschen Operatoren `and`, `or`, `implies` und `not` werden auf die entsprechenden logischen Junktoren in der **DL** übersetzt, also $\wedge, \vee, \rightarrow, \neg$. Die Gleichheit auf Boolean wird dann insbesondere durch $\leftrightarrow$ verkörpert.

In zwei speziellen Situationen jedoch, ist diese Strategie, OCL-Ausdrücke des Typs Boolean auf logische Formeln abzubilden, nicht anwendbar: OCL läßt jeden Basistyp als Elementtyp für Kollektionen, die wir in der **DL** als Terme über ADTs darstellen, zu. Für Kollektionen mit dem Elementtyp Boolean ist das jedoch *nicht* mehr möglich, da die Signatur der ADTs mit **DL**-Sorten arbeitet, also insbesondere keine *Formeln* als Parameter in der Signatur vorkommen dürfen.

Außerdem wollen wir boolesche Attribute und Methoden aus Klassen in $\mathcal{D}$ nicht durch Formeln darstellen.

In diesem Fall müssen wir auf die **DL**-Sorte boolean ausweichen. Dort sind geeignete Symbole für die einzelnen logischen Verknüpfungen aus OCL vorhanden: `&&` für die Konjunktion, `||` für die Disjunktion, `!` für die Negation, `==` für die Äquivalenz; die Implikation wird durch kein eigenes Symbol repräsentiert und somit durch die Definition mit Negation und Disjunktion dargestellt[25].

Eine Wandlung zwischen den beiden alternativen Darstellungsformen ist immer möglich:

Die Übersetzung $\lceil \mathtt{b} \rceil$ des boolschen OCL-Ausdrucks `b` sei eine Formel $\Phi_b \in For^{DL}$. Wollen wir nun stattdessen einen Term t_b der Sorte `boolean` benutzten, so generieren wir unter Anwendung unserer Benennungstechnik ein *neues* Funktionssymbol s_b in Σ^*

[25] Im allgemeinen Fall einer Logiksprache L würden wir stattdessen einen ADT BOOLEAN mit Funktionssymbolen $and, or, implies, not, \doteq$ benutzen.

(mit Ergebnissorte `boolean`) und definieren $t_b = s_b(p_1, \ldots, p_n)$, wobei $p_1, \ldots, p_n$ genau die freien Variablen in Φ_b sind. Um die Semantik dieses neuen Symbols festzulegen, führen wir als Axiom die folgende Formel ein:

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, ((s_b(p_1, \ldots, p_n) \doteq \texttt{true}) \leftrightarrow \Phi_b(p_1, \ldots, p_n))$$

Sei nun andererseits die Übersetzung $\lceil \texttt{b} \rceil$ des booleschen OCL-Ausdrucks `b` ein Term t_b und wir benötigen stattdessen eine Darstellung in Form einer Formel Φ_b. Dann lautet diese Formel $\Phi_b = (t_b \doteq \texttt{true})$.

Der Operator `xor` wird als die Negation der Gleichheit, also durch `!=` behandelt.

Beispiel 12
Seien `p`, `q` boolesche OCL-Ausdrücke, deren Übersetzung $\lceil \texttt{p} \rceil$ bzw. $\lceil \texttt{q} \rceil$ eine Formel ist. Sei außerdem `r` ein boolescher Ausdruck, der durch einen Term der Sorte `boolean` dargestellt wird. Dann wird der Ausdruck E = `p and (q or r) implies false` abgebildet auf

$$\lceil \texttt{E} \rceil = (\lceil \texttt{p} \rceil \wedge \neg(\lceil \texttt{q} \rceil \leftrightarrow (\lceil \texttt{r} \rceil \doteq \texttt{true})) \rightarrow \mathit{false})$$

Die Menge der Axiome Ax_E der Übersetzung von E entspricht außerdem $Ax_E = Ax_p \cup Ax_r \cup Ax_s$, da keine zusätzlichen Axiome für die Übersetzung von E notwendig waren.

Der OCL-Typ Boolean stellt außerdem noch ein recht interessantes Konstrukt zur Verfügung, das möglicherweise einer kurzen Erläuterung bedarf: Seien `b` ein boolescher OCL-Ausdruck, `t` ein OCL-Ausdruck Typs T und `s` ein OCL-Ausdruck des Typs S. Sei O der kleinste gemeinsame Obertyp[26] von T und S im Typsystem von OCL. Die Auswertung des OCL-Ausdruck `E = if (b) then t else s endif` (bzgl. eines Snapshots D des Modells $\mathcal{D}$) besitzt dann die **DL**-Sorte O (welche O in der Logik repräsentiert) und entspricht genau der Auswertung des Ausdrucks `t`, wenn `b` zu wahr ausgewertet wird, und der Auswertung des Ausdrucks `s` im anderen Fall.

Wir haben hier ein schönes Beispiel für eine Situation, in der wir für die Übersetzung unsere Benennungstechnik anwenden müssen: Die Übersetzung von `E` läßt sich *nicht explizit* als Aggregation der Übersetzungen der Teilausdrücke `p`, `t` und `s` angeben!

Seien $p_1, \ldots, p_n$ genau die freien Variablen aus den Übersetzungen $\lceil \texttt{b} \rceil$, $\lceil \texttt{t} \rceil$, $\lceil \texttt{s} \rceil$ der Teilausdrücke und $T_1, \ldots, T_n$ die zugehörigen Sorten.

Wir führen nun ein neues Symbol $\mathit{if}_E{:}T_1 \times \ldots \times T_n \rightarrow O$ in Σ^* ein.

Dann definieren wir $\lceil \texttt{E} \rceil = \mathit{if}_E(p_1, \ldots, p_n)$ und fügen zur Festlegung der Bedeutung des neuen Symbols folgende Axiome in Ax_E ein:

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, (\lceil \texttt{b} \rceil \rightarrow \mathit{if}_E(p_1, \ldots, p_n) \doteq \lceil \texttt{t} \rceil)$$
$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, (\neg\lceil \texttt{b} \rceil \rightarrow \mathit{if}_E(p_1, \ldots, p_n) \doteq \lceil \texttt{s} \rceil)$$

Falls einer der Ausdrücke `t`, `s` boolesch ist und die zugehörige Übersetzung $\lceil \texttt{t} \rceil$ (bzw. $\lceil \texttt{s} \rceil$) einer Formel Φ entspricht, so muß (wegen der Typisierung von if_E) möglicherweise ein Repräsentationswechsel für diese Übersetzung zu einer Termdarstellung (gemäß dem oben geschilderten Vorgehen) vorgenommen werden.

[26] Man beachte die Bemerkung auf Seite 63.

Für O = Boolean, d.h. einen `if`-Ausdruck mit booleschen Zweigen, können wir prinzipiell genauso vorgehen. Wir können die Behandlung in diesem speziellen Fall jedoch auch etwas vereinfachen, da die Einführung eines neuen Symbols nicht erforderlich ist: Seien OBdA. $\lceil$`t`$\rceil$ und $\lceil$`s`$\rceil$ durch Formeln dargestellt. Dann definieren wir

$$\lceil \texttt{E} \rceil = (\lceil \texttt{b} \rceil \rightarrow \lceil \texttt{t} \rceil) \wedge (\neg \lceil \texttt{b} \rceil \rightarrow \lceil \texttt{s} \rceil)$$

Beispiel 13
Seien `b` ein boolescher Ausdruck, dessen Übersetzung $\lceil$`b`$\rceil$ die freie Variable $self : Person$ enthalte, `t` ein Ausdruck des Modelltyps Person, dessen Übersetzung ebenfalls die freie Variable $self : Person$ enthalte, und `s` ein Ausdruck des Modelltyps Customer, dessen Übersetzung zusätzlich die freie Variable $c : Customer$ enthalte. Dann erhalten wir als Übersetzung für den Ausdruck $E =$ `if (b) then t else s endif` den Term der Sorte $Person$

$$\lceil \texttt{E} \rceil = if_E(self, c)$$

und als zusätzliche Axiome in Ax_E:

$$\dot{\forall} self{:}Person\, \dot{\forall} c{:}Customer\, (\lceil \texttt{b} \rceil \rightarrow if_E(self, c) \doteq \lceil \texttt{t} \rceil)$$
$$\dot{\forall} self{:}Person\, \dot{\forall} c{:}Customer\, (\neg \lceil \texttt{b} \rceil \rightarrow if_E(self, c) \doteq \lceil \texttt{s} \rceil)$$

Bemerkung (OCL und `if`-Operator). Wir sind hier in der Darstellung des `if`-Operators sogar etwas allgemeiner, als die OCL-Spezifikation fordert: Die Typsierung des `if`-Ausdrucks E entspricht gemäß der OCL-Spezifikation [Obj01, Seite 6-83] genau dem Typ des Ausdrucks `t` im `then`-Zweig. Diese Definition erscheint recht merkwürdig[27] und ist nur dann *korrekt*, falls OCL implizit fordert, daß die Typen T und S übereinstimmen müssen. Wir erachten das als keine sinnvolle Einschränkung, da man sich durchaus sehr nützliche `if`-Ausdrück vorstellen kann, die nicht dieser Anforderung gerecht werden. Unsere Behandlung funktioniert auch für diese Ausdrücke. □

2.3.3.3 Basistypen – Integer, Real, String

Für alle Eigenschaften bzw. Operatoren der OCL-Typen Integer, Real, String enthält die Signatur der entsprechenden ADTs INTEGER, REAL, STRING ein (im allgemeinen gleichnamiges) Funktionssymbol.

Wir ziehen dann das entsprechende Funktions- oder Relationssymbol *feature* für die Übersetzung eines Operator `feature` dieser OCL-Typen heran. Insbesondere verwenden wir ein geeignetes Gleichheitsymbol $\doteq$ für die Darstellung des `=`-Operators.

Beispiel 14
Seien `i`, `j` OCL-Ausdrücke des Typs Integer, `p`, `q` OCL-Ausdrücke des Typs Real und `s`, `t` OCL-Ausdrücke des Typs String.

Dann gilt:

[27] Eigentlich scheint das auch eher ein Fehler in der OCL-Spezifikation zu sein.

$$\lceil \texttt{i + j} \rceil = \lceil \texttt{i} \rceil + \lceil \texttt{j} \rceil$$
$$\lceil \texttt{i.mod(j)} \rceil = mod(\lceil \texttt{i} \rceil, \lceil \texttt{j} \rceil)$$
$$\lceil \texttt{p.max(q)} \rceil = max(\lceil \texttt{p} \rceil, \lceil \texttt{q} \rceil)$$
$$\lceil \texttt{p.floor()} \rceil = floor(\lceil \texttt{p} \rceil)$$
$$\lceil \texttt{s = t} \rceil = \lceil \texttt{s} \rceil \doteq \lceil \texttt{t} \rceil$$
$$\lceil \texttt{s.concat(t)} \rceil = append(\lceil \texttt{s} \rceil, \lceil \texttt{t} \rceil)$$
$$\lceil \texttt{s.substring(i,j)} \rceil = substring(\lceil \texttt{s} \rceil, \lceil \texttt{i} \rceil, \lceil \texttt{j} \rceil)$$

2.3.3.4 Basistypen – OclAny

Der OCL-Typ OclAny ist der gemeinsame Obertyp aller Basistypen – speziell der Modelltypen aus dem Modell $\mathcal{D}$ – im Typsystem von OCL. Er ist im Prinzip als abstrakter Typ zu verstehen, d.h jede Instanz dieses Typs ist gleichermaßen eine Instanz eines spezielleren Typs T, und definiert Eigenschaften bzw. Operatoren, die auf alle Instanzen eines OCL-Typs T anwendbar sind.

Seien im folgenden e1 (bzw.e2) ein OCL-Ausdruck des Typs T1 (bzw. T2), T1 und T2 beide Basistypen in OCL und T ihr kleinster gemeinsamer Obertyp[28]. Sei außerdem S das Symbol in OclType, das den OCL-Typ S repräsentiert.

- Gleichheit — Der OCL-Ausdruck `e1 = e2` wird unter Verwendung des Gleichheitssymbols $\doteq$ (auf der Sorte T) übersetzt:

  $$\lceil \texttt{e1 = e2} \rceil = \lceil \texttt{e1} \rceil \doteq \lceil \texttt{e2} \rceil$$

  Entsprechende wird die Ungleichheit in OCL (`e1 <> e2`) als Negation der Gleichheit gehandhabt.

- `oclIsKindOf` — Mittels `e1.oclIsKindOf(S)` ist es möglich zu testen, ob eine bestimmte Instanz `e1` einem Typen S angehört bzw. einem seiner Untertypen S1, ..., Sn.

  Für die Übersetzung benutzten wir nun die folgende Formel:

  $$\lceil \texttt{e1.oclIsKindOf(S)} \rceil = \dot{\exists} s{:}S\ (s \doteq \lceil \texttt{e1} \rceil)$$

- `oclIsTypeOf` — Der Operator `oclIsTypeOf` ist recht ähnlich zu `oclIsKindOf`, mit dem wesentlichen Unterschied, daß der Ausdruck `e1.oclIsKindOf(S)` es einem Modellierer ermöglicht zu testen, ob eine bestimmte Instanz `e1` einem Typen S angehört, aber keinem seiner Untertypen S1, ..., Sn, d.h. daß die Instanz `e1` S als ihren speziellsten Typen besitzt.

  Für die Übersetzung benutzten wir in diesem Fall daher

  $$\begin{aligned} \lceil \texttt{e1.oclIsTypeOf(S)} \rceil = {} & \dot{\exists} s{:}S\ (s \doteq \lceil \texttt{e1} \rceil) \\ & \wedge \dot{\forall} s_1{:}S_1\ \neg(s_1 \doteq \lceil \texttt{e1} \rceil) \\ & \wedge \ldots \\ & \wedge \dot{\forall} s_n{:}S_n\ \neg(s_n \doteq \lceil \texttt{e1} \rceil) \end{aligned}$$

[28] Man beachte die Bemerkung auf Seite 63.

Beispiel 15
Sei `prs` ein OCL-Ausdruck des Typs Person. Benutzen wir nun den Ausdruck `prs.oclIsTypeOf(Person)` um zu testen, ob `prs` zwar eine Person, aber kein Kunde ist, dann liefert uns die Abbildung als Ergebnis

$$\dot{\exists} p{:}Person\,(p \doteq \lceil \texttt{prs} \rceil) \wedge \dot{\forall} c{:}Customer\,\neg(c \doteq \lceil \texttt{prs} \rceil)$$

- `oclAsType` — Der Operator `oclAsType` dient zur Typwandlung in OCL, das bedeutet `e.oclAsType(S)` beschreibt eine Instanz, die mit `e` bis auf die Tatsache identisch ist, daß sie dem OCL-Typen S (anstatt T) angehört. Prinzipiell sollte die Typwandlung sowohl in einen *allgemeineren*, als auch in einen *spezielleren* Typen möglich sein. Man muß außerdem beachten, daß der OCL-Ausdruck undefiniert ist, sofern der tatsächliche Typ des durch `e` beschriebenen Objekts (d.h. der speziellste Typ dem dieses Objekt angehört) nicht S oder einem Untertypen von S entspricht.

  Für die Abbildung des Ausdrucks in die **DL** enthält Σ^* je ein Funktionsymbol $oclAsType_{T,S}{:}T \to S$ für jedes solche Paar T, S von OCL-Typen.

  Wir generieren wir als Übersetzung von `e1.oclAsType(S)`

  $$\lceil \texttt{e.oclAsType(S)} \rceil = oclAsType_{T,S}(\lceil \texttt{e} \rceil)$$

  Die axiomatische Definition läßt sich für den *wichtigeren* Fall der Typwandlung in einen Untertypen sehr einfach angeben:

  $$\dot{\forall} s{:}S\,(oclAsType_{T,S}(s) \doteq s)$$

  Im anderen Falle, daß es sich bei S um einen Obertypen von T handelt[29], wird hingegen durch

  $$\dot{\forall} t{:}T\,(\dot{\exists} s{:}S\,(t \doteq s) \to oclAsType_{T,S}(t) \doteq t)$$

  axiomatisiert.

- `oclIsNew` — Der Typ OclAny enthält schließlich noch ein Feature `oclIsNew`, welches nur in Nachbedingungen verwendet werden darf. Die Behandlung dieses Operators erfordert etwas „Zusatzaufwand" an Erläuterungen, der später sowieso geleistet werden muß. Wir verschieben die Besprechung der Behandlung deshalb auf den darstellungstechnisch günstigeren Abschnitt 2.3.3.12.

2.3.3.5 Basistypen – Modelltypen

In OCL fungieren folgenden Größen als Eigenschaften eines Modelltyps bzw. einer Klasse C aus dem UML-Modell $\mathcal{D}$:

[29] Der andere Fall ist nur von Bedeutung, wenn man in einer Unterklasse auf eine in dieser Unterklasse überschriebene *Eigenschaft* in der Oberklasse zugreifen möchte, um Mehrdeutigkeiten bei der Benennung in OCL Ausdrücken aufzulösen. Siehe [Obj01, Seite 6-64]. Dieser Fall scheint jedoch eher pathologischer Natur zu sein, als von hoher praktischer Bedeutung.

- Die (nichtstatischen[30]) Attribute der Klasse C, die in $\mathcal{D}$ deklariert wurden.
- Die (nichtstatischen) Methoden der Klasse C, die in $\mathcal{D}$ deklariert wurden.
- Eigenschaften zur Navigation über Assoziationen in $\mathcal{D}$, an denen die Klasse C teilnimmt.

In Abschnitt 2.3.2 haben wir die Formalisierung des UML-Modells $\mathcal{D}$ in der Logik **DL** diskutiert. Dabei entstand insbesondere eine Basissignatur $\Sigma_{\mathcal{D}}$ in der **DL**. Wir können nun für die Abbildung der Eigenschaften (aus den ersten drei Punkten der obigen Liste) eines Modelltypen die entsprechenden Symbole aus $\Sigma_{\mathcal{D}}$ verwenden:

Sei `e` ein OCL-Ausdruck mit dem (Modell)Typ C.

Ist nun `a:T` ein Attribut der Klasse C, so haben wir ein entsprechendes Attribut $a : T$ in der Kontextklassen C erzeugt. Damit bilden wir den Ausdruck $E =$ `e.a` auf folgenden Term der Sorte T

$$\lceil \texttt{e.a} \rceil = \lceil \texttt{e} \rceil.a$$

Genauso verfahren wir mit Methoden `m(p1:T1,...,pn:Tn):T`: Der Methodenaufruf `e.m(v1,...,vn)` wird durch

$$\lceil \texttt{e.m(v1,...,vn)} \rceil = \lceil \texttt{e} \rceil.m(\lceil \texttt{v1} \rceil, \ldots, \lceil \texttt{vn} \rceil)$$

behandelt.

Für Eigenschaften, die zur Navigation über Assoziationen r (oder von und zu Assoziationsklassen A) dienen, verwenden wir genauso die entsprechenden Symbole aus $\Sigma_{\mathcal{D}}$. Es sei jedoch nochmals angemerkt, daß wir anstatt der gewöhnlichen Funktionsymbol-Darstellung die Darstellung mittels Punkt (wie für Attribute und Methoden) aus Gründen der Ähnlichkeit zur Syntax in OCL bevorzugen und deshalb anwenden.

Nichtqualifizierte Navigationsausdrücke `e.r` werden also wie Attributeauswertungen übersetzt:

$$\lceil \texttt{e.r} \rceil = \lceil \texttt{e} \rceil.r$$

,wohingegen qualifizierte Navigationsausdrücke `e.r[q]` prinzipiell wie Methodenaufrufe übersetzt werden – wobei `q` ein OCL-Ausdruck des Typs Q sei:

$$\lceil \texttt{e.r[q]} \rceil = \lceil \texttt{e} \rceil.r(\lceil \texttt{q} \rceil)$$

Sollten mehrere qualifizierende Werte `q1`, ..., `qN` verwendet werden, so wird die Angabe jedes einzelnen dieser Werte von UML/OCL gefordert (d.h. partielle Angaben von qualifizierenden Werten sind nicht erlaubt) und wir erhalten

$$\lceil \texttt{e.r[q1, ..., qN]} \rceil = \lceil \texttt{e} \rceil.r(\lceil \texttt{q1} \rceil, \ldots, \lceil \texttt{qN} \rceil)$$

Beispiel 16
Betrachten wir wieder OCL-Ausdrücke über unserem Beispielmodell, die bezüglich der Klasse *Customer* als Kontext formuliert wurden: Das Alter einer Person `self` wird durch `self.age` in OCL beschrieben und durch den Term $\lceil$`self.age`$\rceil$ = *self.age*

[30] Die statischen Attribute und Methoden werden in OCL als Eigenschaften des Symbols C aus OclType angesehen, welches den Modelltyp C verkörpert. Wir werden sie deshalb in Abschnitt 2.3.3.6 gesondert behandeln.

übersetzt.

Der Einkommen einer Person `self` zu einem bestimmten Datum `d` (des Typs Date) wird durch `self.income(d)` in OCL dargestellt. Als Übersetzung ergibt sich schließlich $\lceil$`self.income(d)`$\rceil = self.income(\lceil$`d`$\rceil)$.

Die Menge der Arbeitsverträge, die die Person `self` eingegangen ist, entspricht dem OCL-Ausdruck `self.labourContract`. Sie wird durch den Term

$$\lceil \texttt{self.labourContract} \rceil = self.labourContract$$

in der Logik beschrieben.

Wollen wir in OCL schließlich die Person charakterisieren, welche an einem Arbeitsvertrag `ct` (des Typs Contract) teilnimmt, so benutzten wir `ct.employees`. Nach der Formalisierung des Beispielmodells erzeugen wir damit als Übersetzung den Term $\lceil$`ct.employees`$\rceil = ct.employees$

2.3.3.6 Basistypen – OclType

In OCL bietet der Typ OclType in gewissem Umfang Zugriff auf die UML-Metaebene: Es gibt Eigenschaften für den Zugriff auf das UML-Modell $\mathcal{D}$ selbst. Diese Eigenschaften betrachten wir – wie schon in Abschnitt 2.3 erwähnt – hier nicht weiter.

Wir beschränken uns auf die Behandlung von folgenden zwei Eigenschaften, die auf Instanzen `T` des Typs OclType anwendbar sind:

- `allInstances` — Der Operator `allInstances` erlaubt die Darstellung der Menge der Instanzen eines Typs T bzw. genauer: Die Auswertung des Ausdrucks `T.allInstances` bzgl. eines Snapshots D des UML-Modells $\mathcal{D}$ liefert gerade die Menge der Instanzen des Typs T in diesem Snapshot D.

  Für die Übersetzung von `T.allInstances` enthält Σ^* das (nichtrigide) Konstantensymbol $allInstances_T$ der Sorte Set_T. Dieses Symbol wird durch das Axiom

  $$\dot{\forall} t{:}T\ (t \in allInstances_T)$$

  in seiner Interpretation festgelegt.

  Als Ergebnis der Abbildung definieren wir dann einfach als folgenden Term:

  $$\lceil \texttt{T.allInstances} \rceil = allInstances_T$$

  Bemerkung (Nichtrigide Funktionssymbole). Man beachte, daß die Eigenschaft von $allInstances_T$, eine *nichtrigides* Konstante zu sein, wesentlich für die Korrektheit ist, da im Falle von Methodenspezifikationen die Menge der existierenden Instanzen im Vor- und Nachzustand im allgemeinen *verschieden* ist. Wäre $allInstances_T$ (herkömmliches) *rigides*[31] Symbol, so müssten für Methodenspezifikationen *zwei verschiedene* Symbole für die Vor- und Nachbedingungen verwendet werden, um ein korrektes Verfahren zu erhalten. □

[31] D.h. es würde in *jedem* Zustand in der gleichen Weise interpretiert werden!

Bemerkung (OCL und endliche Mengen). OCL selbst betrachtet stets nur endliche Kollektionen. Die OCL-Spezifikation bemerkt deshalb im Zusammenhang mit dem `allInstances`-Operator, daß seine Anwendung auf die Typobjekte `String`, `Integer` und `Real` zu *undefinierten* Ausdrücken führt. Insofern betrachten wir hier die Anwendung des `allInstances`-Operators zunächst nur für Modelltypen `C`. Eine Strategie zur Auflösung von undefinierten Ausdrücken, muß in in den anderen Fällen sicherstellen, daß der oben generierte Term „keinen Schaden" anrichtet, da dann die Interpretion des Symbols $allInstances_T$ nicht der Semantik von OCL genügt. Wir werden diesen Sachverhalt am Ende dieser Arbeit in Abschnitt 4.3.1 nochmals kurz aufgreifen und diskutieren. □

- Statische Attribute und Methoden eines Modelltyps `C` — Statische Attribute die im UML-Modell $\mathcal{D}$ in einer Klasse `C` definiert wurden, werden in OCL als Eigenschaft des OclType-Literals `C`, welches als Symbol in OCL die Klasse repräsentiert, angesehen. Bei der Formalisierung des Modells $\mathcal{D}$ entstanden in $\Sigma_\mathcal{D}$ entsprechende Symbole (bzw. statische Attribute und Methoden in der Kontextklasse C), die wir nun verwenden:

  Seien `a:T` ein statisches Attribut und `m(p1:T1,...,pn:Tn):T` eine statische Methode der Klasse `C`.

  Dann bilden wir den OCL-Ausdruck `C.a` (bzw. `C.m(v1,...,vn)`) auf den Term

  $$\lceil \texttt{C.a} \rceil = C.a$$

  bzw.

  $$\lceil \texttt{C.m(v1,...,vn)} \rceil = C.m(\lceil \texttt{v1} \rceil, \ldots, \lceil \texttt{vn} \rceil)$$

  ab.

Beispiel 17
Betrachten wir die statischen Methoden `duration` und `today` in der Klasse *Date* und sei `e` ein OCL-Ausdruck des Typs Date. Dann bezeichnet der OCL-Ausdruck `Date.duration(Date.today(),e)` die Länge der Zeitspanne zwischen dem heutigen Datum und dem gegebenen Datum `e` und wird übersetzt durch:

$$\lceil \texttt{Date.duration(Date.today(),e)} \rceil = Date.duration(Date.today(), \lceil \texttt{e} \rceil)$$

2.3.3.7 Kollektionstypen – Collection

Das Typsystem in OCL betrachtet Collection(T) als abstrakten Kollektionstypen und verwendet ihn hauptsächlich um Operationen zu definieren, die allen Spielarten konkreter Kollektionen Set(T), Bag(T) und Sequence(T) in OCL gemein sind.

Insbesondere besitzt dieser allgemeinste Kollektionstyp Collection(T) an sich keine praktische Bedeutung, da er in der Anwendung selbst nicht verwendet wird. Wir übersetzen daher alle Eigenschaften, die durch diesen Typen definiert werden, für alle konkreten Ausprägungen von Kollektionen einzeln. Wir wollen im folgenden die einzelnen Eigenschaften durchgehen, die alle Kollektionstypen gleichermaßen aufweisen, und die den einzelnen Konkretisierungen speziellen Features anschließend in den Abschnitten 2.3.3.8, 2.3.3.9 und 2.3.3.10 besprechen.

Sei im folgenden c ein OCL-Ausdruck des Typs Set(T), Bag(T) oder Sequence(T) und T als Elementtyp der Kollektion ein entsprechender Basistyp. Seien desweiteren c1 bzw. c2 OCL-Ausdrücke eines Kollektionstypen mit den Elementtypen T1 bzw. T2,T der kleinste gemeinsame Obertyp[32] von T1 und T2, o bzw. exp Ausdrücke des Typs OclAny und b ein Ausdruck des Typs Boolean.

- size, count, includes, excludes, sum. Für alle diese Eigenschaften gibt es ein korrespondierendes Funktions- oder Relationsymbol in der Σ^*, das wir für die Übersetzung heranziehen. Diese werden durch entsprechende Formeln in der Spezifikation der zugehörigen ADTs axiomatisiert.

  Wir übersetzten deshalb wie folgt:

$$\lceil \texttt{c->size} \rceil = size(\lceil \texttt{c} \rceil)$$
$$\lceil \texttt{c->includes(o)} \rceil = \lceil \texttt{o} \rceil \in \lceil \texttt{c} \rceil$$
$$\lceil \texttt{c->excludes(o)} \rceil = \neg\,(\lceil \texttt{o} \rceil \in \lceil \texttt{c} \rceil)$$
$$\lceil \texttt{c->sum} \rceil = sum(\lceil \texttt{c} \rceil)$$

  Beispiel 18
  Betrachten wir die Übersetzung des OCL-Ausdrucks
  `self.numberOfEmployees = self.employees->size` mit der Klasse *Company* als Kontext, dann erhalten wir nach der Abbildung:

$$\lceil \texttt{self.numberOfEmployees = self.employees->size} \rceil = \\ self.numberOfEmloyees \doteq size(self.employees)$$

  Für den Ausdruck `self.employees->includes(self.boss)` (ebenfalls mit der Klasse *Company* als Kontext) ergibt sich andererseits:

$$\lceil \texttt{self.employees->includes(self.boss)} \rceil = \\ self.boss \in self.employees$$

- includesAll, excludesAll. Der Ausdruck `c1->includesAll(c2)` drückt aus, daß die Kollektion c2 eine Teilkollektion von c1 ist, also jedes Element e aus c2 auch in c1 enthalten ist.

  Sei nun $e{:}T$ eine neue Variable, die nicht frei in $\lceil \texttt{c1} \rceil$ oder $\lceil \texttt{c2} \rceil$ vorkommt.

  Die Abbildung erzeugt somit:

$$\dot{\forall} e{:}T\,(e \in \lceil \texttt{c2} \rceil \rightarrow e \in \lceil \texttt{c1} \rceil)$$

  Die Verwendung von excludes besagt stattdessen, daß kein Element e aus c2 auch in c1 enthalten ist und wird auf folgende Art behandelt:

$$\dot{\forall} e{:}T\,(e \in \lceil \texttt{c2} \rceil \rightarrow \neg(e \in \lceil \texttt{c1} \rceil))$$

  Beispiel 19
  Der Ausdruck `self.customers->includesAll(self.specials)` mit der Klasse *Company* als Kontext, bedeutet, daß die Menge der besonderen Kunden (beispielsweise VIP-Kunden mit besonderen Vergünstigungen oder Dienstleistungen)

[32] Man beachte die Bemerkung auf Seite 63.

eines Unternehmens `self` in der Menge aller Kunden zu diesem Unternehmen enthalten ist. Die Übersetzung liefert:

$$\lceil \texttt{self.customers->includesAll(self.specials)} \rceil = \dot{\forall} c{:}Customer\,(c \in self.specials \rightarrow c \in self.customers)$$

Würde man stattdessen fordern, daß alle Personen, die für das Unternehmen arbeiten, auch Kunden des Unternehmens sind,
d.h. `self.customers->includesAll(self.employees)`, dann ergibt sich jedoch:

$$\lceil \texttt{self.customers->includesAll(self.employees)} \rceil = \dot{\forall} p{:}Person\,(p \in self.employees \rightarrow p \in self.customers)$$

- `isEmpty`, `notEmpty`. Der Ausdruck `c->isEmpty` testet, ob eine Kollektion `c` leer ist.

  Sei nun $e{:}T$ eine neue Variable, die nicht frei in $\lceil \texttt{c} \rceil$ vorkommt.

  Wir erzeugen dann bei der Abbildung:

  $$\dot{\forall} e{:}T\,\neg(e \in \lceil \texttt{c} \rceil)$$

  Analog behandeln wir `c->notEmpty` durch:

  $$\dot{\exists} e{:}T\,(e \in \lceil \texttt{c} \rceil)$$

- `forAll`, `exists`. OCL ermöglicht es zu prüfen, ob alle Elemente e einer Kollektion `c` eine gewisse Eigenschaft `b` haben. Dazu formuliert man den OCL-Ausdruck $E =$ `c->forAll(e|b)`. Etwas genauer gesagt, ergibt die Auswertung von E den Wert *true*, falls die Auswertung von `b` bzgl. jedes Elements e der Kollektion den Wert *true* annimmt.

  Sei nun $e'{:}T$ eine neue Variable, die nicht frei in $\lceil \texttt{c} \rceil$ und $\lceil \texttt{b} \rceil$ vorkommt.

  Wir formalisieren E deshalb durch:

  $$\lceil \texttt{E} \rceil = \dot{\forall} e'{:}T\,(e' \in \lceil \texttt{c} \rceil \rightarrow \lceil \texttt{b} \rceil \{e/e'\})$$ [33]

  Dahingegen wird $E =$ `c->exists(e|b)` wahr, wenn es ein Element e in der Kollektion `c` gibt, für das sich die Auswertung von `b` zu *true* ergibt. Damit behandeln wir E folgendermaßen:

  $$\lceil \texttt{E} \rceil = \dot{\exists} e'{:}T\,(e' \in \lceil \texttt{c} \rceil \wedge \lceil \texttt{b} \rceil \{e/e'\})$$

Beispiel 20
Wir formulieren in OCL durch den Ausdruck $E =$
`cmp.customers->forAll(c|c.age >= 18)` die Anforderung, daß alle Kunden eines Unternehmens `cmp` erwachsen sind.

[33] Die Notation $t\{e/s\}$ steht für den Term, der sich als Ergebnis der syntaktischen Ersetzung aller Vorkommen der Variablen e im Term t durch den Term s ergibt. Diese syntaktische Ersetzung verläuft insbesondere – durch geeignete Umbenennungen von Quantorvariablen und ihren Vorkommen – kollisionsauflösend!

Die Übersetzung von E liefert:

$$\lceil \texttt{E} \rceil = \dot{\forall} c{:}Customer\,(c \in cmp.customers \rightarrow c.age \geq 18)$$

Möchten wir stattdessen formulieren, daß das der Chef des Unternehmens `cmp` in Zukunft mehr verdienen wird – bei welchen Arbeitgebern auch immer –, dann benutzen wir den Ausdruck $E =$

```
Date.allInstances->exists(d|
   Date.today().isBefore(d) and
   cmp.boss.income(d) > cmp.boss.income(Date.today()))
```

und erhalten als Ergebnis der Abbbildung:

$$\begin{aligned}\lceil \texttt{E} \rceil = \dot{\exists} d{:}Date\,(&d \in allInstances_{Date}\\ &\wedge\ Date.today().isBefore(d)\\ &\wedge\ \ cmp.boss.income(d) >\\ &\quad\ cmp.boss.income(Date.today()))\end{aligned}$$

und als Menge Ax_E von Axiomen:

$$\dot{\forall} d{:}Date\,(d \in allInstances_{Date})$$

- `isUnique`. Mit $E =$ `c->isUnique(e|exp)` kann ein Modellierer eine Art von *Schlüsseleigenschaft* des Ausdrucks `exp` für die Kollektion `c` testen: Der Ausdruck E gilt, wenn die Auswertung des Ausdrucks `exp` für jede Instanziierung von `e` mit einem Element der Kollektion einen anderen Wert ergibt.

  Seien nun $e_1, e_2{:}T$ neue Variable, die nicht frei in $\lceil \texttt{c} \rceil$ und $\lceil \texttt{exp} \rceil$ vorkommen.

  Wir beschreiben dann diesen Sachverhalt durch die Formel

$$\begin{aligned}\lceil \texttt{E} \rceil \ \ = \ \ \dot{\forall} e_1{:}T\,\dot{\forall} e_2{:}T\,((&e_1 \in \lceil \texttt{c} \rceil \wedge e_2 \in \lceil \texttt{c} \rceil \wedge\\ &\lceil \texttt{exp} \rceil\{e/e_1\} \doteq \lceil \texttt{exp} \rceil\{e/e_2\}) \ \rightarrow\ e_1 \doteq e_2)\end{aligned}$$

Beispiel 21
In der modellierten Miniwelt soll gelten, daß das Attribut *bankID* als eindeutiger Identifikator für die verschiedenen Banken benutzt werden kann. In OCL formulieren wir dazu den Ausdruck
$E =$ `Bank.allInstances->isUnique(b|b.bankID)` und behandeln diesen wie folgt:

$$\begin{aligned}\lceil \texttt{E} \rceil = \dot{\forall} b_1{:}Bank\,\dot{\forall} b_2{:}Bank\,((&b_1 \in allInstances_{Bank} \wedge\\ &b_2 \in allInstances_{Bank} \wedge\\ &b_1.bankID \doteq b_2.bankID) \ \rightarrow\ b_1 \doteq b_2)\end{aligned}$$

Die Menge der Axiome Ax_E entspricht durch die Übersetzung des Teilausdrucks `Bank.allInstances` außerdem:

$$\dot{\forall} b{:}Bank\,(b \in allInstances_{Bank})$$

- `sortedBy`. Wird der OCL-Ausdruck $E =$ `c->sortedBy(e|exp)` ausgewertet, so ergibt sich eine *Sequenz*, die zum einen genau die gleichen Elemente e (mit der gleichen Anzahl von Vorkommen) besitzt, und zum anderen bzgl. den Werten aus der Auswertung von `exp` (bei Instanziierung von `e` mit einem beliebigen Element aus der Kollektion) aufsteigend geordnet[34] ist.

  Um diese Sequenz in der Logik darzustellen, wenden wir abermals unsere Benennungstechnik an und erzeugen ein neues Symbol $sortedBy_E$ in Σ^*.

  Seien $p_1, \ldots, p_n$ genau die freien Variablen aus den Übersetzungen $\lceil$`c`$\rceil$, $\lceil$`exp`$\rceil$ der Teilausdrücke – mit Ausnahme der Variablen e, da diese in E gewissermaßen gebunden wird –, $T_1, \ldots, T_n$ die zugehörigen Sorten und $e'{:}T$ eine neue Variable, die nicht frei in $\lceil$`c`$\rceil$ und $\lceil$`exp`$\rceil$ vorkommt.

  Dann hat das neue Funktionssymbol die Signatur

  $$sortedBy_E{:}T_1 \times \ldots \times T_n \to Sequence_T$$

  und wir definieren

  $$\lceil \texttt{E} \rceil = sortedBy_E(p_1, \ldots, p_n)$$

  Zu Sicherstellung der beiden oben genannten Eigenschaften der Ergebnissequenz führen wir folgende Axiome in Ax_E ein:

  $$\begin{aligned}
  &\dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \dot\forall e'{:}T\ (count(\lceil \texttt{c} \rceil, e') \doteq \\
  &\qquad\qquad count(sortedBy_E(p_1, \ldots, p_n), e')) \\
  &\dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall i, j{:}Integer\ (\\
  &\quad (1 \le i \wedge i \le j \wedge j \le size(sortedBy_E(p_1, \ldots, p_n))) \to \\
  &\qquad \lceil \texttt{exp} \rceil \{e/at(sortedBy_E(p_1, \ldots, p_n), i)\} \\
  &\qquad \le \lceil \texttt{exp} \rceil \{e/at(sortedBy_E(p_1, \ldots, p_n), j)\})
  \end{aligned}$$

  Beispiel 22
  Der Ausdruck $E =$ `c.employees->sortedBy(e| - e.income(d))` berechnet eine Liste aller Angestellten eines Unternehmens `c`, welche nach dem (Gesamt-)Einkommen der Angestellten zum Datum `d` absteigend geordnet ist.

  Nach der Übersetzung des Ausdrucks E ergibt sich nun

  $$\lceil \texttt{E} \rceil = sortedBy_E(c, d)$$

  und die Menge der Axiome Ax_E entspricht genau

  $$\begin{aligned}
  &\dot\forall c{:}Company \dot\forall d{:}Date \dot\forall p{:}Person\ (\\
  &\quad count(c.employees, p) \doteq count(sortedBy_E(c, d), p)) \\
  &\dot\forall c{:}Company \dot\forall d{:}Date \forall i, j{:}Integer\ (\\
  &\quad (1 \le i \wedge i \le j \wedge j \le size(sortedBy_E(c, d))) \to \\
  &\qquad -(at(sortedBy_E(c, d), i).income(d)) \\
  &\qquad \le -(at(sortedBy_E(c, d), j).income(d)))
  \end{aligned}$$

[34] Das ist nur dann möglich, wenn auf dem Typen `T` des Ausdrucks `e` eine Ordnung $<_T$ definiert ist. Wir betrachten daher nur OCL-Ausdrücke `exp` des Typs `Integer` bzw. `Real`.

- `select`, `reject`. Der `select` bzw. `reject` Operator fungiert als eine Art Filter-Operator zur Bildung von Teilkollektionen. Die Auswertung des OCL-Ausdruck $E =$ `c->select(e|b)` ergibt genau die Teilkollektion von `c`, die alle Elemente `e` (in der gleichen Anzahl) aus `c` mit der Eigenschaft `b` enthält. Für die Übersetzung benutzen wir wieder unsere Benennungstechnik und führen ein neues Funktionssymbol $select_E$ in Σ^* ein.

  Seien $p_1, \ldots, p_n$ genau die freien Variablen aus der Übersetzung $\lceil$`b`$\rceil$ der Filterbedingung – mit Ausnahme der durch E gebundenen Variablen e – und $T_1, \ldots, T_n$ die zugehörigen Sorten. Seien $e'{:}T$ und $s{:}Set_T$ neue Variablen, die nicht frei in $\lceil$`b`$\rceil$ vorkommen.

  Entsprechend der Sorte des Terms $\lceil$`c`$\rceil$ besitzt das neue Symbol die Signatur

  $$\begin{array}{ll} select_E{:}Set_T \times T_1 \times \ldots \times T_n \to Set_T & \textit{oder} \\ select_E{:}Bag_T \times T_1 \times \ldots \times T_n \to Bag_T & \textit{oder} \\ select_E{:}Sequence_T \times T_1 \times \ldots \times T_n \to Sequence_T & \end{array}$$

  Dann verwenden wir

  $$\lceil \texttt{E} \rceil = select_E(\lceil \texttt{c} \rceil, p_1, \ldots, p_n)$$

  und fügen (für *Mengen* `c`) zur Axiomatisierung des neuen Symbols $select_E$ folgende Formeln in Ax_E ein:

  $$\begin{array}{l} \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n\, select_E(emptySet_T, p_1, \ldots, p_n) \doteq emptySet_T \\ \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall s{:}Set_T \dot\forall e'{:}T\, (\lceil \texttt{b} \rceil\{e/e'\} \to \\ \quad select_E(insert(s, e'), p_1, \ldots, p_n) \doteq \\ \quad insert(select_E(s, p_1, \ldots, p_n), e')) \\ \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall s{:}Set_T \dot\forall e'{:}T\, ((\neg\lceil \texttt{b} \rceil)\{e/e'\} \to \\ \quad select_E(insert(s, e'), p_1, \ldots, p_n) \doteq select_E(s, p_1, \ldots, p_n)) \end{array}$$

  Falls die Kollektion `c` keine Menge ist, sondern einer Multimenge bzw. eine Sequenz entspricht, dann ist in den obigen Formeln lediglich das Konstantensymbol für die leere Menge $emptySet_T$ durch das entsprechende Konstantensymbol $emptyBag_T$ bzw. $emptySequence_T$ zu ersetzen.

  Die Quantor-Variable $s{:}Set_T$ wird entsprechend dem Typ des Kollektionsausdrucks `c` zu $b{:}Bag_T$ bzw. $s{:}Sequence_T$ angepaßt, wobei auch diese Variablen nicht frei in $\lceil$`b`$\rceil$ vorkommen.

Da diese Formeln ihrem Charakter nach grundlegend anders sind, als die Axiome, die wir bisher für die Festlegung der Semantik anderer neu erzeugter Symbole generiert haben, wollen wir ihre Intention kurz erläutern:

Diese Formeln[35] entsprechen einer induktiven Definition des operativen Verhaltens der *speziellen* Filterfunktion $select_E$ – in der insbesondere die Filterbedingung `b` implizit steckt! – auf der Struktur eines beliebigen Terms t des entsprechenden ADTs Set_T, Bag_T oder $Sequence_T$:

[35] Eine ähnliches Vorgehen wird bei der Übersetzung des `select`-Operators in [HHK98] gewählt. Dort fehlen jedoch die freien Variablen $p_1, \ldots, p_n$ die als Parameter mit dem neuen Funktionsymbole verwendet werden, was zu einer *inkorrekten* Übersetzung führt.

Das erste Axiom verkörpert dabei den Basisfall (die leere Kollektion), wohingegen die beiden anderen Axiome dem Induktionsschritt (Erweiterung einer bestehenden Kollektion s um ein weiteres Element e) entsprechen und eine Fallunterscheidung hinsichtlich des Zutreffens der Filterbedingung `b` für das betrachtete Element e darstellen.

Bemerkung (Wohldefiniertheit der Definition). Wie immer bei induktiven Definitionen muß man sich auch hier Fragen, ob die Definition wohldefiniert ist. Insbesondere sind die Eigenschaften der Termdarstellungen der Kollektionen miteinzubeziehen:

Die ADTs Set_T, Bag_T und $Sequence_T$ werden generiert von der Konstante für die jeweilige leere Kollektion und der Einfüge-Operation $insert$. Für Sequenzen ist die zugehörige Termdarstellung eindeutig, wohingegen bei Mengen und Multimengen keine Eindeutigkeit gewährleistet ist: Das Ändern der Reihenfolge der Elemente in einer Termdarstellung führt bei Mengen und Multimengen zu einer Beschreibung der *gleichen* Kollektion. Bei Mengen spielt außerdem die Anzahl der Einfügungen eines Elements keine Rolle.

In allen Fällen macht man sich durch Induktion klar, daß das Ergebnis der Funktion $select_E$ unter der angegebenen Definition für eine beliebige Termdarstellung einer Kollektion eindeutig festgelegt ist.

Wir wollen aber außerdem erreichen, daß für *alle* Termdarstellungen *einer Kollektion* die obige Definition einen Ergebnisterm liefert, der jeweils die gleiche Ergebniskollektion beschreibt!

Im Falle von Sequenzen ist die obige Definition also wegen der Eindeutigkeit der Termdarstellung von Sequenzen wohldefiniert. Desweitern macht man sich schnell durch einfache Fallunterscheidung klar das die Reihenfolge der Elemente in einer Termdarstellung keine Rolle spielt, d.h. daß für alle Termdarstellungen einer Multimenge bzw. Menge c und alle Elemente $e_1, e_2{:}T, p_1{:}T_1, \ldots, p_n{:}T_n$ gilt:

$$select_E(insert(insert(c, e_1), e_2), p_1, \ldots, p_n) \text{ und}$$
$$select_E(insert(insert(c, e_2), e_1), p_1, \ldots, p_n)$$
beschreiben die *gleiche* Kollektion.

Somit ist die Definition auch für Multimengen wohldefiniert. Für die Wohldefiniertheit im Falle von Mengen benötigen wir noch die Unabhängigkeit von der Anzahl der Einfügungen eines Elements. Man macht sich auch hier durch einfache Fallunterscheidung klar, daß unter der angegebenen Definition für jede Termdarstellung einer Menge c und alle Elemente $e{:}T, p_1{:}T_1, \ldots, p_n{:}T_n$ gilt:

$$select_E(insert(c, e), p_1, \ldots, p_n) \text{ und}$$
$$select_E(insert(insert(c, e), e), p_1, \ldots, p_n)$$
beschreiben die *gleiche* Kollektion.

Daß bedeutet, daß die angegebene Definition von $select_E$ in allen Fällen wohldefiniert ist. □

Axiome mit einer ähnlichen Intention verwenden wir ebenfalls für die Behandlung des `collect`-Operators.

Man beachte ferner, daß die freien Variablen aus der Übersetzung $\lceil$`c`$\rceil$ des Kollektionsausdrucks `c` *nicht* in die Parameter-Menge des neuen Funktionssymbols $select_E$ eingehen, da die Übersetzung $\lceil$`c`$\rceil$ der zugehörige Kollektion explizit als zusätzlicher Parameterwert übergeben wird!

Da in vielen Fällen der Filter-Ausdruck `b` außer der Iteratorvariablen `e` keine weitere Variable enthält, hat das Vorgehen, die zu verarbeitende Kollektion `c` direkt als Parameter zu übergeben, anstatt sie implizit über die Axiome und zusätzliche Parameter aus den freien Variablen in `c` zu berücksichtigen, den Vorteil, syntaktisch sehr nahe an OCL zu sein.

Da der `reject`-Operator in E = `c->reject(e|b)` gerade die Elemente `e` aus der Kollektion `c` auswählt, die die Eigenschaft `b` *nicht* haben, behandeln wir die Übersetzung dieses Ausdrucks E auf genau dieselbe Weise und ersetzen in den obigen Axiomen die Filterbedingung `b` durch ihre Negation und umgekehrt, d.h. wir übersetzen E wie den Ausdruck `c->select(e| not b)`.

Beispiel 23
Die Menge der verheirateten erwachsenen Personen, die bei einem Unternehmen `cmp` beschäftigt sind wird durch
E = `cmp.employees->select(e| e.age >= 18 and e.married)`.

Als Resultat der Abbildung erhalten wir für E

$$\lceil \texttt{E} \rceil = select_E(cmp.employees)$$

sowie folgende Axiome in Ax_E:

$$\begin{aligned}
&select_E(emptySet_{Person}) \doteq emptySet_{Person}\\
&\forall s{:}Set_{Person}\dot{\forall} p{:}Person\,(\\
&\quad (p.age \geq 18 \wedge (p.married \doteq true)) \rightarrow\\
&\quad select_E(insert(s,p)) \doteq insert(select_E(s),p))\\
&\forall s{:}Set_{Person}\dot{\forall} p{:}Person\,(\\
&\quad \neg(p.age \geq 18 \wedge (p.married \doteq true)) \rightarrow\\
&\quad select_E(insert(s,p)) \doteq select_E(s))
\end{aligned}$$

- `collect`. Der `collect`-Operator erlaubt mathematisch betrachtet die Berechnung des Bildes eines Ausdrucks `exp` (des Typs `S`) über einem bestimmten Urbildbereich – einer Kollektion `c`. Genauer gesagt, bezeichnet E = `c->collect(e|exp)` die Multimenge (wenn `c` eine Menge oder Multimenge ist) bzw. die Sequenz (falls `c` eine Sequenz ist), die für jede mögliche Instanziierung von `e` mit einem Element der Kollektion `c` den entsprechenden Wert der Auswertung von `exp` (unter dieser Instanziierung) enthält.

  Dieser Operator kommt in OCL oft – implizit oder explizit – in der Form von nacheinander ausgeführten Navigationen über Assoziationen im Klassendiagramm $\mathcal{D}$ vor und hat damit eine große Bedeutung für die Ausdrucksfähigkeit von OCL.

  Die Behandlung dieses Operators geschieht nun mit der gleichen Technik wie im Falle der Operatoren `select` bzw. `reject`:

Wir führen ein neues Symbol $collect_E$ in Σ^* ein, das mittels Axiomen in seinem operativen Verhalten über dem strukturellen Aufbau eines Terms des zugehörigen Kollektions-ADTs definiert wird.

Seien $p_1, \ldots, p_n$ wieder genau die freien Variablen aus der Übersetzung $\lceil \texttt{exp} \rceil$ des zu während der Iteration zu berechnenden Ausdrucks – mit Ausnahme der durch E gebundenen Variablen e – und $T_1, \ldots, T_n$ die zugehörigen Sorten. Seien $e'{:}T$ und $s{:}Set_T$ neue Variablen, die nicht frei in $\lceil \texttt{exp} \rceil$ vorkommen.

Entsprechend der Sorte des Terms $\lceil \texttt{c} \rceil$ besitzt das neue Symbol die Signatur

$$\begin{array}{ll} collect_E{:}Set_T \times T_1 \times \ldots \times T_n \to Bag_S & \textit{oder} \\ collect_E{:}Bag_T \times T_1 \times \ldots \times T_n \to Bag_S & \textit{oder} \\ collect_E{:}Sequence_T \times T_1 \times \ldots \times T_n \to Sequence_S & \end{array}$$

Dann verwenden wir

$$\lceil \texttt{E} \rceil = collect_E(\lceil \texttt{c} \rceil, p_1, \ldots, p_n)$$

und fügen (für *Multimengen* c) zur Axiomatisierung des neuen Symbols $collect_E$ folgende Formeln in Ax_E ein:

$$\begin{array}{l} \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n\, collect_E(emptyBag_T, p_1, \ldots, p_n) \doteq emptyBag_S \\ \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall b{:}Bag_T \dot\forall e'{:}T\ (\\ \quad collect_E(insert(b, e'), p_1, \ldots, p_n) \doteq \\ \quad insert(collect_E(b, p_1, \ldots, p_n), \lceil \texttt{exp} \rceil \{e/e'\})) \end{array}$$

Im dem Falle, daß es sich bei c um eine Menge (bzw. Sequenz) handelt, gehen wir exakt in der gleichen Weise vor und ersetzen lediglich die Konstantensymbol $emptyBag_T$ durch $emptySet_T$ (bzw. $emptySequence_T$). Für Sequenzen muß außerdem statt dem Konstantensymbol $emptyBag_S$ das Symbol $emptySequences_S$ benutzt werden.

Für *Mengen* muß zusätzlich das zweite Axiome leicht verändert werden, um eine wohldefinierte Definition zu erhalten:

$$\begin{array}{l} \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall s{:}Set_T \dot\forall e'{:}T\ (\\ \quad collect_E(insert(s, e'), p_1, \ldots, p_n) \doteq \\ \quad insert(collect_E(remove(s, e), p_1, \ldots, p_n), \lceil \texttt{exp} \rceil \{e/e'\})) \end{array}$$

Man beachte, daß diese Erweiterung für Mengen im Falle von `select` *nicht* notwendig war, da unter diesen Umständen dort als Ergebnis immer eine Menge entsteht und die in die Ergebnismenge eingefügten Elemente immer mit den ursprünglichen Elementen übereinstimmen. Mehrfacheinfügungen eines Elements in einer Termdarstellung einer führen dort deshalb zu einer Beschreibung der gleichen Ergebnismenge. Hier ergibt sich hier jedoch als Ergebnismenge unter diesen Umständen eine Multimenge und Mehrfachbehandlungen desselben Elements führen zu Beschreibungen von *unterschiedlichen* Ergebnismengen, sofern das *remove* in dem angegebenen Axiom nicht verwendet wird, was nicht korrekt wäre.

Die Quantor-Variable $s{:}Set_T$ wird entsprechend dem Typ des Kollektionsausdrucks c zu $b{:}Bag_T$ bzw. $s{:}Sequence_T$ angepaßt, wobei auch diese Variablen nicht frei in $\lceil \texttt{exp} \rceil$ vorkommen.

Es gibt jedoch noch eine wichtige Eigenheit dieses Operators: OCL kennt keine geschachtelten Kollektionen. Anstatt Kollektionen von Kollektion zu bilden, werden die einzelnen Kollektionen in OCL einfach verschmolzen bzw. *geflattet*[36]. Um diese Eigenheit korrekt zu behandeln, ersetzen wir in dem Fall, daß der OCL-Ausdruck `exp` einen Kollektionstypen besitzt, das Funktionsymbol *insert* (zum Einfügen eines Elements in eine Kollektion) in den obigen Axiomen durch das Funktionsymbol *union* (um die Elemente der bei Auswertung von `exp` entstehenden Kollektionen zu verschmelzen). Das zweite Axiom lautet dann (für Mengen `c`):

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \forall s{:}Set_T \dot{\forall} e'{:}T\ (\\ collect_E(insert(s,e'),p_1,\ldots,p_n) \doteq \\ union(collect_E(remove(s,e),p_1,\ldots,p_n), \lceil \texttt{exp} \rceil \{e/e'\}))$$

Bemerkung (Wohldefiniertheit der Definition). Wir Fragen uns wieder, ob die angegebene Definition wohldefiniert ist:

In allen Fällen macht man sich wieder durch Induktion klar, daß der Ergebnisterm der Funktion unter der angegebenen Definition von $collect_E$ für eine beliebige Termdarstellung einer Kollektion eindeutig festgelegt ist.

Wir wollen außerdem wieder erreichen, daß für *alle* Termdarstellungen *einer Kollektion* die obige Definition ein Ergebnis liefert, das die jeweils die gleiche Ergebniskollektion beschreibt!

Im Falle von Sequenzen ist die obige Definition also wegen der Eindeutigkeit der Termdarstellung des Ergebnis wohldefiniert. Desweitern macht man sich schnell durch einfache Fallunterscheidung klar das die Reihenfolge der Elemente in einer Termdarstellung keine Rolle spielt, d.h. daß für alle Termdarstellungen einer Multimenge bzw. Menge c und alle Elemente $e_1, e_2{:}T, p_1{:}T_1, \ldots, p_n{:}T_n$ gilt:

$collect_E(insert(insert(c,e_1),e_2),p_1,\ldots,p_n)$ und
$collect_E(insert(insert(c,e_2),e_1),p_1,\ldots,p_n)$
beschreiben die *gleiche* Kollektion.

Somit ist die Definition auch für Multimengen wohldefiniert. Für die Wohldefiniertheit im Falle von Mengen benötigen wir noch die Unabhängigkeit von der Anzahl der Einfügungen eines Elements. Man macht sich auch hier durch einfache Fallunterscheidung klar, daß unter der angegebenen Definition[37] für jede Termdarstellung einer Menge c und alle Elemente $e{:}T, p_1{:}T_1, \ldots, p_n{:}T_n$ gilt:

$collect_E(insert(c,e),p_1,\ldots,p_n)$ und
$collect_E(insert(insert(c,e),e),p_1,\ldots,p_n)$
beschreiben die *gleiche* Kollektion.

Daß bedeutet, daß die angegebene Definition von $collect_E$ in allen Fällen wohldefiniert ist. □

Man beachte wieder, daß die freien Variablen aus der Übersetzung $\lceil \texttt{c} \rceil$ des Kollektionsausdrucks `c` *nicht* in die Parameter-Menge des neuen Funktionssymbols

[36] Das ist auch der Grund, warum bei hintereinander geschachtelten Navigationen in OCL eine Multimenge entsteht!

[37] Die Anwendung von *remove* hat hier wesentlichen Einfluß auf diese Eigenschaft!

$collect_E$ eingehen, da die Übersetzung $\lceil$`c`$\rceil$ der zugehörige Kollektion explizit als zusätzlicher Parameterwert übergeben wird!

Beispiel 24
Die (Multi-)Menge aller Geburtstage von Kunden eines Unternehmens `cmp` wird durch $E =$ `cmp.customers->collect(c| c.birthDate)` in OCL berechnet.

Als Übersetzung erhalten wir für E erhalten wir:

$$\lceil \texttt{E} \rceil = collect_E(cmp.customers)$$

sowie folgende Axiome in Ax_E:

$$collect_E(emptySet_{Customer}) \doteq emptyBag_{Date}$$
$$\forall s{:}Set_{Customer} \dot{\forall} c{:}Customer\ ($$
$$\quad collect_E(insert(s,c)) \doteq insert(collect_E(remove(s,c)), c.birthDate))$$

Bemerkung (Optimierung für Mengen). Man sieht recht einfach, daß der *remove*-Operator im zweiten Axiom zur Definition von $collect_E$ im Falle von Mengen nur deshalb notwendig ist, da der Bildwert $exp(e)$ bei einem Mehrfachvorkommen eines Elements e in der Termdarstellung t_s einer Menge s auch mehrfach in der Termdarstellung t_b der Ergebnismultimenge $t_b = collect_E(t_s)$ eingefügt würde und somit bei äquivalenten Argumenten die Funktion $collect_E$ nichtäquivalente Ergebnisse liefert.

Das wäre in einem speziellen Falle aber gar *nicht* tragisch: Wenn der `collect`-Ausdruck in einem Kontext verwendet wird, in dem die Anzahl der Vorkommen der einzelnen Elemente aus der Multimenge b gar nicht gebraucht werden, und somit die Multimenge eigentlich wie eine Menge behandelt wird. In diesem Falle könnten wir das *remove* auch weglassen und die Übersetzung vereinfachen.

Doch wie erkennt man eine solchen Situtation? Wir entwicklen später – wenn auch in etwas anderem Zusammenhang – in Abschnitt 3.3 eine algorithmische Vorschrift, die es gestattet genau diese Situation zu erkennen. Wir können also in bestimmten Fällen eine Optimierung der Übersetzung von `collect` angeben.

□

- `iterate`. Der `iterate`-Operator ist das komplexeste und mächtigste Sprachkonstrukt in OCL. Viele der anderen Operatoren auf Kollektionen lassen sich auf diesen Operator zurückführen. Eine Behandlung ist daher sehr interessant und wird erwartungsgemäß ähnlich komplex ausfallen. Wir geben hier zunächst ein Verfahren an, das für prinzipiell alle hier betrachteten Logiksprachen anwendbar ist. In Abschnitt 3.6 werden wir separat eine alternative Darstellung untersuchen, die speziell für die **DL** als Zielsprache geeignet sind.

  Sei `c` ein OCL-Ausdruck eines beliebigen Kollektionstyps, `e0` ein OCL-Ausdruck des Typs `T'`, sowie `exp` ein OCL-Ausdruck des Typs `T'`.

  Der Auswertung des OCL-Ausdrucks

  $$E =$$ `c->iterate(e:T; acc:T'= e0 | exp)`

ergibt sich nun folgendermaßen:

Zunächst wird die Variable `acc` mit dem Wert initialisiert, der durch die Auswertung des Ausdrucks `e0` ensteht. Die Variable `e` iteriert nun schrittweise[38] über alle Elemente e in der Kollektion `c`. In jedem Schritt wird für die aktuelle Belegung der beiden Variablen `e` und `acc` der Ausdruck `exp` (der im allgemeinen diese beiden Variablen frei enthält) ausgewertet und dieser Wert der Variablen `acc` als neue Belegung zugewiesen. Sind alle Elemente der Kollektion bearbeitet, so liefert schließlich die Belegung der Variablen `acc` den Wert der gesamtes `iterate`-Ausdrucks E – d.h. insbesondere für leere Kollektionen `c` entspricht die Auswertung von E gerade dem Wert des Initialisierungsausdrucks `e0`.

Die Behandlung dieses Operators können wir nun abermals mit der gleichen Technik wie im Falle der Operatoren `select` bzw. `collect` vornehmen und stellen zunächst die Behandlung auf der allgemeinsten Ebene vor:

Seien $p_1, \ldots, p_n$ die freien Variablen aus $\lceil$`exp`$\rceil$ und $\lceil$`e0`$\rceil$, die sich von e und acc verschieden sind, und $T_1, \ldots, T_n$ die zugehörigen Sorten. Seien dabei OBdA. $p_1, \ldots, p_j$ $(0 \leq j \leq n)$ gerade die freien Variablen in $\lceil$`exp`$\rceil$, die sich von e und acc verschieden sind.

Wir führen vier neue Symbole $itSequence_E$, it_E, $iterate_E$ und exp_E in Σ^* ein.

Das Symbol exp_E hat die Signatur

$$exp_E{:}T \times T' \times T_1 \times \ldots \times T_j \to T'$$

dient dabei zur Berechnung des neuen Wertes des Akkumulators acc aus dem gerade besuchten Element in der Iteration und dem aktuellen Akkumulatorwert: $exp_E(e, acc, p_1, \ldots, p_j)$ stellt gerade den neuen Wert des Akkumulators (in Abhängigkeit der Werte $p_1, \ldots, p_j$) dar, wenn gerade das Element e aus Kollektion betrachtet wird und der aktuelle Akkumulatorwert acc entspricht. Das Symbole it_E mit der Signatur

$it_E{:}Set_T \times T_1 \times \ldots \times T_n \to T'$	(im Fall einer Menge `c`) bzw.
$it_E{:}Bag_T \times T_1 \times \ldots \times T_n \to T'$	(für Multimengen `c`) oder
$it_E{:}Sequence_T \times T_1 \times \ldots \times T_n \to T'$	(für Sequenzen `c`)

wird zur schrittweisen Berechnung des Ergebnis einer Iteration in Form einer Schachtelung von exp_E-Termen entlang der Termdarstellung der nach der Iterationsreihenfolge geordneten Kollektion `c` verwendet und verkörpert die eigentliche Iteration.

Schließlich besitzt das Funktionssymbole $iterate_E$ genau die gleiche Signatur wie it_E und liefert den Wert des gesamten `iterate`-Ausdrucks E nach einer Iteration über `c` dar.

Diese Bedeutung der neuen Symbole wird wieder durch geeignete, neue Axiome in Ax_E festgehalten.

[38] Im Falle von Sequenzen entspricht die Iterationsreihenfolge gerade der Reihenfolge der Elemente in der Sequenz. Ansonsten ist die Bearbeitungsfolge der Elemente in OCL *nicht* festgelegt, was prinzipell zu nicht wohldefinierten Ausdrücken führen kann, wenn der Modellierer bei der Formulierung eines `iterate`-Ausdrucks nicht sorgfälltig genug vorgeht.

Wir verwenden somit im allgemeinsten Falle prinzipell wie folgt vor:

$$\lceil \texttt{E} \rceil = iterate_E(\lceil \texttt{c} \rceil, p_1, \ldots, p_n)$$

und benutzen zur Axiomatisierung der neuen Symbole grundsätzlich folgende Formeln in Ax_E ein:

$$\begin{array}{l}
\dot{\forall} e{:}T \dot{\forall} acc{:}T' p_1{:}T_1 \ldots \dot{\forall} p_j{:}T_j \; (exp_E(e, acc, p_1, \ldots, p_j) \doteq \lceil \texttt{exp} \rceil) \\
\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \; (it_E(emptySequence_T, p_1, \ldots, p_n) \doteq \lceil \texttt{e0} \rceil) \\
\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \forall s{:}Sequence_T \dot{\forall} e{:}T \; (it_E(insert(s, e), p_1, \ldots, p_n) \doteq \\
\qquad\qquad exp_E(e, iterate_E(s, p_1, \ldots, p_n), p_1, \ldots, p_j)) \\
\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \forall c{:}Collection_T \; (iterate_E(c, p_1, \ldots, p_n) \doteq \\
\qquad\qquad it_E(itSequence(c), p_1, \ldots, p_n))
\end{array}$$

wobei das Symbol $itSequence$ ein Funktionssymbol mit der Signatur

$itSequence{:}Set_T \to Sequence_T$	(im Fall einer Menge `c`) bzw.
$itSequence{:}Bag_T \to Sequence_T$	(für Multimengen `c`) oder
$itSequence{:}Sequence_T \to Sequence_T$	(für Sequenzen `c`)

ist und zur Sortierung der Kollektion `c` entsprechend der Reihenfolge, in der OCL über die Kollektion iteriert, dient.

Bemerkung (OCL-Semantik und Wohldefiniertheit der Definition). Unter der Annahme, daß OCL eine saubere, eindeutige Definition der Iterationsreihenfolge – und damit von $itSequence$ – angibt, kann man wieder zeigen, daß die angegebene Definition von $iterate_E$ wohldefiniert ist. In jedem Fall ist aber die Iterations-Funktion it_E wohldefiniert.

Doch leider ist die Annahme *nicht* immer gewährleistet, was ein Schwäche in der *Definition der Semantik von OCL* darstellt: OCL legt lediglich für *Sequenzen* ein solche Reihenfolge fest: Die Iterationsreihenfolge entspricht dann gerade der Reihenfolge der Elemente in der Sequenz `c`. In den anderen Fällen wird *keine* Iterationsordung spezifiziert, was eine bedeutende Konsequenz hat: Im allgemeinen ist dadurch der Wert `iterate`-Ausdruck über Mengen und Multimengen *nicht* eindeutig festgelegt, der Ausdruck E gewissermaßen undefiniert. Die Eindeutigkeit (in OCL selbst!) läßt sich nur dann in diesen Fällen gewährleisten, wenn die Auswertung der Berechnungsvorschrift nicht von der Reihenfolge, in der die Elemente besucht werden, abhängt. Beispiele dafür haben wir schon gesehen: Die Berechnungsvorschrift, die entsteht, wenn man den `select` oder `collect`-Ausdruck durch einen `iterate`-Ausdruck darstellt, hat genau diese Eigenschaft, weshalb diese Operatoren in OCL wohldefiniert sind, obwohl sie in der OCL-Spezifikation [Obj99a, Obj01] durch einen `iterate`-Ausdruck definiert wurden und OCL diese semantische Schwäche enthält.

Es obliegt also grundsätzlich dem *Modellierer* bei der Formulierung von `iterate`-Ausdrücken, darauf zu achten, daß die angegebene Berechnungsvorschrift `exp`, derart gewählt wurde, daß deren Ergebnis *nicht* von der Iterationsreihenfolhe abhängt. □

Für Sequenzen entspricht also die Iterationsreihenfolge gerade der Reihenfolge Vorkommen der Elemente in der Sequenz, also

$$\forall s{:}Sequence_T(itSequence(s) \doteq s)$$

Damit lassen sich die angegebenen Axiome vereinfachen. Insbesondere wird das Symbol $itSequence$ und die Unterscheidung zwischen $iterate_E$ und it_E nicht benötigt.

Wir verwenden somit im Falle von *Sequenzen* `c`

$$\lceil \mathtt{E} \rceil = iterate_E(\lceil \mathtt{c} \rceil, p_1, \ldots, p_n)$$

und fügen zur Axiomatisierung der neuen Symbole folgende Formeln in Ax_E ein:

$$\begin{array}{l}
\dot\forall e{:}T \dot\forall acc{:}T' p_1{:}T_1 \ldots \dot\forall p_j{:}T_j\ (exp_E(e, acc, p_1, \ldots, p_j) \doteq \lceil \mathtt{exp} \rceil) \\
\dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n\ (iterate_E(emptySequence_T, p_1, \ldots, p_n) \doteq \lceil \mathtt{e0} \rceil) \\
\dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall s{:}Sequence_T \dot\forall e{:}T\ (iterate_E(insert(s, e), p_1, \ldots, p_n) \doteq \\
\qquad\qquad exp_E(e, iterate_E(s, p_1, \ldots, p_n), p_1, \ldots, p_j))
\end{array}$$

Bemerkung (Wohldefiniertheit der Definition für Sequenzen). Die angegebene Definition von $iterate_E$ ist im Falle von *Sequenzen* wohldefiniert: Man zeigt durch Induktion sehr einfach, daß der Ergebnisterm der Funktion unter der angegebenen Definition von $iterate_E$ für eine beliebige Termdarstellung der Sequenz eindeutig festgelegt ist. Da für eine beliebige Sequenz die zugehörige Termdarstellung eindeutig ist, erhalten wir insgesamt eine wohldefinierte Funktion. □

Im dem Falle, daß es sich bei `c` um eine Menge oder Multimenge handelt, muß die Berechnungsvorschrift `exp` – betrachtet als Funktion $exp(e, acc, p_1, \ldots, p_j)$ – die folgende Eigenschaft aufweisen (eine Art „Assoziativität"), damit der OCL-Ausdruck E überhaupt wohldefiniert ist:

$$\begin{array}{l}
\text{Für alle } e{:}T, acc{:}T', p_1{:}T_1, \ldots p_j{:}T_j \text{ gilt:} \\
\qquad exp(e_1, exp(e_2, acc, p_1, \ldots, p_j), p_1, \ldots, p_j) = \\
\qquad exp(e_2, exp(e_1, acc, p_1, \ldots, p_j), p_1, \ldots, p_j)
\end{array}$$

wobei e die Iteratorvariable, acc der Akkumulator und $p_1, \ldots, p_j$ die restlichen freien Variablen aus `exp` sind.

Diesen Tatsache können wir nun bei der Formulierung der Axiome für den Fall von *Multimengen und Mengen* ausnutzen: Hat die Berechnungsvorschrift exp diese Eigenschaft, so ist die *Reihenfolge*, in der die Elemente während der Iteration besucht werden für das Endergebnis irrelevant. Wir benötigen somit nicht die (im Sinne von OCL[39]) richtig geordnete Termdarstellung der zu iterierenden Kollektion, sondern lediglich eine beliebig permutierte. Für *Multimengen* bedeutet das, daß wir bei den oben angegebenen Axiomen für den allgemeinsten Fall auf die Sortierfunktion $itSequence$ verzichten und it_E direkt auf die zu behandelnde Kollektion `c` anwenden können, ohne die Wohldefiniertheit von $iterate_E$ zu zerstören.

[39] Wenn es denn einen solchen Sinne gäbe!

Wir haben für die formale Darstellung im Falle von Multimengen und Mengen außerdem zwei grundsätzliche Optionen zur Verfügung: Die erste und einfachere Variante verkörpert genau die Informationen, die in der OCL-Spezifikation festgehalten sind, und kümmert sich gar nicht darum, daß die Prämisse der Reihenfolgeunabhängigkeit gewährleistet sein muß, d.h. die mögliche Undefiniertheit des OCL-Ausdrucks. Die Übersetzung macht in diesem Fall nur dann Sinn, wenn auch der ursprüngliche OCL-Ausdruck Sinn macht. Die Verantwortung obliegt dem Modellierer!

In diesem Fall benötigen wir dann keine Unterscheidung zwischen den Symbolen $iterate_E$ und it_E und können direkt mit $iterate_E$ arbeiten. Wir dann erhalten als zusätzliche Axiome in Ax_E

$$\begin{aligned}
&\dot\forall e{:}T \dot\forall acc{:}T' p_1{:}T_1 \ldots \dot\forall p_j{:}T_j \; (exp_E(e, acc, p_1, \ldots, p_j) \doteq \lceil \texttt{exp} \rceil) \\
&\dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \; (iterate_E(emptyBag_T, p_1, \ldots, p_n) \doteq \lceil \texttt{e0} \rceil) \\
&\dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall b{:}Bag_T \dot\forall e{:}T \; (iterate_E(insert(b, e), p_1, \ldots, p_n) \doteq \\
&\qquad\qquad exp_E(e, iterate_E(b, p_1, \ldots, p_n), p_1, \ldots, p_j))
\end{aligned}$$

Bemerkung (Wohldefiniertheit der Definition für Multimengen). Unter der Prämisse der Reihenfolgeunabhängigkeit der Berechnungsvorschrift gilt, daß die angegebene Definition für $iterate_E$ wohldefiniert ist. □

Für Mengen können wir genauso vorgehen, müssen jedoch dafür sorgen, daß die mehrfache Behandlung eines Elementes in der Termdarstellung ausgeschlossen ist und fügen daher – wie bei `collect` – ein *remove* im dritten Axiom ein, benutzen also statt diesem Axiom die folgende Formel:

$$\begin{aligned}
&\dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall s{:}Set_T \dot\forall e{:}T \; (\\
&\quad iterate_E(insert(s, e), p_1, \ldots, p_n) \doteq \\
&\quad exp_E(e, iterate_E(remove(s, e), p_1, \ldots, p_n), p_1, \ldots, p_j))
\end{aligned}$$

Ansonsten wird in den restlichen Formeln die Konstante $emptyBag_T$ durch $emptySet_T$ und die Quantorvariable $b{:}Bag_T$ durch $s{:}Set_T$ ersetzt.

Bemerkung (Wohldefiniertheit der Definition für Mengen). Unter der Prämisse der Reihenfolgeunabhängigkeit der Berechnungsvorschrift gilt, daß die angegebene Definition für $iterate_E$ wohldefiniert ist. □

Nun zur zweiten Variante, die komplexer aber in gewissem Sinne sicherer ist: Wir können die Mehrdeutigkeit des `iterate`-Ausdrucks im Falle eines Bruchs der erforderlichen Anforderung an die Berechnungsvorschrift – also einem unbewußten Modellierungsfehler! – *explizit* durch eine Unterspezifikation des Ergebniswertes modellieren:

Wir brauchen in diesem Fall die Unterscheidung zwischen dem Ergebnis der Iteration it_E und dem eigentlichen Ergebnis $iterate_E$ zur Weiterverarbeitung und ein zusätzliches Axiom, welches die beiden Symbole verknüpft. Für Multimengen

ergäbe das insgesamt die folgende Formelmenge:

$$\dot{\forall} e{:}T \dot{\forall} acc{:}T' p_1{:}T_1 \ldots \dot{\forall} p_j{:}T_j \, (exp_E(e, acc, p_1, \ldots, p_j) \doteq \lceil \texttt{exp} \rceil)$$
$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, (it_E(emptyBag_T, p_1, \ldots, p_n) \doteq \lceil \texttt{e0} \rceil)$$
$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \dot{\forall} b{:}Bag_T \dot{\forall} e{:}T \, (it_E(insert(b, e), p_1, \ldots, p_n) \doteq exp_E(e, it_E(b, p_1, \ldots, p_n), p_1, \ldots, p_j))$$
$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_j{:}T_j \, ($$
$$\quad \dot{\forall} e_1, e_2{:}T \, acc{:}T' (exp_E(e_1, exp_E(e_2, acc, p_1, \ldots, p_j), p_1, \ldots, p_j) \doteq exp_E(e_2, exp_E(e_1, acc, p_1, \ldots, p_j), p_1, \ldots, p_j))$$
$$\quad \rightarrow \dot{\forall} p_{j+1}{:}T_{j+1} \ldots \dot{\forall} p_n{:}T_n \dot{\forall} b{:}Bag_T \dot{\forall} e{:}T \, (iterate_E(b, p_1, \ldots, p_n) \doteq it_E(b, p_1, \ldots, p_n)))$$

Für Mengen kann man wieder analog vorgehen und muß jedoch die *remove*-Operation einfügen.

Bemerkung (Sicherheit vs. Einfachheit). Man könnte an dieser Stelle die Position vertreten, daß sich ein Modellierer dieser Schwäche von OCL bezüglich der Anforderung an die Berechnungsvorschrift im `iterate`-Operator bewußt sein sollte und damit keine ungültigen OCL-Ausdrücke formuliert und die Übersetzung sich an dieser Stelle darauf verlassen kann. Wenn man so entscheidet, dann wäre eine einfachere Behandlung für Multimengen und Mengen zu bevorzugen.

Will man sich hingegen nicht blind darauf verlassen, daß der Modellierer vernünftig mit `iterate`-Ausdrücken umgeht und sieht somit die sichere Behandlung als notwendig an, so erkauft man die Sicherheit um einen Preis: Die Komplexität steigt. Es ist jedoch recht wahrscheinlich, daß ein, das ein Software-Ingenieur, der OCL zur Modellierung verwendet und sich ausschließlich mit der OCL-Spezifikation [Obj99a, Obj01] oder dem Buch [WK99] beschäftigt, sich *nicht* explizit der Notwendigkeit der Reihenfolgeunabhängigkeit des Berechnungsausdrucks bewußt ist – in diesen Dokumenten wird nirgends darauf hingewiesen! – und somit diese Anforderung an die Modellierung übersehen kann, ohne sich über die Folgen klar zu sein. Spätestens beim Arbeiten mit den generierten Formeln sollte dann für den Modellierer klar werden, daß irgendetwas mit dem ursprünglichen OCL-Ausdruck – also der Modellierung – nicht stimmt. Insofern kann die angegebenere komplexere Übersetzung zur Fehlersuche in der OCL-Spezifikation dienen. Wir begegnen somit der möglichen Undefiniertheit des `iterate`-Ausdrucks E in einer Instanziierung D durch die Unterspezifikation des entsprechenden Funktionssymbols $iterate_E$ unter der korrespondierenden Σ^*-Struktur $\mathcal{S}_D$. □

Wir wollen uns hier aus Platzgründen auf die einfachere Variante festlegen, ziehen aber prinzipiell die sichere Variante vor. In einer konkreten Anwendung wäre auch denkbar, daß der Benutzer bei der Übersetzung einer UML/OCL-Spezifikation festlegen kann, welche Variante gewünscht ist. So könnte ein ,,erfahrener" Modellierer die einfache Übersetzung anwenden, wohingegen ein ,,unerfahrener" Modellierer die sichere Variante wählen könnte.

Man beachte wieder, daß die freien Variablen aus der Übersetzung $\lceil \texttt{c} \rceil$ des Kollektionsausdrucks `c` *nicht* in die Parameter-Menge der neuen Funktionssymbole

$iterate_E$ und exp_E eingehen, da die Übersetzung $\lceil$`c`$\rceil$ der zugehörige Kollektion explizit als zusätzlicher Parameterwert übergeben wird!

Beispiel 25
Wir betrachten nochmals den OCL-Ausdruck aus Beispiel 24:

Mit E' = `cmp.customers->collect(c| c.birthDate)` beschrieben wir dort die Kollektion aller Geburtstage von Kunden eines Unternehmens `cmp`.

Wir formulieren diese Mehrfachmenge nun unter Verwendung des `iterate`-Operators durch $E =$

```
cmp.customers->iterate(c:Customer;
                       acc:Bag(Date) = Bag{} |
                           acc->including(c.birthDate))
```

Als Übersetzung erhalten wir für E erhalten wir:

$$\lceil \texttt{E} \rceil = iterate_E(cmp.customers)$$

sowie folgende Axiome in Ax_E:

$$\begin{array}{l}
\dot\forall c{:}Customer \dot\forall acc{:}Bag_{Date}\ (exp_E(c, acc) \doteq insert(acc, c.birthDate)) \\
iterate_E(emptySet_{Customer}) \doteq emptyBag_{Date} \\
\forall s{:}Set_{Customer} \dot\forall c{:}Customer\ (iterate_E(insert(s, c)) \doteq \\
\qquad\qquad exp_E(e, iterate_E(remove(s, c)))
\end{array}$$

Durch einfache syntaktische Ersetzung von $exp_E(e, iterate_E(remove(s, e)))$ im dritten Axiom entsprechend dem ersten Axiom erhalten wir genau dieselbe Definition für $iterate_E$ wie im Beispiel 24 für $collect_E$:

$$\begin{array}{l}
collect_E(emptySet_{Customer}) \doteq emptyBag_{Date} \\
\forall s{:}Set_{Customer} \dot\forall c{:}Customer\ (\\
\quad collect_E(insert(s, c)) \doteq insert(collect_E(remove(s, c)), c.birthDate))
\end{array}$$

Man könnte somit die durch $\lceil E$ `=` $E' \rceil$ verkörperte Aussage sehr leicht durch einfache Induktion über den Aufbau von *cmp.customers* beweisen.

Das Beispiel weist somit auf eine mögliche Optimierung des bisherigen Vorgehens hin: Die im Beispiel benutzte Vereinfachung durch syntaktische Ersetzung ist *immer* möglich, d.h. wir können prinzipiell in jedem Fall auf die Einführung des Synbols exp_E und damit auch auf das erste Axiom verzichten. Das dritte Axiom müßte dann – für Mengen – wie folgt abgewandelt werden:

$$\begin{array}{l}
\dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall s{:}Set_T \dot\forall e{:}T\ (iterate_E(insert(s, e), p_1, \ldots, p_n) \doteq \\
\qquad\qquad \lceil \texttt{exp} \rceil \{acc/iterate_E(remove(s, e), p_1, \ldots, p_n)\})
\end{array}$$

Für Multimengen uns Sequenzen entfällt wieder die *remove* Operation, d.h. wir ersetzen außerdem den Term $remove(s, e)$ durch den Term e.

Bemerkung (Verwendung von exp_E). Die Verwendung des Symbols exp_E ist zwar nicht zwingend, kann unter Umständen aber (an anderer Stelle) trotzdem sinnvoll sein: Möchte man nämlich die komplexere Übersetzung anwenden, die die Reihenfolgeunabhängigkeit für exp_E explizit in der Formel fordert bzw. überprüft, so ist die Einführung eines Symbols (anstelle der einfachen syntaktischen Ersetzung) dann sinnvoll, wenn exp_E ein komplexer Term ist und damit die Teilformel zum Nachweis der Reihenfolgeunabhängigkeit – als Ganzes betrachtet – undurchsichtig würde. □

Wie man unschwer erkennt, ist diese Übersetzung wesentlich komplizierter, als die der anderen Operatoren aus OCL, – was prinzipiell mit der Mächtigkeit dieses Operators zu tun hat – aber immer noch im Rahmen prädikatenlogischer Ausdrucksmittel, sofern man generierte ADTs verwendet[40].

Die erzeugte Formel kann vereinfacht werden, wenn man Zusatzwissen über die in jedem Schritt berechnete Formel besitzt: Weiß man beispielsweise (durch eine syntaktische Analyse des Ausdrucks E), daß die Berechnungsvorschrift zulässig ist (also die Formel zur Reihenfolgeunabhängigkeit erfüllt), so kann man die zugehörige Formel als Prämisse wegfallen lassen und damit eine einfachere Übersetzung verwenden[41].

Man beachte desweiteren, daß sich viele andere OCL-Operatoren `o` durch einen `iterate`-Ausdruck `it-expr` darstellen lassen. Mit einem konkreten Übersetzungsverfahren von `iterate` erhält man durch einfache Abbildung des Ausdrucks `it-expr` daher eine alternative Übersetzung des Operators `o` selbst.

Man kann diese Möglichkeit der Simulation von OCL-Operatoren (auf OCL-Ebene) nun auf zwei verschiedene Weisen gewinnbringend zu nutzen versuchen:

Zum einen ist es unter diesen Umständen möglich, die alternative Übersetzung des Operators `o` mit der ursprünglichen Übersetzung zu vergleichen, um Erkenntisse darüber zu gewinnen, in welchen *speziellen Fällen* Vereinfachungen für die Übersetzung von `iterate` selbst möglich sind – d.h. man entwickelt schlußendlich allgemeine Heuristiken, die die Behandlung des `iterate` in bestimmten Fällen verbessern. Wir werden diesen sehr interessanten Aspekt in dieser Arbeit leider nicht mehr genauer untersuchen können.

Zum anderen erhält man aber auch eine *neue* Übersetzung für den OCL-Operator `o` selbst, die in bestimmten Kontexten Vorteile gegenüber der bestehenden Behandlung bieten kann. Das bedeutet insbesondere, daß es sehr gewinnbringend sein kann, nach einer *anderen*, möglicherweise recht verschiedenen Übersetzung

[40] In [Sch01a] wird gezeigt, daß aus logischer Sicht ein solcher `iterate`-Operator aus einer Prädikatenlogik erster Stufe (mit einer linearen Ordnung) eine Logik – *Iterate Logic* – erzeugt, die über *endlichen* Strukturen echt ausdruckstärker ist, als reine Prädikatenlogik erster Stufe (mit einer lineare Ordnung). Insofern ist die Verwendung *generierter* ADTs Set_T, Bag_T, $Sequence_T$ für Kollektionen wesentlich, um den `iterate`-Operator durch rein prädikatenlogische Sprachmittel erster Stufe darzustellen.

[41] Sehr allgemeine Aussagen werden hierzu vermutlich schwer anzugeben sein, da ein solches Verfahren, daß beispielsweise für sehr viele prädikatenlogische Terme exp_E funktioniert, im Grunde eine Art ,,Beweiser" für die zugehörigen prädikatenlogischen Formel darstellen, welche jeweils die Reihenfolgeunabhängigkeit der durch exp_E dargestellten Funktion fordern.

des `iterate`-Operators zu suchen, um für eine Vielzahl von OCL-Operatoren *alternative* Übersetzungen zu erhalten.

Zusammen mit einer (anwendungsabhängigen) Vorschrift, die dann beschreibt, unter welchen Umständen welches der möglichen Verfahren für einen Operator `o` angewendet werden soll, erhält somit potentiell eine sehr mächtige Übersetzung.

Wir werden in Abschnitt 3.6 daher andere Darstellungsformen für das `iterate`-Konstrukt untersuchen, die für das KeY-Projekt besser geeignet sein mögen.

2.3.3.8 Kollektionstypen – Set

Seien im folgenden `s`, `s1`, `s2` OCL-Ausdruck für beliebige Mengen, `c` ein OCL-Ausruck für eine Menge oder Multimenge und `o` ein Ausdruck des Basistyps `T`.

- **Gleichheit von Mengen.** Wir behandeln die Gleichheit zweier Mengen `s1`, `s2` durch eine Formel, die besagt, daß die beiden Mengen genau die gleichen Elemente haben.

  Sei `T` der kleinste gemeinsame Obertyp[42] der Elementtypen `T1` und `T2` der beiden Mengen. Sei $e{:}T$ eine neue Variable, die nicht frei in $\lceil$`s1`$\rceil$ oder $\lceil$`s2`$\rceil$ vorkommt.

  Dann lautet die gewünschte Formel

  $$\lceil \texttt{s1 = s2} \rceil = \dot{\forall} e{:}T\, (e \in \lceil \texttt{s1} \rceil \leftrightarrow e \in \lceil \texttt{s2} \rceil)$$

  Analog übersetzen wir die Ungleichheit durch die Negation dieser Formel.

- `union`, `intersection`, `including`, `excluding`, `-`, `symmetricDifference`. Für alle diese Eigenschaften gibt es ein korrespondierendes Funktionssymbol in Σ^*, das wir für die Übersetzung heranziehen. Diese werden durch entsprechende Formeln in der Spezifikation der zugehörigen ADTs axiomatisiert.

  Wir übersetzten deshalb wie folgt:

  $\lceil$`s->union(c)`$\rceil$ = $union(\lceil$`s`$\rceil, \lceil$`c`$\rceil)$
  $\lceil$`s->intersection(c)`$\rceil$ = $intersection(\lceil$`s`$\rceil, \lceil$`c`$\rceil)$
  $\lceil$`s->including(o)`$\rceil$ = $insert(\lceil$`c`$\rceil, \lceil$`o`$\rceil)$
  $\lceil$`s->excluding(o)`$\rceil$ = $remove(\lceil$`c`$\rceil, \lceil$`o`$\rceil)$
  $\lceil$`s1 - s2`$\rceil$ = $\lceil$`s1`$\rceil - \lceil$`s2`$\rceil)$
  $\lceil$`s1->symmetricDifference(s2)`$\rceil$ = $symmetricDifference(\lceil$`s1`$\rceil, \lceil$`s2`$\rceil)$

- `asBag`,`asSequence`. OCL bietet die Möglichkeit, zwischen den verschiedenen Spielarten von Kollektionen hin- und herzuwandeln.

  Der Ausdruck $E =$ `s->asBag` erzeugt eine Multimenge E, die genau die Elemente der Menge `s` enthält. Außerdem gilt für die entstehende Multimenge E, daß kein Element in der Multimenge mehrfach vorkommt.

  Wir formalisieren wieder unter Anwendung unserer Benennungstechnik wie folgt: Wir generieren ein neues Funktionssymbol $asBag_E$ mit der Ergebnissorte Bag_T in Σ^*.

[42] Man beachte die Bemerkung auf Seite 63.

Seien $p_1, \ldots, p_n$ wieder genau die freien Variablen aus der Übersetzung $\lceil \mathtt{s} \rceil$ von `s` und $T_1, \ldots, T_n$ die zugehörigen Sorten. Sei $e{:}T$ eine neue Variable, die nicht frei in $\lceil \mathtt{s} \rceil$ vorkommt.

Das neue Symbol besitzt somit die Signatur

$$asBag_E{:}T_1 \times \ldots \times T_n \to Bag_T$$

und die Übersetzung des Ausdrucks E entspricht gerade dem Term

$$\lceil \mathtt{E} \rceil = asBag_E(p_1, \ldots, p_n)$$

Zur Festlegung der Interpretation des neuen Symbols $asBag_E$ benutzten wir zudem in Ax_E:

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \dot{\forall} e{:}T\ ((e \in \lceil \mathtt{s} \rceil) \leftrightarrow count(asBag_E(p_1, \ldots, p_n), e) \doteq 1)$$
$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \dot{\forall} e{:}T\ (count(asBag_E(p_1, \ldots, p_n), e) \leq 1)$$

Für die Behandlung des Operators `asSequence` verfahren wir genauso; wir verwenden in den obigen Formeln lediglich ein neues Funktionssymbol $asSequence_E$ mit der Ergebnissorte $Sequence_T$ anstelle von $asBag_E$.

Man beachte insbesondere, daß wir *keine* Aussage über die Ordnung in der entstehenden Sequenz machen. Das Ergebnis der Übersetzung erscheint an dieser Stelle unterspezifiziert, entspricht aber genau der Spezifikation von OCL, d.h. OCL läßt an dieser Stelle offen, *welche* Sequenz genau als Ergebnis der Operator-Anwendung entsteht, da die Reihenfolge der Ergebnissequenz in der OCL-Spezifikation *nicht* festgelegt wird!

Beispiel 26
Wollen wir eine Sequenz aller Konten eines Kunden `c` berechnen, dann könnten wir den folgenden OCL-Ausdruck im Kontext $Customer$ verwenden:

$$E = \mathtt{c.accounts->asSequence}.$$

Die Übersetzung erzeugt aus diesem Ausdruck schließlich:

$$\lceil \mathtt{E} \rceil = asSequence_E(c)$$

und als zusätzlich Axiome in Ax_E

$$\dot{\forall} c{:}Customer \dot{\forall} a{:}Account\ (\ (a \in c.accounts) \leftrightarrow count(asSequence_E(c), a) \doteq 1)$$
$$\dot{\forall} c{:}Customer \dot{\forall} a{:}Accounts\ (count(asSequence_E(c), a) \leq 1)$$

2.3.3.9 Kollektionstypen – Bag

Seien im folgenden `b`, `b1`, `b2` OCL-Ausdruck für beliebige Multimengen, `c` ein OCL-Ausruck für eine Menge oder Multimenge und `o` ein Ausdruck des Basistyps `T`.

- **Gleichheit von Multimengen.** Wir behandeln die Gleichheit zweier Multimengen `b1`, `b2` durch eine Formel, die besagt, daß die beiden Mengen genau die gleichen Elemente mit der gleichen Anzahl von Vorkommen haben

  Sei `T` der kleinste gemeinsame Obertyp der Elementtypen `T1` und `T2` der beiden Multimengen. Sei $e{:}T$ eine neue Variable, die nicht frei in $\lceil$`b1`$\rceil$ oder $\lceil$`b2`$\rceil$ vorkommt.

  Dann lautet die gewünschte Formel

  $$\lceil \texttt{b1 = b2} \rceil = \dot{\forall} e{:}T\, (count(\lceil \texttt{b1} \rceil, e) \doteq count(\lceil \texttt{b2} \rceil, e))$$

  Analog übersetzen wir die Ungleichheit wieder durch die Negation dieser Formel.

- `union`, `intersection`, `including`, `excluding`. Zur Übersetzung aller dieser Eigenschaften nutzten wir wieder die korrespondierenden Funktionssymbole in Σ^*.

  Wir bilden also wie folgt ab:

  $$\lceil \texttt{b->union(c)} \rceil = union(\lceil \texttt{b} \rceil, \lceil \texttt{c} \rceil)$$
  $$\lceil \texttt{s->intersection(c)} \rceil = intersection(\lceil \texttt{s} \rceil, \lceil \texttt{c} \rceil)$$
  $$\lceil \texttt{s->including(o)} \rceil = insert(\lceil \texttt{c} \rceil, \lceil \texttt{o} \rceil)$$
  $$\lceil \texttt{s->excluding(o)} \rceil = remove(\lceil \texttt{c} \rceil, \lceil \texttt{o} \rceil)$$

- `asSet`,`asSequence`. Der Ausdruck $E =$ `b->asSequence` erzeugt eine Sequenz E, die genau die Elemente der Multimenge `b` enthält und zwar mit jeweils der gleichen Anzahl von Vorkommen.

  Wir formalisieren wieder unter Anwendung unserer Benennungstechnik wie folgt: Wir generieren ein neues Funktionssymbol $asSequence_E$ mit der Ergebnissorte $Sequence_T$ in Σ^*.

  Seien $p_1, \ldots, p_n$ wieder genau die freien Variablen aus der Übersetzung $\lceil$`b`$\rceil$ von `b`.Sei $e{:}T$ eine neue Variable, die nicht frei in $\lceil$`b`$\rceil$ vorkommt.

  Das neue Symbol besitzt somit die Signatur

  $$asSequence_E{:}T_1 \times \ldots \times T_n \rightarrow Sequence_T$$

  und die Übersetzung des Ausdrucks E entspricht gerade dem Term

  $$\lceil \texttt{E} \rceil = asSequence_E(p_1, \ldots, p_n)$$

  Zur Festlegung der Interpretation des neuen Symbols $asSequence_E$ benutzten wir zudem in Ax_E:

  $$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \dot{\forall} e{:}T\, (count(\lceil \texttt{b} \rceil, e) \doteq count(asSequence_E(p_1, \ldots, p_n), e))$$

  Man beachte, daß OCL (und damit unsere Abbildung) die genaue Reihenfolge *nicht* festlegt und damit die entstehende Ergebnissequenz nicht eindeutig spezifiziert ist.

Für die Behandlung des Operators `asSet` erzeugen wir stattdessen ein neues Funktionssymbol $asSet_E{:}T_1 \times \ldots \times T_n \rightarrow Set_T$ in Σ^* und definieren als Übersetzung von $E =$ `b->asSet` den Term

$$\lceil \texttt{E} \rceil = asSet_E(p_1, \ldots, p_n)$$

Die Menge der Axiome Ax_E enthält dann zusätzlich

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \dot{\forall} e{:}T\,(e \in \lceil \texttt{b} \rceil \leftrightarrow e \in asSet_E(p_1, \ldots, p_n))$$

Beispiel 27
Wollen wir die Menge aller Geburtsdaten der Angestellten des Unternehmens `cmp` berechnen, dann könnten wir den folgenden OCL-Ausdruck im Kontext *Company* verwenden:
$E =$ `cmp.employees->collect(e|e.birthDate)->asSet`.

Die Übersetzung erzeugt aus diesem Ausdruck schließlich:

$$\lceil \texttt{E} \rceil = asSet_E(cmp)$$

und als zusätzlich Axiome in Ax_E

$$\begin{array}{l}
\dot{\forall} cmp{:}Company \dot{\forall} d{:}Date\,(d \in collect_{E'}(cmp.employees) \leftrightarrow \\
\qquad\qquad d \in asSet_E(cmp)) \\
collect_{E'}(emptySet_{Person}) \doteq emptyBag_{Date} \\
\dot{\forall} s{:}Set_{Person} \dot{\forall} p{:}Person\,(\\
\quad collect_{E'}(insert(s,p)) \doteq insert(collect_{E'}(remove(s,p)), p.birthDate))
\end{array}$$

2.3.3.10 Kollektionstypen – Sequence

Seien im folgenden `s`, `s1`, `s2` OCL-Ausdruck für beliebige Sequenzen, `i` und `j` OCL-Ausdrücke des Typs `Integer` und `o` ein Ausdruck des Basistyps `T`.

- **Gleichheit von Sequenzen.** Wir behandeln die Gleichheit zweier Sequenzen `s1`, `s2` durch Verwendung des Gleichheitsymbols $\doteq$, d.h. wir erzeugen die Formel

  $$\lceil \texttt{s1 = s2} \rceil = \lceil \texttt{s1} \rceil \doteq \lceil \texttt{s2} \rceil$$

  Analog übersetzen wir die Ungleichheit wieder durch die Negation dieser Formel.

- `union`, `append`, `prepend`, `subSequence`, `first`, `last`, `including`, `excluding`.

  Für die Darstellung aller dieser Eigenschaften in der **DL** nutzten wir wieder die korrespondierenden Funktionssymbole in Σ^*.

  Wir bilden also wie folgt ab:

  $$\begin{array}{l}
  \lceil \texttt{s1->union(s2)} \rceil = union(\lceil \texttt{s1} \rceil, \lceil \texttt{s2} \rceil) \\
  \lceil \texttt{s->including(o)} \rceil = append(\lceil \texttt{s} \rceil, \lceil \texttt{o} \rceil) \\
  \lceil \texttt{s->excluding(o)} \rceil = remove(\lceil \texttt{s} \rceil, \lceil \texttt{o} \rceil) \\
  \lceil \texttt{s->append(o)} \rceil = append(\lceil \texttt{s} \rceil, \lceil \texttt{o} \rceil) \\
  \lceil \texttt{s->prepend(o)} \rceil = prepend(\lceil \texttt{s} \rceil, \lceil \texttt{o} \rceil) \\
  \lceil \texttt{s->first} \rceil = first(\lceil \texttt{s} \rceil) \\
  \lceil \texttt{s->last} \rceil = last(\lceil \texttt{s} \rceil) \\
  \lceil \texttt{s->subSequence(i,j)} \rceil = subSequence(\lceil \texttt{s} \rceil, \lceil \texttt{i} \rceil, \lceil \texttt{j} \rceil)
  \end{array}$$

Man beachte insbesondere das in OCL für Sequenzen die Operatoren `including` und `append` in ihrer Semantik übereinstimmen.

- `asSet`,`asBag`. Die Behandlung des Operators `asSet` für Sequenzen geschieht auf exakt dieselbe Weise wie für Multimengen (siehe Abschnitt 2.3.3.9). Ebenso verfahren wir bei der Übersetzung des Operators `asBag` für Sequenzen auf genau die gleiche Weise wie der Abbildung des Operators `asSequence` für Multimengen, die ebenfalls in Abschnitt 2.3.3.9 erläutert wurde.

2.3.3.11 Sonstige OCL-Konstrukte

`let`-Konstrukt. Bei der Formulierung komplexerer Constraints trifft man häufiger auf die Situation, daß ein bestimmter Teilausdruck `e1` mehrfach benutzt wird. OCL bietet für diese Fälle die Möglichkeit an, eine Abkürzung in Form einer *lokalen* Variablendefinition zu verwenden, um den (möglicherweise recht langen) Teilsausdruck `e1` nur einmal aufschreiben zu müssen. Der Gesamtconstraint wird durch die Einführung der Abkürzung außerdem übersichtlicher und lesbarer.

Eine solche lokale Variablendefinition wird über das `let`-Konstrukt formuliert: Seien `e1` bzw. `e2` OCL-Ausdrücke des Typs `T1` bzw. `T2` und der Typ `T` ein Obertyp von `T1`.

Dann definiert der Ausdruck E = `let v:T = e1 in e2` eine *lokale* Variable `v`, die mit der Auswertung von `e1` initialisiert ist. Durch die Lokalität der eingeführten Variablen ist diese nur für den Teilausdruck `e2` sichtbar bzw. gültig.

Bei der Auswertung von `e2` benutzt man nun für alle Vorkommen der Abkürzung `v` den Wert von `e1`.

Die Übersetzung des `let`-Konstrukts ist damit prinzipiell sehr einfach: Man ersetzt einfach alle Vorkommen von `v` – beispielsweise in einem weiteren Normalisierungsschritt vor der eigentlichen Übersetzung – syntaktisch durch `e1` und übersetzt anschließend den entstehenden Ausdruck `e2{v/e1}`. Damit nutzen wir einen der oben erwähnten Vorteile der Verwendung von `let` auf OCL-Ebene auf der Logikebene nicht aus: In der Übersetzung $\lceil$`e2{v/e1}`$\rceil$ des expandierten Ausdrucks tritt der auf der OCL-Ebene abgekürzte Ausdruck `e1` in Form seiner Übersetzung $\lceil$`e1`$\rceil$ mehrfach auf. Da durch die Verwendung einer Abkürzung `v` für den Ausdruck `e1` anzunehmen ist, daß `e1` in aller Regel etwas komplexer[43] sein wird, wird auch die zugehörige Übersetzung $\lceil$`e1`$\rceil$ ähnlich komplex ausfallen, d.h. der entstehende Term bzw. Formel aus der Abbildung von `e2{v/e1}` wäre unnötig kompliziert und würde für einen Menschen unter Umständen syntaktisch wenig Ähnlichkeit mit dem ursprünglich zu übersetzenden OCL-Ausdruck haben.

Um dem vorzubeugen, wollen wir einen anderen Weg gehen und dieselbe Abkürzungstechnik auch auf der Ebene der Logik anwenden:

Seien $p_1, \ldots, p_n$ genau die freien Variablen aus der Übersetzung $\lceil$`e1`$\rceil$ des abzukürzenden Ausdrucks `e1` und $T_1, \ldots, T_n$ die zugehörigen Sorten.

Wir führen in die Signatur Σ^* nun wieder ein neues n-stelliges Funktionssymbol $v{:}T_1 \times \ldots \times T_n \to T$ ein, welches prinzipiell der OCL-Variablen `v` entspricht bzw. diese Variable auf der Logikebene wiederspiegelt.

[43] Sonst würde sich die Verwendung einer Abkürzung *nicht* lohnen!

Dann verwenden wir als Übersetzung des Ausdrucks E dem Term

$$\lceil \texttt{E} \rceil = \lceil \texttt{e2} \rceil \{v/v(p_1, \ldots, p_n)\}$$

Zur Festlegung der Interpretation des neuen Funktionssymbols v benutzten wir zudem in Ax_E:

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, (v(p_1, \ldots, p_n) \doteq \lceil \texttt{e1} \rceil)$$

Beispiel 28
Wir wollen modellieren. daß alle verheirateten Frauen, die bei einer Bank `bnk` beschäftigt sind, auch als besondere Kunden dieses Unternehmens angesehen werden, und es mindestens eine solche Person gibt, der ein Konto mit einem positiven Kontostand bei dieser Bank zugeordnet ist. Dazu formulieren wir den OCL-Ausdruck $E =$

```
let marriedFem =
  bnk.employees->select(e| e.sex = 'F' and e.married) in

 marriedFem->forAll(f| bnk.specials->includes(f)) and
  marriedFem->exists(f|
    f.accounts->exists(a| a.accountBalance>0 and a.bank=bnk))
```

Als Resultat der Abbildung erhalten wir für E

$$\begin{aligned}\lceil \texttt{E} \rceil = \dot{\forall} f{:}Person \, (f \in married Fem(b) \rightarrow (f \in bnk.specials)) \wedge \\ \dot{\exists} f{:}Person \, (f \in married Fem(b) \wedge \\ \dot{\exists} a{:}Account \, (a \in f.accounts \wedge \\ a.accountBalance > 0 \wedge \\ a.bank \doteq bnk))\end{aligned}$$

sowie folgende Axiome in Ax_E:

$$\dot{\forall} b{:}Bank \, (married Fem(b) \doteq select_{E'}(b.employees))$$

$$select_{E'}(emptySet_{Person}) \doteq emptySet_{Person}$$

$$\begin{aligned}&\forall s{:}Set_{Person} \dot{\forall} p{:}Person \, (\\ &\quad (p.sex \doteq \lceil \texttt{F} \rceil \wedge p.married \doteq true) \rightarrow \\ &\quad select_{E'}(insert(s,p)) \doteq insert(select_{E'}(s), p))\end{aligned}$$

$$\begin{aligned}&\forall s{:}Set_{Person} \dot{\forall} p{:}Person \, (\\ &\quad \neg(p.sex \doteq \lceil \texttt{F} \rceil \wedge p.married \doteq true) \rightarrow \\ &\quad select_{E'}(insert(s,p)) \doteq select_{E'}(s))\end{aligned}$$

Bemerkung (`let`-Konstrukt in OCL Version 1.4). In der Version 1.4 der Spezifikation von OCL wurde das `let`-Konstrukt auf sinnvolle Art und Weise etwas verallgemeinert: Es ist nun nicht nur eine lokale *Variablendefinition* erlaubt, der Modellierer kann vielmehr eine lokale Definition von *Funktionen* – die im allgemeinen Parameter enthalten – vornehmen. Unsere Übersetzung paßt nahtlos zu dieser Neuerung, denn eine Funktionsdefinition mittels `let` kann auf exakt die gleiche Art wie eine lokale Variablendefinition behandelt werden!

Eine weitere kleine Neuerung hat sich für lokale Definitionen in der neuen Version von OCL ergeben: Lokal definierte Größen – also Variablen oder Funktionen – werden implizit als (virtuelle) Eigenschaften, d.h. Attribute bzw. Methoden, der entsprechenden Kontextklasse betrachtet und dementsprechend notiert. Diese neue Betrachtung

von lokal definierten Größen hat nur insofern eine Auswirkung auf unser Vorgehen, als das wir ein entsprechendes, neu eingeführtes (nicht-rigides) Funktionssymbol wie ein Attribut bzw. eine Methode der zum Kontextelement gehörenden Kontextklasse in der **DL** notieren, um einen Term zu erhalten, der syntaktisch stark an den ursprünglichen OCL-Ausdruck erinnert. Man beachte, daß es sich lediglich um eine andere Notation eines nicht-ridigen Funktionssymbols in der **DL** – also um *kein* neues Konzept – handelt. □

2.3.3.12 Besondere Konstrukte für OCL-Nachbedingungen.

Nachbedingungen nehmen insofern eine gewisse Sonderrolle in OCL ein, als daß OCL drei verschiedene Features bzw. Konstrukte kennt, die nur in OCL-Ausdrücken für die Beschreibung von Nachbedingungen erlaubt sind: Das `@pre`-Konstrukt, das Literal `result`, sowie `oclIsNew`. Wir werden die Übersetzung dieser Konstrukte in den folgenden Abschnitte diskutieren.

`@pre`-Konstrukt. Bei der Formulierung einer Spezifikation einer Methode ist es ungemein wichtig, auch die Werte eines Attributs (eines Objektes) *vor* Methodenausführung verwenden zu können. Allgemeiner wünscht man sich, eine Eigenschaft eines OCL-Typs im Zustand vor Ausführung der Methode auswerten zu können. Daher stellt OCL ein syntaktisches Konstrukt zur Verfügung, um eine solche Auswertung im Vorzustand zu notieren: Der `@pre`-Operator.

Beschreibe `o` eine Instanz des OCL-Typs T und `feature` eine für diesen Typen definierte Eigenschaft.

Dann bezeichnet der OCL-Ausdruck `o.feature@pre` eine Instanz, die durch Auswertung der Eigenschaft `feature` bezüglich der Instanz `o` *vor* Methodenausführung entsteht.

Für die Behandlung dieses Operators aus OCL in unserer Übersetzung haben wir uns für die folgende Vorgehensweise entschieden: Wir führen für die Abbildung einer mit `@pre` gekennzeichneten Eigenschaft `feature` zunächst ein (nicht-rigides) Funktionssymbol *feature*@*pre* in Σ^* ein, das wir direkt in der Übersetzung heranziehen, also

$$\lceil \texttt{o.feature@pre} \rceil = \mathit{feature@pre}(\lceil \texttt{o} \rceil)$$

Die Intention des neuen Symbols entspricht genau der von OCL, d.h. die zugehörige Funktion liefert den Wert der Auswertung der betrachteten Eigenschaft bzgl. des gegebenen Objekts im Vorzustand zurück.

Diese Semantik des Funktionssymbols *feature*@*pre* wird aber durch die oben angegebene Abbildung noch nicht eingefangen. Doch irgendwie müssen wir dieser Intention formal gerecht werden!

Da die Behandlung von `@pre` auf viele unterschiedliche Weisen denkbar ist – man betrachte beispielsweise [BBS01] – , eine solche Behandlung im allgemeinen sogar *anwendungsabhängig*[44] sein kann und eigentlich unabhängig von der Behandlung der OCL-Eigenschaften während der Übersetzung an sich ist, haben wir uns entschieden, zunächst bei der Abbildung ein nicht-rigides Funktionssymbol in der Logik zu verwenden.

Die Semantik dieses Funktionssymbols an sich wird schließlich in einem separaten Bereinigungslauf über der im ersten Durchlauf der Übersetzung erzeugten Formel

[44] im Sinne der Verwendung der generierten Formeln

eingefangen; entsprechende neue Formeln stellen sicher, daß das eingeführte Funktionssymbol in der gewünschten Weise interpretiert wird.

Dieses Verfahren hat zudem den Vorteil, allgemein genug zu sein, um auf beliebige Logiksprachen übertragbar zu sein. Die Art und Weise, wie man in einer konkreten Logiksprache (wie der **DL**) über die Gültigkeit einer Formel in einem bestimmten Zustands spricht, kann sehr unterschiedlich sein und die Behandlung von `@pre` ist dem entsprechend sogar Zielsprachen abhängig.

Wir werden in dieser Arbeit auf diese semantische Behandlung des eingeführten Funktionssymbols nicht weiter eingehen, um den Rahmen dieser Arbeit nicht zu sprengen. Der interessierte Leser sei auf die Arbeit [BBS01] verwiesen, in der zwei verschiedene Methoden der Behandlung vorgestellt werden, die jeweils unterschiedliche Eigenschaften mit Hinblick auf die deduktive Verarbeitung der transformierten und nachbehandelten Formeln in einem logischen Kalkül aufweisen.

Bemerkung (Attribute und Methoden). Falls die oben betrachtete Eigenschaft durch ein Attribut oder eine Methode in einer Kontextklasse der **DL** verkörpert wird, so notieren wir das Funktionssymbol *feature*@*pre* in einem Funktionsterm entsprechend der Attribut- bzw. Methodenschreibweise, d.h. anstelle von *feature*@*pre*(⌈`o`⌉) notieren wir ⌈`o`⌉.*feature*@*pre*. Man beachte, daß es sich lediglich um eine andere Notation eines (nicht-rigiden) Funktionssymbols handelt, die sich syntaktisch näher an OCL orientiert und somit hoffentlich die Lesbarkeit der entstehenden Formeln für den Modellierer erhöht. □

`result`-Literal. Die Übersetzung des Literals `result`, dessen Typ dem Rückgabetyp `T` der betrachteten Methode m der Kontextklasse C entspricht, geschieht durch eine gleichnamige Konstante $result{:}T$, also

$$\lceil \texttt{result} \rceil = result$$

In der **DL** verwenden wir stattdessen aus technischen Gründen eine *Programmvariable*, d.h. eine Variable, die sich wie eine modale Konstante verhält.

Die formale Charakterisierung der eigentlichen Bedeutung dieses Literals wird erst später bei der Übersetzung von Nachbedingungen behandelt. Notwendigerweise ist auch diese Formalisierung abhängig von der verwendeten Zielsprache.

`oclIsNew`. Die Anwendung von `oclIsNew` auf einen OCL-Ausdruck `o` eines Basistypen erlaubt es einem Modellierer zu testen, ob die durch den Ausdruck `o` beschriebene Instanz erst während der Ausführung der betrachteten Methode m entstanden ist, also *vor* deren Ausführung *nicht* existierte, danach aber schon.

In der **DL** gibt es für Kontextklassen ein geeignetes boolesches Attribut `created`, welches angibt, ob ein betrachtetes Objekt der Klasse (im betrachteten Systemzustand) existiert[45].

Für Modelltypen `C` nutzen wir dieses Attribut bei der Übersetzung von `oclIsNew` in Kombination mit dem `@pre`-Operator:

[45] Die **DL** betrachtet also nur Modelle mit *konstanten* Universen, d.h. insbesondere alle *möglichen* Objekte sind Teil des Universums, wobei die für objektorientierte Systeme charakteristische Eigenschaft der dynamischen Extensionen der Klassen, die Modellen mit dynamischen Universen nähersteht, durch das zusätzliche boolesche Attribut `created` für Kontextklassen nachempfunden wird.

Betrachtet man eine beliebige Instanz `o` einer Klasse C im Snapshot D, so läßt sich der OCL-Ausdruck `o.oclIsNew` für die zugehörige Kontextklasse[46] `C` *äquivalent* in folgenden Ausdruck umschreiben:

$$\texttt{o.oclIsNew} = \texttt{o.created and (not o.created@pre)}$$

Wir nutzen diese Umformulierung für die Abbildung und erhalten:

$$\lceil \texttt{o.oclIsNew} \rceil = (o.created \doteq \texttt{true}) \wedge (o.created@pre \doteq \texttt{false})$$

Beispiel 29
In einer Nachbedingung für die Methode *startAccount* der Klasse *Person* könnte man fordern, daß (nach Ausführung dieser Methode) ein neues Objekt der Klasse *Account* erzeugt wurde, welches das neue Konto repräsentiert, sofern die Methode den booleschen Wert `true` zurückgibt. Also beispielsweise (im Kontext der Klasse *Person*):
$E =$ `result = true implies self.accounts->exists(a|a.oclIsNew)`.

Nach der Übersetzung diese Ausdrucks erhalten wir:

$$\begin{aligned}\lceil \texttt{E} \rceil = {} & (result \doteq true) \rightarrow \\ & \dot{\exists} a{:}Account\,(a \in self.accounts \wedge \\ & \qquad ((a.created \doteq \texttt{true}) \wedge (a.created@pre \doteq \texttt{false})))\end{aligned}$$

Für alle anderen Basistypen, also Boolean, String, Integer und Real gibt es in der **DL** kein entsprechendes Attribut, da wir ADTs anstelle von Kontextklassen zu deren Repräsentation in der Logik nutzen. Die Anwendung des Operators `oclIsNew` erscheint aber für diese Typen nicht unbedingt sinnvoll: Was bedeutet es für einen booleschen Wert oder ein natürliche Zahl, während der Methodenausführung erzeugt worden zu sein[47]? Man beachte an dieser Stelle, daß OCL selbst und damit unsere Formalisierung in Form von ADTs diese Basistypen als *mathematische* Konzepte und *nicht* als *programmiersprachliche* Konzepte auffaßt! Als Konsequenz daraus betrachten wir es als unmöglich, von der Existenz bzw. Nichtexistenz einer Instanz eines solchen Typs in einem bestimmten Systemzustand zu sprechen. Wir erlauben daher die Anwendung von `oclIsNew` nur für Modelltypen!

Bemerkung (Vorgehen für beliebige Zielsprachen). Betrachtet man ein Attribut einer Kontextklasse einfach als ein *nichtrigides* Funktionssymbol, dann läßt sich das oben angegebene Verfahren unter Verwendung eine solchen Funktionssymbols (anstelle eines Attributs) in exakt der gleichen Weise auch für andere Zielsprachen L verwenden. Für jede solche Zielsprache beleibt dann lediglich in einem auf die Übersetzung folgende Bereinigungschritt festzulegen bzw. zu detaillieren, wie in der Zielsprache über die Existenz bzw. Nichtexistenz von Objekten gesprochen wird. Im einfachsten Fall reicht es möglichereise aus ein Axiom einzuführen, welches die Funktion *created* definiert! □

[46] Man beachte, daß eine Kontextklasse `C` einer Klasse C aus dem UML-Modell $\mathcal{D}$ eine Erweiterung dieser Klasse darstellt. Insbesondere handelt es sich bei `C` um eine herkömmliche Klasse in der **DL**.

[47] Eine ähnliche Frage findet sich in der OCL-Spezifikation [Obj99a, Seite 6-66] für den Operator `allInstances`.

2.3.3.13 Neue Konstrukte in OCL Version 1.4.

any-Konstrukt. Um aus einer Collection `c` (mit Elementtyp `T`) ein *beliebiges* Element `e` auszuwählen, welches einer bestimmten Eigenschaft `b` genügt, gibt es in OCL nun einen Operator `any`, der auf jede Kollektion angewendet werden darf. Beinhaltet die betrachtete Kollektion *kein* solches Element, so ist der gesamte `any`-Ausdruck *undefiniert*.

Für die Übersetzung des Ausdrucks E = `c->any(e|b)` benutzten wir abermals unsere Benennungstechnik:

Seien $p_1, \ldots, p_n$ die freien Variablen aus den Übersetzungen $\lceil$`c`$\rceil$ und $\lceil$`b`$\rceil$ der direkten Subausdrücke in E und $T_1, \ldots, T_n$ die zugehörigen Sorten. Seien $e', e'':T$ neue Variable, die nicht frei in $\lceil$`c`$\rceil$ oder $\lceil$`b`$\rceil$ vorkommen.

Dann definieren wir – unter Verwendung eines neuen Funktionssymbols any_E in $\Sigma_{\mathcal{D}}$ mit mit den Argumentsorten $T_1, \ldots, T_n$ und der Ergebnissorte T – als Übersetzung von E den Term

$$\lceil \texttt{E} \rceil = any_E(p_1, \ldots, p_n)$$

und fügen in die Axiomenmenge Ax_E zusätzlich die folgende Formel ein:

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, (\dot{\exists} e'{:}T \, (e' \in \lceil \texttt{c} \rceil \wedge \lceil \texttt{b} \rceil \{e/e'\}) \\ \rightarrow \dot{\exists} e''{:}T \, (e'' \doteq any_E(p_1, \ldots, p_n) \wedge e'' \in \lceil \texttt{c} \rceil \wedge b\{e/e''\}))$$

Man beachte, daß der Teilterm $e'' \doteq any_E(p_1, \ldots, p_n)$ in Verbindung mit dem punktierten Existenzquantor sichert, daß der Funktionswert einem *existierenden* Objekt entspricht!

Bemerkung (Undefiniertheit). Wie oben angemerkt kann es geschehen, daß die betrachtete Kollektion `c` kein Element `e` enthält, welches der Eigenschaft `b` genügt. In diesem Falle ist der OCL-Ausdruck *undefiniert*. Wir modellieren diese Situation in der obigen Formel durch eine Unterspezifikation des Funktionswertes von any_E im Falle der Undefiniertheit in OCL. Verwendet eine Anwendung eine spezielle Methode zur Auflösung von undefinierten Werten in der Logik, so muß sie an dieser Stelle eventuell geeignete Anpassungen vornehmen. □

Bemerkung (Darstellung über ϵ-Terme). Der `any`-Operator hat sehr viel Ähnlichkeit zu sogenannten ϵ-Termen, die beispielsweise in [DH39] oder in [Gie00] behandelt werden. Falls die betrachtete Logiksprache solche Terme unterstützt, so könnte man zur Übersetzung des Ausdrucks E auch direkter einen ϵ-Term benutzen:

$$\lceil \texttt{E} \rceil = \epsilon e'{:}T \, (e'.created \doteq \texttt{true} \wedge \neg(e' \doteq null) \wedge e' \in \lceil \texttt{c} \rceil \wedge \lceil \texttt{b} \rceil \{e/e'\})$$

wobei $e'{:}T$ wieder eine neue Variable sei, die nicht frei in $\lceil$`c`$\rceil$ oder $\lceil$`b`$\rceil$ vorkommt.

Man beachte, daß im Falle der Undefiniertheit des ursprünglichen OCL-Ausdrucks E – wie bei der vorher angegebenen Abbildung auch – ein beliebiges Element e der Sorte T zurückgegeben wird. Man muß also auch bei dieser Variante an der entsprechenden Stelle in der Methode zur Auflösung der undefinierten Werte in der entsprechenden Logik Sorgfalt walten lassen, da – im Gegensatz zu der axiomatischen Methoden von oben – der undefinierte Fall hier in *keiner Weise* beachtet wird. Eine mögliche Lösung wäre eine indirekte Beschreibung des Ergebniswerts durch die Anwendung der Bennungstechnik, wobei – wie im obigen Fall auch – ein Axiom eingeführt

wird, welches den Ergebniswert des Funktionswert nur dann spezifiziert, wenn der ursprüngliche OCL-Ausdruck definiert ist. □

one-Konstrukt. Der Ausdruck E = `c->one(e|b)` ist in OCL genau dann erfüllt, wenn es exakt ein Element `e` in der Kollektion `c` (mit Basistyp T) gibt, welches die Bedingung `b` erfüllt.

Seien $e', e''{:}T$ neue Variable, die nicht frei in $\lceil \mathtt{c} \rceil$ oder $\lceil \mathtt{b} \rceil$ vorkommen.

Wir gehen dann wie folgt vor:

$$\begin{aligned} \lceil \mathtt{E} \rceil \;\; = \;\; & \dot{\exists} e'{:}T\,(e' \in \lceil \mathtt{c} \rceil \wedge \lceil \mathtt{b} \rceil \{e/e'\} \wedge \\ & \quad \dot{\forall} e''{:}T\,((e'' \in \lceil \mathtt{c} \rceil \wedge \lceil \mathtt{b} \rceil \{e/e''\}) \;\rightarrow\; e' \doteq e'')) \end{aligned}$$

2.3.4 Abbildung von OCL-Constraints in die DL

Wir erweitern in diesem Abschnitt die Abbildung von OCL-*Ausdrücken* aus dem Abschnitt 2.3.3 zu einer Übersetzung für OCL-*Constraints*, d.h. Invarianten bzw. Vor- und Nachbedingungen.

2.3.4.1 Abbildung von OCL-Invarianten

Invarianten I werden in OCL durch Constraints der Form

```
context [c:]C
  inv: b
```

formuliert, wobei `C` ein Klassenbezeichner aus dem UML-Modell $\mathcal{D}$ und `b` ein boolescher OCL-Ausdruck ist. Optional kann explizit ein Bezeichner `c` für das Kontextelement angegeben werden, welches standardmäßig durch `self` verkörpert wird.

Die Semantik einer solchen Invariante besagt nun, daß für ein *beliebiges* Kontextelement *self* (bzw. c) des Typs `C`, welches in einem betrachteten Systemzustand D des UML-Modells $\mathcal{D}$ existiert, die Bedingung `b` erfüllt ist.

Entsprechend formulieren wir als Übersetzung Th_I der Invariante I

$$\bigwedge Ax_{\mathtt{b}} \rightarrow \dot{\forall} \mathit{self}{:}C\,\lceil \mathtt{b} \rceil$$

Falls das Kontextelement durch `c` bezeichnet wird, so wird in der obigen Formel die Quantorvariable $\mathit{self}{:}C$ durch die Quantorvariable $c{:}C$ ersetzt.

Beispiel 30
Wir wollen fordern, daß für jedes Unternehmen gilt, daß zu jedem Zeitpunkt die Anzahl der Mitarbeiterinnen mindestens so groß ist, wie Anzahl der Mitarbeiter. Dazu verwenden wir den Constraint C

```
context Company
  inv: self.employees->select(e|e.sex = 'm')->size
       <=  self.employees->select(e|e.sex = 'f')->size
```

Als Übersetzung Th_C erhalten wir nunmehr

$$
\begin{aligned}
&((select_E(emptySet_{Person}) \doteq emptySet_{Person}) \wedge \\
&(\forall s{:}Set_{Person}\dot{\forall} e{:}Person\,(\\
&\quad (e.sex \doteq \texttt{'m'}) \rightarrow select_E(insert(s,e)) \doteq insert(select_E(s),e))) \wedge \\
&(\forall s{:}Set_{Person}\dot{\forall} e{:}Person\,(\\
&\quad \neg(e.sex \doteq \texttt{'m'}) \rightarrow select_E(insert(s,e)) \doteq select_E(s))) \wedge \\
&(select_{E'}(emptySet_{Person}) \doteq emptySet_{Person}) \wedge \\
&(\forall s{:}Set_{Person}\dot{\forall} e{:}Person\,(\\
&\quad (e.sex \doteq \texttt{'f'}) \rightarrow select_{E'}(insert(s,e)) \doteq insert(select_{E'}(s),e))) \wedge \\
&(\forall s{:}Set_{Person}\dot{\forall} e{:}Person\,(\\
&\quad \neg(e.sex \doteq \texttt{'f'}) \rightarrow select_{E'}(insert(s,e)) \doteq select_{E'}(s)))) \\
&\rightarrow size(select_E(self.employees)) \leq size(select_{E'}(self.employees))
\end{aligned}
$$

Bemerkung (Vereinfachung für die DL). Der angegebene Ausdruck ist für beliebige Logiksprachen geeignet. In der Zielsprache **DL** wollen wir jedoch zur Vereinfachung eine spezielle Form von Variablen für die freien Variablen im Constraint verwenden: *Programmvariable*, die sich wie modale Konstanten verhalten.

Da Programmvariablen in der **DL** als modale Konstanten angesehen werden können, benötigen wir die Quantifizierungen über diese Variablen nichtmehr. Außerdem spielen diese Variablen – die nun als Konstanten betrachtet werden!– keine Rolle mehr als Parameter für irgendwelche bei der Übersetzung erzeugten Funktionssymbole.

Ein entsprechender Bereinigungslauf über der oben angegebenen Formeln kann diese Vereinfachungen vornehmen und bereinigte Formeln generieren.

Man mache sich klar, daß das Ergebnis dann im Sinne von Abschnitt 2.2.3 korrekt ist. Der Grund dafür liegt in der *impliziten* Allquantifikation über die Welten w_D und somit aller möglichen Interpretationen dieser modalen Konstanten. □

Bemerkung (Semantik von Invarianten). Ein Aspekt der Semantik von Invarianten scheint uns in der OCL-Spezifikation nicht ganz klar herausgearbeitet zu sein: In welchen Zuständen bzw. Snapshots D des Systems $\mathcal{D}$ muß die obige Aussage zutreffen?

In der OCL-Spezifikation [Obj01][Abschnitt 6.3.3, Seite 6-52] heißt es hierzu

> An OCL expression is an invariant of the type and must be true for all instances of that type at *any* time.

Ob mit dieser Formulierung wirklich die Deutung „für alle Objekte, in allen Snapshots" gemeint ist, scheint nicht sicher, denn betrachtet man das gegebene System $\mathcal{D}$ in seiner *dynamischen* Entwicklung, so kann man sich vorstellen, daß das System durchaus „illegale" Zustände *während* einer Methodenausführung durchläuft, ohne das dabei ein Schaden entstünde.

Man findet diese Situation beispielsweise bei Datenbanksystemen, die einen Transaktionsmechanismus unterstützen. Dort ist gewährleistet, daß zu *bestimmten* Zeitpunkten, nämlich dem Start und dem Ende einer Transaktion, alle Konstistenzbedingungen erfüllt sind und während der Ausführung einer Transaktion die möglicherweise „illegalen" Zwischenzustände niemals für einen Benutzer des Systems sichtbar werden.

Man könnte also ein schwächere Bedingung für die Erfüllung von Invarianten verwenden: Die geforderte Eigenschaft muß nur dann für einen Snapshot D gelten, wenn

in diesem Snapshot *kein* Objekt der im Constraint verwendeten Klassen *aktiv* ist, d.h. gerade ein Methodenaufruf auf diesem Objekt durchgeführt wird oder auf ein Attribut dieses Objekt zugegriffen wird.

Ein solcher Ansatz findet sich beispielsweise in Darstellung einer Übersetzung von OCL nach BOTL in [DKR00b].

Man beachte, daß die obige Formel Th_I dadurch prinzipiell nicht verändert werden muß, da für die Korrektheit die Formel Th_I relativ zum Constraint I betrachtet wird, und die geschilderte Unklarheit eine Schwäche von UML/OCL darstellt.

Möchte man jedoch eine Formel erhalten, die bezgl. einer bestimmten Menge von Welten allgemeingültig ist, so muß gegebenenfalls die erzeugte Formel Th_I entsprechend den Ausdrucksmitteln der Zielsprache erweitert werden. □

2.3.4.2 Abbildung von OCL-Vor/Nachbedingungen

Ein OCL-Constraint C zur Beschreibung von Vor-/Nachbedingungen einer Methode `m` aus einer Klasse `C` hat die Gestalt

```
context [c:]C::m(p1:T1,...pN:TN):T0
  pre:  b1
  post: b2
```

wobei `C` eine Klasse aus dem UML-Modell $\mathcal{D}$ ist, `m` eine Methode dieser Klasse mit den Parametern p_k des Typs $T_k, k = 1, \ldots, n$ und dem Rückgabetyp T_0 verkörpert und `b1`, `b2` boolesche OCL-Ausdrücke darstellen.

Ein solcher Constraint C hat die folgende Bedeutung: Für *jede* Instanz e der Klasse `C` and *beliebige* Werte v_i der Parameter p_i im Zustand D_{pre} des Systems $\mathcal{D}$ gilt: Wenn die Aussage, die durch den Ausdruck `b1` verkörpert wird, in einem Zustand D_{pre} gilt, dann terminiert der Aufruf der Methode m mit den gegebenen Parameterwerten v_i auf dem Objekt e im[48] Zustand $D_{post} \equiv D_{post}(e, v_1, \ldots, v_n, D_{pre})$ und es ist anschließend `b2` im Nachzustand D_{post} erfüllt ist.

Bemerkung (Mehrere Vor- und Nachbedingungen). OCL erlaubt prinzipiell beliebig viele – also auch keine – Vor- und Nachbedingungen bei der Spezifikation einer Methode, was unter anderem die Lesbarkeit einer solchen Spezifikation verbessert. Wir können einen Constraint C solcher Art jedoch in einen *äquivalenten* Constraint C' wandeln, der genau eine Vor- und eine Nachbedingung enthält:

Die vorhandenen Vorbedingungen werden dabei in OCL einfach konjunktiv verknüpft bzw. falls keine Vorbedingung angegeben wurde, so verwenden wir `true`.

Für die Nachbedingungen verfahren wir genau auf die selbe Weise. □

Nun, wie können wir diese Bedeutung eines Constraints C in der **DL** formalisieren?

Seien $\lceil$`b1`$\rceil$ bzw. $\lceil$`b2`$\rceil$ die Terme bzw. Formeln, welche die Übersetzung der Ausdrücke für die Vor- bzw. Nachbedingung beschreiben, und $Ax_{\texttt{b1}}$ bzw. $Ax_{\texttt{b2}}$ die entsprechenden Axiomenmengen. Dann charakterisiert die Formel

$$\theta^{pre} = \bigwedge Ax_{\texttt{b1}} \wedge \lceil \texttt{b1} \rceil$$

[48] Wir betrachten hier lediglich *deterministische* Implementierungssprachen.

die Anforderung an den Vorzustand, die durch die Vorbedingung formuliert wurde. Gleichermaßen beschreibt die Formel

$$\theta^{post} = \bigwedge Ax_{\mathtt{b2}} \rightarrow \lceil \mathtt{b2} \rceil$$

die Anforderung an den Nachzustand, die durch die Nachbedingung formuliert wurde.

Nun brauchen wir noch eine geeignete Beschreibung des Nachzustands D_{post} in Abhängigkeit der angegebenen Größen, d.h $D_{post}(e, v_1, \ldots, v_n, D_{pre})$. Aber das ist gerade in der **DL** unter Benutzung eines Programms, welches den Methodenaufruf darstellt, und einem entsprechenden Diamond-Term , der die Terminierung der Methodenausführung ausdrückt, sehr elegant möglich: Als Übersetzung Th_C des Constraints C verwenden wir die Formel

$$\begin{array}{l}\dot{\forall} self{:}C \dot{\forall} v_1{:}T_1 \ldots \dot{\forall} v_n{:}T_n \, (\\ \quad (\bigwedge Ax_{\mathtt{b1}} \wedge \lceil \mathtt{b1} \rceil) \rightarrow \\ \quad \langle \mathtt{T0}\ result = self.m(v_1, \ldots, v_n) \rangle (\bigwedge Ax_{\mathtt{b2}} \rightarrow \lceil \mathtt{b2} \rceil))\end{array}$$

Man beachte,auf welch einfache Weise die Semantik der Programmvariablen *result*, welche in Abschnitt 2.3.3.12 eingeführt wurde, nun formal in dieser **DL**-Formel festgelegt wird: Die Zuweisung stellt sicher, daß der Wert dieser Programmvariablen genau dem Rückgabewert des Methode bei dem entsprechenden Methodenaufruf entspricht.

Beispiel 31
Wir wollen fordern, daß für jeden Kunden gilt, wenn für diese Person eine Konto bei einer Bank `aBank` eröffnet wird, dann gilt danach: Falls eine positive Bestätigung zurückgegeben wurde, dann gibt es ein neues Konto, welches zur angegebenen Bank und der betrachteten Person gehört und mit dem Kontostand 0 initialisiert ist. Zudem darf dann nur ein einziges Konto zu der Menge der Konten, die der betrachteten Person zugeordnet sind, hinzugefügt worden sein. Als Vorbedingung fordern wir außerdem, daß die Person volljährig ist. Wir verwenden deshalb als Constraint C

```
context c:Customer::startAccount(aBank:Bank):Boolean
  pre:  c.age >= 18
  post: result implies
          c.accounts->exists(a|a.oclIsNew
                                and a.accountBalance = 0
                                and a.bank = aBank)
  post: result implies
          c.accounts->size = c.accounts->size@pre + 1
```

Als Übersetzung Th_C erhalten wir nunmehr

$$\begin{array}{l}\dot{\forall} c{:}Customer \dot{\forall} aBank{:}Bank \, (\\ c.age \geq 18 \rightarrow \\ \\ \langle \mathtt{boolean}\ result = c.startAccount(aBank) \rangle \\ \quad ((result \doteq true) \rightarrow \\ \quad\quad \dot{\exists} a{:}Account \, (a \in c.accounts \wedge \\ \quad\quad\quad\quad (a.created \doteq true \wedge a.created@pre \doteq false) \wedge \\ \quad\quad\quad\quad a.accountBalance \doteq 0 \wedge \\ \quad\quad\quad\quad a.bank \doteq aBank)) \wedge \\ \quad ((result \doteq true) \rightarrow \\ \quad\quad size(c.accounts) = size@pre(c.accounts) + 1))\end{array}$$

Bemerkung (Statische Methoden und `void`-Methoden). Es gilt auch an dieser Stelle den verschiedenen Ausprägungen von Methoden gerecht zu werden: Obiges Verfahren muß für *statische* Methoden und Methoden ohne Rückgabewert angepaßt werden.

Für statische Methoden gibt es kein Kontextelement, da solche Methoden an die Klasse an sich und nicht an eine Instanz der Klasse gebunden sind; solche Methoden können insbesondere auch dann ausgeführt werden, wenn *keine* Instanz der entsprechenden Klassen im Snapshot D existiert!

Für diese Methodenart fällt das Kontextelement *self* aus der Formel Th_C heraus und der Methodenaufruf wird syntaktisch in geeigneter Weise angepaßt:

$$Th_C = \dot{\forall} v_1{:}T_1 \ldots \dot{\forall} v_n{:}T_n \, (\theta^{pre} \rightarrow \langle \texttt{T0}\ result = C.m(v_1, \ldots, v_n) \rangle \theta^{post})$$

Für Methoden ohne Rückgabewert, d.h. `T0` = `void` entfällt in der Übersetzung die Zuweisung, da man nur noch den eigentlichen Methodenaufruf benötigt. Man beachte, daß ein gültiger OCL-Ausdruck für die Nachbedingung in einem solchen Fall das Literal `result` *nicht* verwenden darf; damit kommt die Programmvariable *result* ebenfalls in der Formel θ^{post} nicht vor!

Wir erhalten also für die Abbildung dieser Art von Methode

$$Th_C = \dot{\forall} self{:}C \dot{\forall} v_1{:}T_1 \ldots \dot{\forall} v_n{:}T_n \, (\theta^{pre} \rightarrow \langle self.m(v_1, \ldots, v_n) \rangle \theta^{post})$$

Beide Varianten können auch kombiniert werden, die angegeben Abbildungen sind völlig unabhängig voneinander und können deshalb in der gleichen Weise einfach kombiniert werden. □

Bemerkung (Behandlung von `@pre`). Die generierte Übersetzung Th_C des Constraints C enthält möglicherweise noch die in Abschnitt 2.3.3.12 eingeführten Funktionssymbole $f@pre$, deren Intention dort zwar erklärt wurde, deren Semantik jedoch noch nicht formal gefaßt wurde.

Wir sind jetzt an einem Punkt angekommen, an dem wir die Semantik dieser Funktionssymbole, die ja auf Auswertungen bzgl. des Vorzustandes verweisen, formal beschreiben können: Wir werden die Formel Th_C in einem weiteren und von der Übersetzung prinzipiell unabhängigen Transformationsschritt geeignet bereinigen.

Wir möchten abermals kurz daran erinnern, daß die genaue Transformation *anwendungsabhängig* und damit für jede Anwendung einer solchen Abbildung geeignet definiert werden muß. Als Einflußfaktoren lassen sich beispielsweise die benutzte Logiksprache oder aber der Zweck, für den die generierten Formeln verwendet werden, identifizieren. Beispielsweise werden in [BBS01] für die gleiche Zielsprache zwei verschiedene Verfahren der Auflösung dieser speziellen Funktionssymbole vorgestellt, die auch unterschiedliche „logische" Eigenschaften haben. Welches Verfahren „das" geeignete ist, hängt vom Verwendungszweck der generierten Formeln ab.

Wir gehen an dieser Stelle nicht weiter ins Detail und möchten nur kurz erwähnen, daß das einfachere der beiden Verfahren in der Formel θ^{pre} für ein Funktionssymbol $f@pre$ schlicht eine Gleichung einfügt, die die besagt, daß die Auswertung des Funktionssymbols $f@pre$ genau der des Funktionssymbols f *im Vorzustand* der Methodenausführung entspricht. Genauer gesagt, wird Formel

$$\dot{\forall} v_1{:}T_1 \ldots \dot{\forall} v_n{:}T_n \, (f@pre(v_1, \ldots, v_n) \doteq f(v_1, \ldots, v_n))$$

(wobei $v_1, \ldots, v_n$ die Parameterwerte der entsprechenden Parameter mit dem Typen $T_1, \ldots, T_n$ des Funktionssymbols f sind) konjunktiv zu $\bigwedge Ax_{\mathtt{b1}}$ in θ^{pre} hinzugefügt.

Bemerkung (Übersetzung in beliebige Zielsprachen). Möchte man die angegebene Abbildung für andere Zielsprachen als die **DL** verwenden, so ist das durchaus möglich, wenngleich wir an dieser Stelle weitaus weniger detailliert formalisieren können. Wir wissen insbesondere nicht, wie in einer solchen Zielsprache über Zustände oder Methodenaufrufe gesprochen wird.

Wir generieren im Falle von Contraints C zur Methodenspezifikation unter diesen Umständen nur das Paar von Formeln, welches die Eigenschaften des Snapshots für den Vor- bzw. Nachzustand der Methodenausführung beschreibt. Also $Th_C = (\theta^{pre}, \theta^{post})$.

Eine solche Abbildung ist unserer Ansicht nach durchaus gewinnbringend, da eine Anwendung, die eine andere Logiksprache benutzt, lediglich die Formalisierung von Zuständen und Methodenaufrufen mit ihren Rückgabewerten durch eine geeignete Transformation des Paares Th_C einbringen muß. Die eigentliche Struktur der Formeln θ^{pre} bzw. θ^{post} dürfte im allgemeinen aber weitgehend übernommen werden können, da die Übersetzungen von OCL-Ausdrücken und OCL-Constraints unabhängig voneinander sind und man sich eigentlich erst auf der Ebene von Constraints mit Zuständen und Methodenaufrufen beschäftigt. Man beachte desweiteren, daß die Übersetzung von OCL-Ausdrücken (d.h. wenn zur Auswertung ein Zustand bzw. Zustandspaar fixiert ist) wie sie bisher dargestellt wurde keine Konstrukte verwendet, die nicht prinzipiell auch in einer Logiksprache erster Stufe nachgebildet werden könnten. □

Bemerkung (Vereinfachung für die DL). Man beachte, daß die angegebenen Ausdrück für die Zielsprache **DL** noch zu vereinfachen sind, da wir Programmvariablen für die Darstellung der freien Variablen im Constraint verwenden wollen.

Da Programmvariablen in der **DL** als modale Konstanten angesehen werden können, benötigen wir die Quantifizierungen über diese Variablen nichtmehr. Außerdem spielen diese Variablen – die nun als Konstanten betrachtet werden!– keine Rolle mehr als Parameter für irgendwelche bei der Übersetzung erzeugten Funktionssymbole.

Ein entsprechender Bereinigungslauf über der oben angegebenen Formeln kann diese Vereinfachungen vornehmen und bereinigte Formeln generieren.

Man mache sich klar, daß das Ergebnis dann im Sinne von Abschnitt 2.2.3 korrekt ist. Der Grund dafür liegt wieder in der *impliziten* Allquantifikation über die Welten w_D und somit aller möglichen Interpretationen dieser modalen Konstanten. □

2.3.4.3 Neue Constraints in OCL Version 1.4.

`Definition`-Constraint. In den bisherigen OCL Versionen einschließlich Version 1.3 war die Wiederverwendung von mehrfach verwendeten OCL-Ausdrücken nur sehr eingeschränkt in Form von `let`-Ausdrücken möglich. Eine Wiederverwendung war somit *lokal* auf *einen* Constraint beschränkt, ein Benutzung des lokal definierten Ausdrucks in anderen Constraints derselben Kontextklasse war nicht möglich.

In der Version 1.4 von OCL wurde nun dafür ein neuer Constraint-Typ eingeführt, der genau solch eine Art von Wiederverwendung über mehrere Constraints derselben Kontextklasse hinweg erlaubt. Ein solcher Constraint C trägt den Stereotypen

`<<definition>>` und entspricht im wesentlichen einer (bzgl. der Kontextklasse `C`) *globalen* Definition einer Abkürzung.

Er hat im allgemeinen die Form

```
context [c:]C def:
  l1
  l2
  ...
  lN
```

wobei die OCL-Ausdrücke `l1`, ..., `lN` `let`-Ausdrücke darstellen.

Die Intention des neuen Constraint-Typs ist nun die folgende: Die durch die angegebenen `let`-Ausdrücke definierten Variablen bzw. Funktionen werden implizit als *Pseudo*-Attribute bzw. -Methoden der Kontextklasse `C` betrachtet und entsprechend bei der Verwendung in OCL-Ausdrücken notiert.

Die Namenskonventionen von OCL fordern dabei, daß keine Namenskonflikte mit bestehenden Attributen oder Methoden der Kontextklasse bzw. anderen definierten Abkürzungen in der gleichen Kontextklasse auftreten.

Die Behandlung diese neuen Constraint-Typs ist nun sehr einfach, da es im wesentlichen um eine Folge von `let`-Ausdrücken handelt, wir die Behandlung von `let`-Ausdrücken bereits vorgestellte haben und unsere Behandlung von `let`-Ausdrücken in keiner Weise auf den *lokalen* Charakter dieser Abkürzungstechnik in OCL beschränkt ist.

Wir übersetzen also einen solchen Constraint C einfach durch Abbildung der einzelnen `let`-Ausdrücke, die in dem Constraint angegeben sind. Man beachte, daß wir lediglich neue Funktionsymbole in Σ^* einführen und geeignete Axiome Ax_C generieren, die schließlich in die Axiomenmenge $Ax_{\texttt{exp}}$ für die Übersetzung von OCL-Ausdrücken `exp` einfließen, die eine solche Abkürzung verwenden! Der Constraint C an sich wird durch keinen Term bzw. keine Formel explizit verkörpert.

Die Namenskonventionen und unser vorgehen bei der Formalisierung von UML-Modellen stellen dabei sicher, daß keine Namenskonflikte entstehen. Zur Notation dieser Symbole wollen wir wieder die Punktnotation aus OCL anwenden.

2.3.5 Abschließende Bemerkungen zur Basisabbildung

Wir wollen in diesem Abschnitt das gesamte Vorgehen der in den vorangehenden Abschnitten entwickelten Basisabbildung von UML/OCL-Modellen in eine Logiksprache – insbesondere **DL** – noch einmal Revue passieren lassen und eine kurze Übersicht bieten.

Das gesamte Verfahren der Übersetzung geschieht nun in drei aufeinanderfolgenden Schritten, die unabhängige Aufgaben lösen und deren Umsetzung möglicherweise sogar *anwendungsspezifisch*[49] sein kann. Um die vorhandenen Ansatzpunkte und Spielräume zur Optimierung und Anpassung des Gesamtverfahrens an die Bedürfnisse einer speziellen Anwendung zu verdeutlichen, haben wir diese verschiedenen Ebenen schon beim

[49] Im KeY-System wären das beispielsweise die Anwendung der generierten Formeln zum Nachweis, ob eine Methode nach der Modellierung im UML-Modell $\mathcal{D}$ eine gewisse Eigenschaft (z.B Erhaltung des Liskov-Prinzips im Kontext einer Generalisierungs/Spezialisierungsbeziehung zwischen zwei Klassen) *unabhängig* von der Implementierung dieser Methode besitzt oder ob eine gegebene Implementierung der Methode ihrer Spezifikation genügt.

Entwurf des *Verfahrens* entsprechend dem Grundsatz einer *Separation of Concerns* sauber getrennt:

Gegeben sei ein Constraint C, der in eine Formel Th_C übersetzt werden soll. Wir verfahren dann folgendemaßen:

1. **Normalisierung von** C — Der Constraint C wird auf *OCL-Ebene* in eine Normalform C' gebracht, die gewisse Eigenschaften des Constraints C', der dann eigentlich durch die Abbildungsprozedur übersetzt wird, sicherstellt. Diese Eigenschaften können dann vom Verfahren selbst genutzt werden und dieses unter Umständen vereinfachen – man denke beispielsweise die Auflösung einer impliziten Verwendung des `collect`-Operators.

2. **Übersetzung des normalisierten Constraints** C' — Hier wenden wir die eigentliche Abbildungprozedur gemäß der Abschnitte 2.3.3 und 2.3.4 auf den normalisierten Constraint C' an und generieren so eine Formel $Th_{C'}$

3. **Nachbehandlung der Übersetzung** $Th_{C'}$ — Schlußendlich wird die generierte Formel $Th_{C'}$ auf der Ebene der Logiksprache entsprechend der Bedürfnisse der konkreten Anwendung, die eine solche Formalisierung von UML/OCL-Modellen verwendet, in eine geeignete Zielformel Th_C der Zielsprache transformiert. Dabei muß insbesondere die Semantik von Funktionssymbolen der Form $f@pre$, der Variablen $result$ sowie der punktierten Quantoren unter Verwendung der Ausdrucksmöglichkeiten der Zielsprache formalisiert werden.

   Doch auch weitere anwendungsabhängige Transformationen sind denkbar, beispielsweise um Formeln zu erzeugen, die für ein möglicherweise verwendetes Deduktionssystem besonders zugeschnitten sind. In der **DL** ersetzen wir zum Beispiel die logischen Variablen, die zu den freien Variablen des Constraints gehören, durch Programmvariable und verändern die erzeugten Formeln in der weiter vorne skizzierten Art und Weise.

Schließlich erhalten wir so das gewünschte Ergebnis Th_C der Übersetzung des Constraints C.

Das vorgestellte Verfahren wurde detailliert für unsere Zielsprache **DL** vorgestellt, die Verallgemeinerbarkeit unseres Vorgehens auf beliebige Logiksprachen oder sonstigen Formalismen, die eine Erweiterung einer prädikatenlogischen Sprache erster Ordnung darstellen, jedoch an allen notwendingen Stellen herausgehoben. Sofern die Zielsprache bzw. der Zielformalismus ausdrucksstark genug ist, um mit Zuständen bzw. Snapshots D eines UML-Modells $\mathcal{D}$ umgehen zu können, dürfte die Anpassung unseres Verfahrens keinen allzugroßen Aufwand bereiten, insbesondere sollte eine Neuentwicklung der gesamten Abbildung nunmehr nicht mehr notwendig sein.

2.4 Ein Anwendungsbeispiel

Wir wollen das vorgestellte Basisverfahren nun anhand einer Reihe von Constraints über einem etwas komplexeren UML-Modell *Vortragswelt* illustrieren, welches in Abbildung 2.2 dargestellt ist.

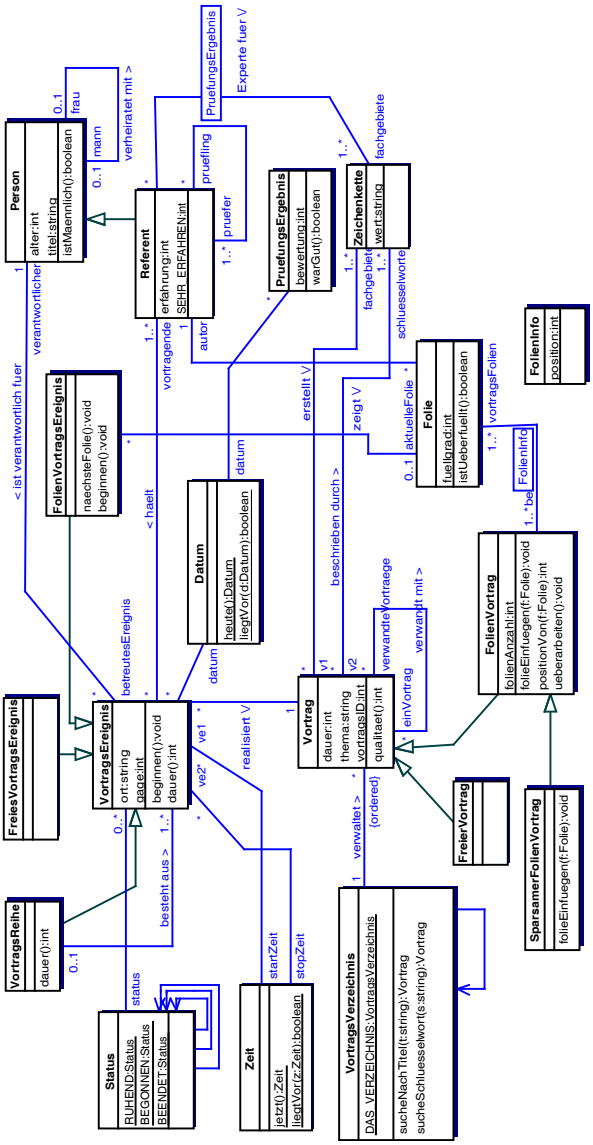


Abbildung 2.2: Vortragswelt — Ein komplexeres Anwendungsbeispiel.

Wir werden später in Abschnitt 3.4 nochmals auf dieses Anwendungsbeispiel und die zugehörigen Constraint zurückkommen, um die Wirkung einer optimierten Version der Abbildung zu demonstrieren und deren Ergebnisse mit den von der reinen Basisabbildung generierten Formeln zu vergleichen.

2.4.1 Einige Constraints über das Modell

Wir werden im folgenden einige natürlichesprachliche Anforderungen an die modellierte Miniwelt in Form von OCL-Constraints formalisieren und diese Constraints anschließend mit unserer Basisabbildung übersetzen.

Die Übersetzung wird dabei nicht wie bisher unter Verwendung von mathematischen Symbolen aufgeschrieben, sondern vielmehr über eine rein textuelle Darstellung der erzeugten Formeln festgehalten, die von einer Implementierung auf dem Bildschirm ausgegeben würde. Wir haben zur Generierung unsere eigene Implementierung herangezogen, um zumindest an dieser Stelle eine kleine Vorstellung von den Ergebnissen dieser Arbeit im realen Einsatz zu vermitteln. Zur besseren Lesbarkeit wurde der Text formatiert und die Variablenbezeichnungen vereinfacht. Die punktierten Quantoren $\dot{\forall}, \dot{\exists}$ wurden noch nicht durch eine entsprechende Bereinigung expandiert und sind durch `all'` bzw. `ex'` notiert. Wir verzichten an dieser Stelle außerdem auf die Darstellung der freien Variablen des Constraints durch Programmvariable. Für die Junktoren $\wedge, \vee, \rightarrow, \leftrightarrow$ schreiben wir im folgenden `&`,`|`,`->` bzw. `<->`. Die Enthaltensein-Relation $\in$ wird durch das Symbol `contains` verkörpert. Die weiteren Korrespondenzen zwischen den Symbolen in der bisherigen Notation und der rein textuellen Darstellung sollten im einzelnen problemlos klar werden.

2.4.1.1 Invarianten

Beispiel 32
Natürlichsprachliche Anforderung:

> Ein Prüfer wird von keinem Referenten geprüft, der von diesem Prüfer bewertet wurde.

OCL-Constraint C:

```
context Referent inv:
  self.pruefling->notEmpty implies
    not Referent.allInstances->exists(r|
        r.pruefling->includes(self) and
        self.pruefling->includes(r))
```

Übersetzung Th_C des OCL-Constraint:

```
all' self:Referent.(
  all' r1:Referent.contains(allInstancesOfReferent,r1) ->
    (ex' r0:Referent.contains(self.pruefling,r0) ->
       !ex' r:Referent.(
           contains(allInstancesOfReferent,r) &
           contains(r.pruefling,self) &
           contains(self.prruefling,r)
        )
    )
)
```

Beispiel 33
Natürlichsprachliche Anforderung:

> Die Prüfungsergebnisse eines Prüflings müssen sich mit der Zeit immer weiter verbessern, oder, falls es 30 Punkte oder mehr beträgt, nicht mehr unter 30 Punkte fallen.

OCL-Constraint C:

```
context Referent inv:
  self.pruefungsErgebnis[pruefer]->forAll(pe1,pe2|
      pe1.datum.liegtVor(pe2.datum) implies
         (pe1.bewertung < pe2.bewertung or
          pe2.bewertung >= 30))
```

Übersetzung Th_C des OCL-Constraint:

```
all' self:Referent.(
  all' self:Referent.let-0(self) =
         pruefungsErgebnis[pruefer](self) ->
    all' pe1:PruefungsErgebnis.(
      contains(let-0(self),pe1) ->
         all' pe2:PruefungsErgebnis.(
           contains(let-0(self),pe2) ->
              pe1.datum.liegtVor(pe2.datum) ->
                ( pe1.bewertung < pe2.bewertung |
                  pe2.bewertung >= #30
                )
         )
    )
)
```

Beispiel 34
Natürlichsprachliche Anforderung:

> Ein Referent hält ausschließlich Vorträge über seine Fachgebieten.

OCL-Constraint C:

```
context Referent inv:
  self.vortragsEreignis->collect(ve| ve.vortrag)->forAll(v|
    self.fachgebiete->intersection(v.fachgebiete)->notEmpty)
```

Übersetzung Th_C des OCL-Constraint:

```
all' self:Referent.(
 (  all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.
      collect-0(insert(s,ve)) =
          insert(collect-0(remove(s,ve)),ve.vortrag)
  & collect-0(Set_Of_VortragsEreignis::emptySet) =
          Bag_Of_Vortrag::emptyBag  ) ->
      all' v:Vortrag.(
        contains(collect-0(self.vortragsEreignis),v) ->
          ex' zk:Zeichenkette.
            contains(
              intersection(self.fachgebiete,v.fachgebiete)),zk)
)
```

Beispiel 35
Natürlichsprachliche Anforderung:

> Die von einem Vortragsverzeichnis verwalteten Vorträge sind bezüglich der Vortragsqualität absteigend geordnet.

OCL-Constraint C:

```
context VortragsVerzeichnis inv:
 Sequence{1 .. self.vortrag->size}->forAll(i,j| j >= i
  implies
    self.vortrag->at(i).qualitaet() >=
    self.vortrag->at(j).qualitaet())
```

Übersetzung Th_C des OCL-Constraint:

```
all' self:VortragsVerzeichnis.(
  (
    ( all' self:VortragsVerzeichnis.all i:Integer,j:Integer.(
       (#1 <= i) & (i <= size(sequence-1(self))) &
       (#1 <= j) & (j <= size(sequence-1(self))) & (i <= j) ->
           at(sequence-1(self),i) <= at(sequence-1(self),j))
    ) & (
      all' self:VortragsVerzeichnis.all i:Integer.(
        count(sequence-1(self),i) <= #1)
    ) & (
      all' self:VortragsVerzeichnis.(let-0(self) = sequence-1(self))
    ) & (
      all i:Integer.all' self:VortragsVerzeichnis.(
        contains(sequence-1(self),i)
        <-> ((#1 <= i) & (i <= size(self.vortrag)))
      )
    )
   ) -> all i:Integer.(
          contains(let-0(self),i) ->
           all j:Integer.(
             contains(let-0(self),j) ->
               (j >= i) ->
                ( at(self.vortrag,i).qualitaet() >=
                  at(self.vortrag,j).qualitaet() )
           )
   )
)
```

Beispiel 36
Natürlichsprachliche Anforderung:

> Alle Vorträge werden von dem gleichen Vortragsverzeichnis verwaltet und dieses Verzeichnis entspricht genau dem Verzeichnis, welches durch die statische Variable `DAS VERZEICHNIS` in VortragsVerzeichnis zugreifbar ist.

OCL-Constraint C:

```
context VortragsVerzeichnis inv:
 let alleVerz:Set(VortragsVerzeichnis) =
       Vortrag.allInstances->collect(v|
                         v.vortragsVerzeichnis)->asSet in
   alleVerz = Set{VortragsVerzeichnis.DAS_VERZEICHNIS}
```

Übersetzung Th_C des OCL-Constraint:

```
(
   (
     all' v:Vortrag.contains(allInstancesOfVortrag,v)
   ) & (
     all s:Set_Of_Vortrag.all' v:Vortrag.
        collect-1(insert(s,v)) =
           insert(collect-1(remove(s,v)),v.vortragsVerzeichnis)
      & collect-1(Set_Of_Vortrag::emptySet) =
           Bag_Of_VortragsVerzeichnis::emptyBag
   ) & (
     all' vv:VortragsVerzeichnis.(
       contains(collect-1(allInstancesOfVortrag),vv)
       <-> contains(asSet-1,vv)
     )
  ) & (
     alleVerz = asSet-1
  )
) -> all' vv:VortragsVerzeichnis.(
        contains(alleVerz,vv)
        <-> contains(
              insert(Set_Of_VortragsVerzeichnis::emptySet,
                     VortragsVerzeichnis.DAS_VERZEICHNIS)
             ,vv)
      )
```

Beispiel 37
Natürlichsprachliche Anforderung:

> Wir wollen eine alternative OCL-Formulierung zum vorangehenden Beispiel bieten:
>
> Alle Vorträge werden von dem gleichen Vortragsverzeichnis verwaltet und dieses Verzeichnis entspricht genau dem Verzeichnis, welches durch die statische Variable `DAS VERZEICHNIS` in VortragsVerzeichnis zugreifbar ist.

OCL-Constraint C:

```
context VortragsVerzeichnis inv:
 let alleVerz:Set(VortragsVerzeichnis) =
       Vortrag.allInstances->collect(v|
                              v.vortragsVerzeichnis)->asSet in
   alleVerz->size = 1 and
   alleVerz->forAll(v|
                   v = VortragsVerzeichnis.DAS_VERZEICHNIS)
```

Übersetzung Th_C des OCL-Constraint:

```
(
   (
     all' v:Vortrag.contains(allInstancesOfVortrag,v)
   ) & (
     all s:Set_Of_Vortrag.all' v:Vortrag.
        collect-1(insert(s,v)) =
           insert(collect-1(remove(s,v)),v.vortragsVerzeichnis)
```

```
      & collect-1(Set_Of_Vortrag::emptySet) =
           Bag_Of_VortragsVerzeichnis::emptyBag
   ) & (
     all' vv:VortragsVerzeichnis.(
       contains(collect-1(allInstancesOfVortrag),vv)
       <-> contains(asSet-1,vv)
     )
  ) & (
     alleVerz = asSet-1
  )
) -> size(alleVerz) = #1 &
     all' vv:VortragsVerzeichnis.(
        contains(alleVerz,vv) ->
            vv = VortragsVerzeichnis.DAS_VERZEICHNIS
      )
)
```

Beispiel 38
Natürlichsprachliche Anforderung:

> Alle Folien, die ein Referent überhaupt verwendet, wurden auch von diesem Referenten erstellt.

OCL-Constraint C:

```
context Referent inv:
 self.vortragsEreignis->select(ve|
     ve.oclIsKindOf(FolienVortragsEreignis))
 ->collect(ve| ve.vortrag.oclAsType(FolienVortrag))
 ->collect(fv| fv.vortragsFolien)->forAll(f|f.autor = self)
```

Übersetzung Th_C des OCL-Constraint:

```
all' self:Referent.(
 (
    collect-0(Set_Of_VortragsEreignis::emptySet) =
          Bag_Of_FolienVortrag::emptyBag
  & collect-1(Bag_Of_FolienVortrag::emptyBag) =
          Bag_Of_Folie::emptyBag
  & all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.
       collect-0(insert(s,ve)) =
          insert(collect-0(remove(s,ve)),
                 asType-Vortrag-FolienVortrag(ve.vortrag))
  & all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.(
        (ex fve:FolienVortragsEreignis.(fve = ve)) ->
            select-0(insert(s,ve)) =
               insert(select-0(s),ve)
    )
  & all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.(
        (!ex fve:FolienVortragsEreignis.(fve = ve)) ->
            select-0(insert(s,ve)) =
                select-0(s)
    )
  & select-0(Set_Of_VortragsEreignis::emptySet) =
         Set_Of_VortragsEreignis::emptySet
```

```
  & all b:Bag_Of_FolienVortrag.all' fv:FolienVortrag.(
      collect-1(insert(b,fv)) =
          union(collect-1(b),fv.vortragsFolien)
    )
  & all' fv:FolienVortrag.(
      asType-Vortrag-FolienVortrag(fv) = fv
    )
 ) -> all' f:Folie.(
        contains(
          collect-1(collect-0(select-0(self.vortragsEreignis)))
          ,f) -> f.autor = self
      )
)
```

Beispiel 39
Natürlichsprachliche Anforderung:

Verwandte Vorträge haben mindestens zwei Schlüsselworte gemein.

OCL-Constraint C:

```
context Vortrag inv:
 self.verwandteVortraege->forAll(v|
   v.schluesselworte->intersection(
         self.schluesselworte)->size >= 2)
```

Übersetzung Th_C des OCL-Constraint:

```
all' self:Vortrag.all' v:Vortrag.(
  contains(self.verwandteVortraege,v) ->
       size(intersection(v.schluesselworte,
                         self.schluesselworte)) >= #2
)
```

Beispiel 40
Natürlichsprachliche Anforderung:

Jede Vortragsreihe umfaßt mindestens zwei Folienvorträge, einen Vortrag mit dem Schlüsselwort ,,Eroeffnungsvortrag' und keine weitere Vortragsreihe.

OCL-Constraint C:

```
context VortragsReihe inv:
 self.vortragsEreignis->select(ve|
    ve.oclIsKindOf(FolienVortragsEreignis))->size >= 2 and
 self.vortragsEreignis->exists(ve|
    ve.vortrag.schluesselworte->exists(zk|
      zk.wert = 'Eroeffnungsvortrag')) and
 self.vortragsEreignis->select(ve|
    ve.oclIsKindOf(VortragsReihe))->isEmpty
```

Übersetzung Th_C des OCL-Constraint:

```
all' self:VortragsReihe.(
 (   all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.(
       (ex fve:FolienVortragsEreignis.(fve = ve)) ->
```

```
          select-0(insert(s,ve)) =
              insert(select-0(s),ve)
     )
   & select-0(Set_Of_VortragsEreignis::emptySet) =
        Set_Of_VortragsEreignis::emptySet
   & all' ve:VortragsEreignis.all s:Set_Of_VortragsEreignis.(
       (!ex vr:VortragsReihe.(vr = ve)) ->
           select-1(insert(s,ve)) = select-1(s)
     )
   & all' ve:VortragsEreignis.all s:Set_Of_VortragsEreignis.(
       (!ex fve:FolienVortragsEreignis.(fve = ve)) ->
           select-0(insert(s,ve)) = select-0(s)
     )
   & select-1(Set_Of_VortragsEreignis::emptySet) =
        Set_Of_VortragsEreignis::emptySet
   & all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.(
       (ex vr:VortragsReihe.(vr = ve)) ->
          select-1(insert(s,ve)) =
             insert(select-1(s),ve)
     )
 ) -> (
        (size(select-0(self.vortragsEreignis)) >= #2) &
        ex ve:VortragsEreignis.(
          contains(self.vortragsEreignis,ve) &
          ex zk:Zeichenkette.(
            contains(ve.vortrag.schluesselworte,zk) &
            zk.wert = 'Eroeffnungsvortrag'
          )
        ) &
        all ve:VortragsEreignis.
          !contains(select-1(self.vortragsEreignis),ve)
)
```

2.4.1.2 Vor-/Nachbedingungen

Beispiel 41
Natürlichsprachliche Anforderung:

> Vor dem Aufruf der Methode beginnen in FolienVortragsEreignis muß gelten, daß der Status des Ereignisses „ruhend" entspricht. Nach dem Aufruf der Methode gilt das Ereignis als „begonnen" und es wird die Folie mit der Foliennummer 1 gezeigt.
>
> Man beachte die Behandlung der 0..1-Assoziation im Teilausdruck `self.aktuelleFolie`.

OCL-Constraint C:

```
context FolienVortragsEreignis::beginnen():void
 pre: self.status = Status.RUHEND
 post: self.status = Status.BEGONNEN
 post: self.aktuelleFolie->notEmpty and
       self.aktuelleFolie.folienInfo->exists(fi|
         fi.folienVortrag = self.vortrag and fi.position = 1)
```

Übersetzung Th_C des OCL-Constraint:

```
all' self:FolienVortragsEreignis.(
  self.status = Status.RUHEND ->
   <{
      self.beginnen ();
   }>( (self.status = Status.BEGONNEN) &
        ex' f:Folie.(
           contains(
             insert(Set_Of_Folie::emptySet,
                    self.aktuelleFolie)
            ,f)
        ) &
        ex' fi:FolienInfo.(
           contains(self.aktuelleFolie.folienInfo,fi) &
           fi.folienVortrag = self.vortrag &
           fi.position = #1
        )
     )
)
```

Beispiel 42
Natürlichsprachliche Anforderung:

> Die Methode `qualitaet` liefert im Falle von Folienvorträgen einen umso größeren Wert, je näher der mittlere Füllgrad der Folien des Vortrags an 0.5 liegen.

OCL-Constraint C:

```
context Vortrag::qualitaet():Integer
post:
   let avgFuellgrad(fv: FolienVortrag): Real =
       (fv.vortragsFolien->collect(f| f.fuellgrad)->sum /
        fv.vortragsFolien->size) in
   self.oclIsKindOf(FolienVortrag) implies
     FolienVortrag.allInstances->forAll(fv|
       ((avgFuellgrad(self) - 0.5).abs() <=
        (avgFuellgrad(fv) - 0.5).abs())
          implies result >= fv.qualitaet())
```

Übersetzung Th_C des OCL-Constraint:

```
all' self:Vortrag.(
  true ->
   <{
      Integer result = self.qualitaet ();
   }>(
 (collect-0(Set_Of_Folie::emptySet) =
          Bag_Of_Integer::emptyBag
  & all s:Set_Of_Folie.all' f:Folie.(
       collect-0(insert(s,f)) =
          insert(collect-0(remove(s,f)),f.fuellgrad)
    )
  & all' fv:FolienVortrag.(
       avgFuellgrad(fv) =
```

```
         ( sum(collect-0(fv.vortragsFolien)) /
           size(fv.vortragsFolien) )
    )
  & all' fv:FolienVortrag.
        contains(allInstancesOfFolienVortrag,fv)
 ) -> ex' fv:FolienVortrag.(fv = self) ->
         all' fv:FolienVortrag.(
            contains(allInstancesOfFolienVortrag,fv) ->
               ( (abs(avgFuellgrad(self) - #0.5) <=
                 (abs(avgFuellgrad(fv) - #0.5)) ) ->
                     result >= fv.qualitaet())
         )
      )
)
```

Beispiel 43
Natürlichsprachliche Anforderung:

> Nach dem Aufruf der Methode `folieEinfuegen` in FolienVortrag wurde die übergebene Folie in die Menge der Vortragsfolien als letzte Folie eingefügt, sofern die Folie nicht schon vorhanden war.
>
> Man beachte die Behandlung von `oclIsNew`, sowie dem `@pre`-Operator. Außerdem entstehen bei der termbasierten Übersetzung des `select`-Ausdrucks *zusätzliche* Parameter!

OCL-Constraint C:

```
context FolienVortrag::folieEinfuegen(f:Folie):void
 pre: self.vortragsFolien->excludes(f)
 post: FolienInfo.allInstances->select(fi|
         fi.oclIsNew and
         fi.folienVortrag = self and
         fi.vortragsFolien = f and
         fi.position = self.vortragsFolien@pre->size + 1
       )->size = 1
```

Übersetzung Th_C des OCL-Constraint:

```
all' f:Folie.all' self:FolienVortrag.(
 ( !contains(self.vortragsFolien,f) &
   all' fi:FolienInfo.(fi.created@pre <-> fi.created) &
   all' fv:FolienVortrag.(
      fv.vortragsFolien@pre = fv.vortragsFolien
   )
 ) -> <{
         self.folieEinfuegen (f);
       }>(
 (
  all' self:FolienVortrag.all s:Set_Of_FolienInfo.
      all' f:Folie.all' fi:FolienInfo.(
       (
         (created(fi) & !created@pre(fi)) &
         fi.folienVortrag = self &
         fi.vortragsFolien = f &
         di.position = size(self.vortragsFolien@pre) + #1
```

```
      ) ->
         select-0(insert(s,fi),f,self) =
           insert(select-0(s,f,self),fi)
 ) &
 all' fi:FolienInfo.
     contains(Set_Of_FolienInfo::allInstancesOfFolienInfo,fi) &
 all' self:FolienVortrag.all s:Set_Of_FolienInfo.
     all' f:Folie.all' fi:FolienInfo.(
      !(
        (created(fi) & !created@pre(fi)) &
        fi.folienVortrag = self &
        fi.vortragsFolien = f &
        di.position = size(self.vortragsFolien@pre) + #1
      ) ->
         select-0(insert(s,fi),f,self) =
           select-0(s,f,self)
 ) &
 all' self:FolienVortrag.all' f:Folie.(
     select-0(Set_Of_FolienInfo::emptySet,f,self) =
          Set_Of_FolienInfo::emptySet
 )
) -> size(select-0(allInstancesOfFolienInfo,f,self)) = #1))
```

Kapitel 3

Optimierung der Abbildung

3.1 Allgemeines

Die in Kapitel 2 vorgestellte Basisversion der Übersetzung stellt eine naheliegende Methode zur Abbildung von OCL-Ausdrücken in die **DL** dar. Sie ist vor allem getrieben von der Prämisse, einen Term (bzw. eine Formel) $\lceil \mathtt{e} \rceil$ zu generieren, der in seiner syntaktischen Struktur sehr nahe an dem ursprünglich zu übersetzenden OCL-Ausdruck `e` ist, da es sinnvoll erscheint, anzunehmen, daß der Modellierer, der die OCL-Constraints formuliert hat, diese Constraints auch gut versteht bzw. in der Lage ist, diese zu lesen und damit eine syntaktische Verwandtschaft zwischen dem erzeugten Term $\lceil \mathtt{e} \rceil$ und dem ursprünglichen OCL-Ausdruck `e` das Verstehen der generierten Formeln erleichtert.

Wie bei vielen Problemen erhält man bei der Anwendung *eines* Lösungsansatzes auf eine große Menge verschiedenartiger Probleminstanzen eine oftmals *suboptimale* Lösung für eine ganze Reihe derartiger Instanzen. In einem solchen Fall wäre es wünschenswert, sozusagen *nicht* jeden Nagel mit *demselben* Hammer zu bearbeiten, sondern verschiedene „spezialisierte" Hämmer für unterschiedliche Teilklassen von Probleminstanzen zu verwenden: Man wünscht sich Möglichkeiten zur Optimierung.

Ein wesentlicher Bestandteil eines jeden Optimierungsproblems ist bekanntlich das Optimierungskriterium beziehungsweise die zu optimierende Gütefunktion. Selbst für unsere spezielle Aufgabe – der Abbildung von UML/OCL-Modellen in Formeln einer Logiksprache, insbesondere **DL** – gibt es viele verschiedene Alternativen, die man sich vorstellen kann und die wiederum von dem Verwendungzweck der generierten Formeln abhängen: Lesbarkeit bzw. Verständlichkeit der generierten Formeln für einen Menschen, besondere Eignung für die Behandlung in einem benutzten Deduktionssystem, Kompaktheit der Formeln, Verwendung eines möglichst eingeschränkten Teils der Ausdrucksmöglichkeiten der Zielsprache und viele andere mehr.

Wir können und wollen hier gar nicht alle solche Möglichkeiten detailliert besprechen, sondern beschränken uns im weiteren auf die ausführliche Betrachtung und Behandlung *eines* solchen Kriteriums für eine *spezielle* Anwendung: Dem KeY-System. Das Vorgehen für andere Anwendungen sollte dann leicht ableitbar sein.

Im KeY-System werden die generierten Formeln als Eingabe für eine Verifikationskomponente verwendet, die selbst wiederum auf einem interaktiven und teilautomatisierten Beweisssystem aufbaut und diese Deduktionskomponente zum formalen Nach-

weis von sogenannten *Beweisverpflichtungen* benutzt.

Besonders wichtig und durchaus charakteristisch für diese Anwendung ist, daß ein *interaktives* Beweisystem zum Zuge kommt, da die Beweisaufgaben im Umfeld der formalen Methoden in der Softwareentwicklung – insbesondere der Programmverifikation – im allgemeinen so schwer sind, daß eine vollständige Automatisierung nicht möglich ist. Vielmehr führt ein menschlicher Beweiser in Zusammenarbeit mit dem KeY-System formale Beweise, wobei die wesentlichen Kernideen des Beweises durch den (intelligenten) Menschen generiert werden und KeY hauptsächlich die Korrektheit und detaillierte Ausführung eines Beweises sicherstellt und einfache, routinemäßige Teilprobleme vollautomatisch und damit eigenständig löst.

Aus dieser Situation ergibt sich nun in natürlicher Weise ein Kriterium, welches uns für die Anwendbarkeit der generierten Formeln in diesem Umfeld entscheidend zu sein scheint: Die Lesbarkeit und Verständlichkeit der generierten Formeln für den menschlichen Beweiser. Wir wollen daher im folgenden besonderes Augenmerk auf Techniken legen, die dieses Kriterium optimieren.

Im allgemeinen ist zu erwarten, daß es viele unterschiedliche Techniken geben wird, um ein für die betrachtete Anwendung relevantes Gütekriterium zu optimieren. Diesen Techniken mag manchmal ein ähnlicher Gedanke zugrunde liegen, sie können aber genausogut auf völlig unterschiedlichen Konzepten basieren. Diese verschiedene Techniken zur Optimierungen müssen schließlich unter Rückgriff auf den zu übersetzenden OCL-Ausdruck – insbesondere seiner syntaktischen Struktur – algorithmisch formuliert werden, damit sie in einer Implementierung einer Abbildung realisiert werden können. Ein solche algorithmische Formulierung wollen wir als *Heuristik* bezeichnen.

Wir fordern daher für eine Implementierung einer solchen Abbildung, daß sie ausreichend flexibel gestaltet wurde, um *beliebige* Gütekriterien unterstützen zu können und für jedes solche Kriterium *beliebig viele*, *verschiedene* Heuristiken einzubinden gestattet. Das Einbinden solcher Heuristiken sollte mit besonderem Hinblick auf das *iterative* Entwickeln einer optimierten Abbildung (bezüglich des anwendungsspezifischen Gütekriteriums) *dynamisch* möglich sein, d.h. der Benutzer sollte zur Laufzeit bestimmte Heuristiken für die Übersetzung einbinden oder ausschalten können.

Wir haben bei unserer eigenen Implementierung dieser Ideen versucht, diesem Anspruch weitgehend gerecht zu werden.

Wir wollen an dieser Stelle nun kurz beleuchten, wie das Zusammenspiel zwischen der Basisabbildung und den Optimierungen in Form von Heuristiken (mit Hinblick auf eine Implementierung) funktionieren kann:

Unsere Basisabbildung, die wir in Kapitel 2 ausführlich besprochen haben, stellt gewissermaßen unsere Standard-Methode dar, um beliebige OCL-Ausdrücke zu übersetzen. Zur Verbesserung dieser Standardmethode hinsichtlich des anwendungsspezifischen Gütekriteriums – und damit dem Zuschnitt der Abbildung auf die Bedürfnisse einer *speziellen* Anwendung – formulieren wir verschiedene Heuristiken. Wie gerade erwähnt orientieren sich diese Heuristiken im allgemeinen an der syntaktischen Struktur des zu übersetzenden OCL-Constraints und werden erwartungsgemäß jeweils *nicht* auf beliebige OCL-Ausdrücke anwendbar sein, vielmehr wird jede einzelne Heuristik auf ein kleines Fragment von OCL abzielen und dort Verbesserungen ermöglichen.

Für die eigentliche Abbildung von Constraints gehen wir wie folgt vor: Der Constraint wird *vor* der Übersetzung an eine Übersetzungssteuerungskomponente übergeben, die die einzelnen Heuristiken verwaltet und diese auffordert, den Constraint

zu analysieren und zu entscheiden, an welchen Stellen bzw. Teilausdrücken des Constraints sie anwendbar sind.

Die entsprechenden Stellen werden dann durch die Steuerungskomponente markiert, so das bei dem eigentlichen Übersetzunglauf später einfach entschieden werden kann, wie ein Teilausdruck genau zu behandeln ist. Für nicht-markierte Teilausdrücke wird als Default-Methode die Basisabbildung gewählt. Geeignete Konsistenzprüfungen oder Bereinigungen (zum Beispiel bei Mehrfachmarkierungen) müssen gegebenenfalls von der Steuerungskomponente durchgeführt werden.

Damit ist für jeden einzelnen Teilausdruck des OCL-Constraints definiert, wie dieser Teilausdruck im eigentlichen Abbildungsprozeß zu behandeln ist. Ein solcher markierter Constraint wird also an die eigentliche Abbildungskomponente übergeben, die eine Behandlung entsprechend der Markierungen der Teilausdrücke vornimmt. Man könnte sich dabei vorstellen, daß es gerade bei vielen verschiedenen Heuristiken sinnvoll wäre, das diese Übersetzungskomponente, die eigentliche Ausführung der Abbildung für einen *einzelnen*, *markierten* Teilausdruck an die zu der Markierung gehörende Heuristik überträgt, und dann im wesentlichen die Realisierung der Basisabbildung und die gesamte Verwaltung der Übersetzungen der einzelnen Teilausdrücke während des Übersetzungslaufes übernimmt.

Dieses Modell hat den Vorteil, daß durch die Basisabbildung sichergestellt ist, daß prinzipiell *alle* OCL-Ausdrücke übersetzt werden können, und durch Heuristiken „lokal" (d.h. in Teilfragmenten von OCL) Verbesserungen erzielt werden. Dadurch wird außerdem die Formulierung von Heuristiken erheblich erleichtert, da man sich bei der Entwicklung von neuen Heuristiken lediglich auf lokale Bereiche der Sprache OCL beschränken kann und sich nicht mehr um die „globale" Vollständigkeit der Abbildung, d.h. die Frage, ob das optimierte Verfahren nun alle OCL-Ausdrücke behandeln kann, sorgen muß. Eine solche Vollständigkeit kann bei hoch spezialisierten Abbildungen, die ohne solche Heuristiken entwickelt wurden und Constraints notwendigerweise gleichzeitig über mehrere syntaktische Ebenen hinweg behandeln müssen, nicht mehr problemlos nachzuvollziehen sein.

Durch diese Trennung in Basisabbildung und Heuristiken könnte gleichfalls ein formaler Nachweise der Korrektheit des gesamten Verfahrens vereinfacht werden, da man sich bei einem bestehenden Korrektheitsbeweis für die Basisabbildung weitgehend auf den Nachweis der Korrektheit der neuen Heuristiken konzentrieren muß.

Die Technik der Verwendung von Heuristiken macht das ganze Verfahren – im Vergleich zu einer einzigen monolithischen und auf eine spezielle Anwendung getrimmten Abbildung – also hoch flexibel, leichter verständlich und sogar dynamisch anpaßbar!

Bemerkung (Heuristiken vs. Rewriting). Wie in Kapitel 2 wollen wir hier nochmals unserer Auffassung Nachdruck verleihen, daß die *explizite* Unterstützung von Optimierungen schon im Abbildungsprozeß wesentlich allgemeiner und mächtiger ist, als eine sich an die Übersetzung anschließende Vorverarbeitung der generierten Formeln durch Rewriting-Techniken *innerhalb* Deduktionssystem selbst.

Rewriting-Methoden arbeiten ausschließlich auf der logischen Ebene – insbesondere *ohne* Rückgriff auf das ursprüngliche UML/OCL-Modell, da sie *innerhalb* des Deduktionssystems durchgeführt werden, für das im allgemeinen die Herkunft der zu verarbeitenden Formeln *keine* Rolle spielt.

Im Gegensatz dazu arbeiten unsere Heuristiken gerade auf der Ebene der UML/-OCL-Beschreibung. Ihnen stehen daher alle Information über das UML/OCL-Modell

zur Verfügung, die überhaupt verfügbar sind. Diese Informationen können somit durch die Heuristiken für Optimierungsmaßnahmen genutzt werden.

Wir sind der Meinung, daß es bei der ausschließlichen Anwendung von Rewriting unter Umständen *zu spät* sein kann, um *grundlegende* Veränderungen in der Darstellung von zu formalisierenden Sachverhalten zu erzielen und damit eine Verbesserung – im Vergleich zu unseren Heuristiken – nur eingeschränkt möglich ist.

Eine *beliebige* zu optimierende Zielfunktion wir im allgemeinen leichter in Form von Heuristiken behandelt werden können, als in einem Deduktionssystem in Form von Rewriting-Regeln.

Ein weiterer bemerkenswerter Vorteil von Heuristiken ist sicherlich ihre mögliche Kontextabhängigkeit bei der Anwendung, das bedeutet, daß ein zu formalisierender Sachverhalt aus einem Modell in Abhängigkeit von dem Umfeld, in dem er in einem OCL-Ausdruck verwendet wird, übersetzt werden kann, und damit ein und derselbe Sachverhalt an verschiedenen Stellen in einem Constraint (möglichweise vollständig) unterschiedlich abgebildet wird. Wir glauben nicht, daß es in solch einfachen Maße möglich ist, mit reinen Rewriting-Methoden ein solches Verhalten der Abbildung nachzuahmen.

Und es sei nochmals darauf hingewiesen, daß wir durch die Unterstützung von Heuristiken lediglich einen *zusätzlichen*, expliziten Freiheitsgrad zur Optimierung bei unserem Verfahren gewinnen. Der Einsatz von Rewriting-Techniken ist nach einer solchen optimierten Abbildung immer noch möglich und vielleicht sogar gewinnbringend.

Grundvoraussetzung dafür ist natürlich, daß das entsprechende Deduktionssystem eine Steuerung bzw. Kontrolle über eine solche Behandlung durch Rewriting nach außen bietet und Rewriting überhaupt unterstützt!

□

Wir wollen nun in Abschnitt 3.2 einen ersten solchen „spezialisierten" Hammer für eine Fragment von OCL vorstellen, der die Lesbarkeit der generierten Formeln verbessern soll, und die Ergebnisse der Optimierung an dem im Abschnitt 2.4 vorgestellten Anwendungsbeispiel verdeutlichen und diskutieren.

3.2 Darstellung von OCL-Kollektionen durch logische Formeln

Obwohl die durch die Basisabbildung erzeugten Terme und Formeln in ihrem syntaktischen Aufbau sehr stark an die ursprünglich zu übersetzenden OCL-Ausdrücke erinnern, erscheinen sie gelegentlich doch unnötig kompliziert und schwer zu verstehen.

Eine nähere Betrachtung der generierten Formeln zeigt, daß die Hauptursache dafür in den während der Übersetzung zusätzlich eingeführten Axiomen liegt, welche dazu dienen, die Interpretation der ebenfalls während der Abbildung neu eingeführten Funktionssymbole in Σ^* entsprechend der OCL-Semantik einzuschränken.

Selbst für recht kleine OCL-Ausdrücke können auf diese Weise eine Fülle von zusätzlichen Axiomen bei der Übersetzung entstehen, was dazu führt, daß die Ergebnisformel Th_C für einen Constraint C ungemein lang wird.

Zum anderen stellen alle diese Axiome semantische Informationen über die relevanten Größen aus dem UML/OCL-Modell auf recht *implizite* und *verteilte* Art und Weise in der Ergebnisformel dar, was das Verstehen der Intention hinter dieser Formel vielfach erschwert! Man muß die (möglicherweise zahlreichen) Axiome Schritt für

Schritt durchgehen, um die *exakte*[1] Bedeutung der einzelnen Symbole zu verstehen, und darf dabei nicht den Überblick verlieren.

Ein Beispiel soll das Problem verdeutlichen:

Beispiel 44
Betrachten wir wieder unser einfaches Beispielmodell aus Abbildung 2.1. Wir wollen nun ausdrücken, daß für jedes Unternehmen zu jedem Zeitpunkt gilt, daß die Menge der über 30-jährigen Angestellten, die ausschließlich überzogene Konten besitzen, leer ist und formulieren dazu den Constraint C

```
context c:Company
  inv: c.employees->select(e|
             e.age > 30 and
             not Account.allInstances->exists(a|
                    a.accountOwner = e and
                    a.accountBalance > 0)
          )->isEmpty
```

Als Übersetzung Th_C erhalten wir mit unserer Basisabbildung schließlich

$$
\begin{aligned}
&\dot{\forall} a{:}Account\,(a \in allInstances_{Account}) \wedge \\
&((select_E(emptySet_{Person}) \doteq emptySet_{Person}) \wedge \\
&(\forall s{:}Set_{Person} \dot{\forall} e{:}Person\,(\\
&\quad (e.age > 30 \wedge \\
&\quad \neg(\dot{\exists} a{:}Account(a \in allInstances_{Account} \wedge \\
&\qquad\qquad a.accountOwner \doteq e \wedge \\
&\qquad\qquad a.accountBalance > 0)) \\
&\quad) \rightarrow select_E(insert(s,e)) \doteq insert(select_E(s),e))) \wedge \\
&(\forall s{:}Set_{Person} \dot{\forall} e{:}Person\,(\\
&\quad \neg(e.age > 30 \wedge \\
&\quad \neg(\dot{\exists} a{:}Account(a \in allInstances_{Account} \wedge \\
&\qquad\qquad a.accountOwner \doteq e \wedge \\
&\qquad\qquad a.accountBalance > 0)) \\
&\quad) \rightarrow select_E(insert(s,e)) \doteq select_E(s))) \\
&\rightarrow \dot{\forall} c{:}\,Company \dot{\forall} p{:}Person\,\neg(p \in select_E(c.employees))
\end{aligned}
$$

Die Übersetzung dieses sicherlich etwas komplexeren Constraints ist zunächst nicht ganz einfach zu verstehen. Man benötigt einige Augenblicke, bis man die einzelnen Teile der Formeln richtig interpretiert.

Man beachte vor allem, daß die Formel, die eigentlich der Übersetzung des Constraints (ohne die Betrachtung der Axiome) entspricht, hinter dem Implikationspfeil in der letzten Zeile steht und an sich sehr kurz ist.

[1] Die Verwendung der gleichen Bezeichner aus OCL hilft zwar, löst das Problem aber nur teilweise. Man denke beispielsweise an den `select`-Operator, bei dem die Intention durch den Namen schon offeriert wird, das genau verwendete Filterkriterium hingegen *nicht* einfach aus dem Namen zu erkennen ist!

Sie ist jedoch *vollkommen nichtssagend* bzw. *unverständlich*, wenn man sie *ohne* die Axiome in den vorangehenden Zeile ließt:

$$\dot{\forall} c{:}\, Company \dot{\forall} p{:}Person\, \neg(p \in select_E(c.employees))$$

Welche Aussage steht denn hinter dieser einen Formel?

Das sollte verdeutlichen, was wir oben mit *impliziter* und *verteilter* Darstellung von semantischen Informationen meinten: Ohne die Betrachtung der Axiome ist der eigentliche Übersetzungsterm nicht vollständig zu verstehen.

Die generierte Formel ist jedoch sicherlich nicht der Weisheit letzter Schluß: Würde man einen Menschen beauftragen, die oben angegebene natürlichsprachliche Aussage über der Signatur $\Sigma_{\mathcal{D}}$ in der Logik zu formalisieren, so würde wahrscheinlich eine Formel der folgenden Art entstehen:

$$\begin{aligned}\dot{\forall} c{:}\, Company \dot{\forall} p{:}Person\, \neg(&p \in c.employees \wedge \\ &p.age > 30 \wedge \\ &\neg(\dot{\exists} a{:}Account(a.accountOwner \doteq p \wedge \\ &\qquad a.accountBalance > 0))\end{aligned}$$

Das Wissen aus dem ursprünglichen OCL-Ausdruck ist hier *explizit* und *lokal* in der Formel enthalten, die den eigentlichen Constraint formalisiert; keine zusätzlichen Axiome wurden verwendet. Dadurch wirkt diese Formulierung ungemein klarer und ist unbestreitbar verständlicher.

Wir sollten es also bei der Übersetzung solcher Constraints bzw. solcher Ausdrücke nach Möglichkeit vermeiden, neue Funktionssymbole und damit neue Axiome einzuführen.

Eine Analyse unserer Basisabbildung ergibt, daß diese neuen Symbole hauptsächlich bei der Übersetzung von Eigenschaften von Kollektionstypen aus OCL – beispielsweise `select`, `collect` oder `asSequence` – generiert werden.

Es liegt daher nahe, nach einer *anderen* Darstellungsform für Kollektionen in der Logik zu suchen, die es erlaubt, die in den ursprünglichen OCL-Ausdrücken verkörperte Information – wie im obigen Beispiel – *expliziter* und *lokaler* darzustellen.

Für Mengen liegt eine solche alternative Möglichkeit zur Darstellung recht nahe: Anstatt ein Menge s durch einen Term t_s über einem ADT Set_T zu beschreiben, könnten wir die zur Menge gehörende *charakteristische Funktion* bzw. das entsprechende *Prädikat* ϕ_s einsetzen, d.h. eine Formel $\phi_s(e)$ mit der Eigenschaft:

$$\forall e : T[e \in s \leftrightarrow \phi_s(e)] \ .$$

Diese Formeln beschreibt damit sozusagen eine charakteristische Eigenschaft, welche allen Elementen e der Kollektion s gemein ist und die diese Werte – im Gegensatz zu den anderen Werten aus dem Universum T, die *nicht* in der Kollektion enthalten sind – eindeutig als zu der Kollektion gehörend auszeichnet.

Diese Darstellungsform scheint einen entscheidenden Vorteil zu haben: Viele der Operationen auf Mengen in OCL können sehr einfach durch diese *prädikative Beschreibung* dargestellt werden, *ohne* neue Symbole (und damit Axiome) einführen zu müssen.

Wir wollen nun in Abschnitt 3.2.2 untersuchen, in wie weit sich diese ,,natürliche" Darstellungsform – die Darstellung einer Menge durch Beschreibung der Eigenschaften

ihrer Elemente – für die Mengen-Operationen in OCL eignet. Wir werden diese Frage am Ende sehr erfreulich beantworten können.

Naheliegender Weise stellt sich uns anschließend die Frage, ob es prinzipiell möglich ist, diese Beschreibungstechnik auch auf die anderen Spielarten von Kollektionen in OCL zu übertragen und wie sinnvoll bzw. vorteilhaft diese Beschreibungsmethode für diese Kollektionsarten ist. Wir werden uns dieser Frage für den Fall von Multimengen in Abschnitt 3.2.3 etwas ausführlicher zuwenden, und schließlich zu einem zunächst ernüchternden Ergebnis kommen.

In Abschnitt 3.2.4 wenden wir uns schlußendlich noch den Sequenzen zu, werden dort aber sehr schnell an die Grenzen der Nützlichkeit dieser Methode stoßen und uns dort deshalb sehr kurz halten.

Anschließend zeigen wir, daß sich diese neue Darstellungsform mit der termbasierten Repräsentation durch ADTs kombinieren läßt, die von unserer Basisabbildung verwendet wird.

Nach einer kurzen Zusammenfassung der Ergebnisse unserer Untersuchung werden wir in Abschnitt 3.3 gewissermaßen als Essenz aus den Betrachtungen der vorangehenden Abschnitten eine einfache, aber recht mächtige Heuristik angeben, die die prädikative Darstellung von Kollektionen in der **DL** zur Verbesserung der Lesbarkeit und Verständlichkeit der generierten Formeln mit unserer Basisabbildung verbindet.

Abschließend soll die Wirkung dieser Heuristik anhand der Constraints aus dem Anwendungsbeispiel für die Basisabbildung verdeutlicht werden.

3.2.1 Prädikative Darstellung von Kollektionen

3.2.1.1 Grundlegende Definitionen

Wir wollen zunächst versuchen zu präzisieren, was wir unter einem Prädikat bzw. einer prädikativen Beschreibung einer Menge verstehen:

Definition 16 (Prädikat über dem Universum T)
Sei Σ eine Signatur, $\Phi \in For_\Sigma^{DL}$ eine **DL***-Formel über dieser Signatur und e eine Variable der Sorte T. Dann bezeichnen wir das Paar (Φ, e) als* ***Σ-Prädikat über dem Universum*** *T.*

Wir schreiben in diesem Fall das Paar (Φ, e) zur Vereinfachung als $\Phi[e]$. ◁

Ein Prädikat repräsentiert sozusagen (bzgl. einer gegebenen Σ-Struktur $\mathcal{S} = (M, I)$ und einer Variablenbelegung β in $\mathcal{S}$) einen Indikator, der Elemente aus diesem Universum T auszeichnet: Die Variable e kann alle möglichen Werte $w \in I(T)$ des Universums T annehmen und dient als ausgezeichneter Eingabeparameter für den Test $\Phi[e]$, dessen Auswertung unter der Interpretation I und der Modifikation[2] β_e^w der Belegung β (an der Stelle w zu e) angibt, ob der Wert w einer bestimmten Eigenschaft genügt.

Betrifft diese bestimmte Eigenschaft das Enthaltensein in einer bestimmten Menge S, so können wir das Prädikat auch als grundlegend andere Beschreibungsform für die Menge S verstehen: Eine *prädikative Darstellung.*

Wir definieren daher allgemein:

[2]Wir bezeichnen im folgenden durch β_e^w die Modifikation der Variablenbelegung β an der Stelle e zum Wert w, d.h. die Variablenbelegung β' mit $\beta'(v) = w$ für $v = e$ und $\beta'(v) = \beta(v)$ sonst.

Definition 17 (Prädikative Beschreibung einer Menge)
Sei Σ eine Signatur, $\mathcal{S} = (M, I)$ eine Σ-Struktur und β ein Variablenbelegung in $\mathcal{S}$.

Sei S eine Menge über dem Universum T, d.h. $S \subseteq I(T)$, und $\Phi(e)$ ein Σ-Prädikat über dem Universum T.

Dann nennen wir $\Phi[e]$ eine ***prädikative Beschreibung der Menge*** *S (bzgl. der Σ-Struktur $\mathcal{S}$ und der Belegung β) genau dann wenn für alle $w \in I(T)$ gilt:*

$$w \in S \textbf{ gdw. } \mathcal{S}, \beta_e^w \models \Phi$$

Wir schreiben in diesem Fall $S \simeq_{\mathcal{S},\beta} \Phi[e]$, oder – sofern die betrachtete Struktur $\mathcal{S}$ aus dem Kontext klar ist – einfach $S \simeq_\beta \Phi[e]$. ◁

Bemerkung . Man beachte, daß die Formel Φ im allgemeinen beliebig viele freie Variablen enthalten kann und die im Prädikat ausgezeichnete e nicht als freie Variable in Φ vorkommen muß.

Insofern hängt die unter einer festen Struktur $\mathcal{S}$ durch ein Prädikat beschriebene Menge S wesentlich von der Belegung β der freien Variablen aus $\Phi[e]$ ab, die von e verschieden sind. □

Wir wollen nun versuchen, diese Darstellung auf Multimengen und Sequenzen zu erweitern.

Der Kernpunkt der Darstellung ist die Verwendung einer Formel als Indikator für die Elemente in der Kollektion bezüglich eines Universums T. Mag diese Formel für die vollständige Beschreibung einer Menge ausreichen, so ist das für Multimengen und Sequenzen nicht mehr der Fall:

Zur vollständigen Charakterisierung von Multimengen ist neben der formalen Beschreibung der Elemente – der *Trägermenge* – zusätzlich die Anzahl der Vorkommen dieser Elemente zu spezifizieren. Gleichermaßen ist für eine lückenlose Beschreibung von Sequenzen stattdessen die Reihenfolge der Elemente in der Sequenz unabdingbar.

Deshalb müssen wir im Falle von Multimengen bzw. Sequenzen unsere Definitionen von oben um eine formale Beschreibung der Anzahl bzw. der Positionen der Vorkommen erweitern:

Wir suchen zunächst nach einer Charakterisierung des Begriffs Multimenge, die Ähnlichkeiten und Unterschiede zu einer Menge deutlich macht. Wir führen deshalb den Begriff der ,,mengenartige Charakterisierung" für Multimenge ein.

Definition 18 (Mengenartige Charakterisierung einer Multimenge)
Sei B eine Multimenge mit Elementen aus einer Grundgesamtheit U.

Das Paar $(M_B, count_B)$ heißt ***mengenartige Charakterisierung einer Multimenge*** *B falls gilt:*

- *$count_B : U \to \mathbf{N}$ ist eine Abbildung, die jedem Element $e \in U$ die Anzahl der Vorkommen von e in B zuordnet und*
- *$M_B \subseteq U$ ist eine Menge, die genau die Elemente $e \in U$ enthält, die in B vorkommen, d.h. für alle $e \in U$ gilt:*

$$e \in M_B \textbf{ gdw. } count_B(e) > 0$$

Wir bezeichnen M_B als ***Trägermenge*** *von B und $count_B$ als* ***Anzahlfunktion*** *von B.* ◁

Bemerkung (Eindeutigkeit). Es ist klar, daß sich jede Multimenge B (mit Elementen aus einer Grundgesamtheit U) mengenartig charakterisieren läßt und diese Charakterisierung eindeutig ist. □

Diese semantische Form der Beschreibung von Multimengen wollen wir nun formalisieren:

Die Trägermenge kann durch ein Prädikat $\Phi_B[e]$ formalisiert werden, wohingegen für die formale Darstellung der Anzahlfunktion ein Funktionssymbol cnt_B ausgezeichnet wird. Um die Semantik dieses Funktionssymbols als formales Äquivalent zu der Anzahlfunktion unter Interpretationen I zu sichern, wird eine Menge Ax_B von **DL**-Formeln verwendet. Diese Formeln bezeichnen wir im weiteren als (nichtlogische) Axiome. Wir müssen wieder beachten, daß das Prädikat $\Phi_B[e]$ freie Variablen enthalten kann, die von e verschieden sind. Die durch $\Phi_B[e]$ konkret beschriebene Trägermenge B hängt also wieder von der Belegung dieser freien Variablen ab. Insofern hängt auch der Wert des Symbols cnt_B zu Formalisierung der Anzahlfunktion – neben dem betrachteten Element e selbst – von diesen freien Variablen aus $\Phi_B[e]$ ab.

Definition 19 (Prädikative Beschreibung einer Multimenge)
Sei Σ eine Signatur, $\mathcal{S} = (M, I)$ eine Σ-Struktur und β ein Variablenbelegung in $\mathcal{S}$. Sei B eine Multimenge über dem Universum T und $(M_B, count_B)$ die mengenartige Charakterisierung von B.

Sei $\Phi_B[e]$ ein Σ-Prädikat über dem Universum T, $p_1, \ldots, p_n$ die freien Variablen aus $\Phi_B[e]$, die von e verschieden sind, und $T_1, \ldots, T_n$ die entsprechenden Sorten.

Außerdem sei cnt_B:$T \times T_1 \times \ldots \times T_n \to$ INTEGER ein Funktionssymbol in Σ und $Ax_B \subseteq For_\Sigma^{DL}$ eine Menge von geschlossenen **DL***-Formeln über der Signatur Σ.*

Das Tripel $(\Phi_B[e], cnt_B, Ax_B)$ heißt ***prädikative Beschreibung der Multimenge*** *B (bzgl. der Σ-Struktur $\mathcal{S}$ und der Variablenbelegung β) genau dann wenn*

- *$\Phi_B[e]$ ist die prädikative Beschreibung der Trägermenge M_B bzgl. $\mathcal{S}$ und β, d.h. $M_B \simeq_{\mathcal{S},\beta} \Phi_B[e]$ und*
- *für alle Σ-Strukturen $\mathcal{S}' = (M', I')$ mit $M' = M$ gilt:*

$$\begin{array}{c} \text{Für alle } w \in T \text{ gilt: } val_{I',\beta_e^w}(cnt_B(e, p_1, \ldots, p_n)) = count_B(w) \\ \textbf{gdw.} \\ \mathcal{S}' \models Ax_B, \text{ also } \mathcal{S}' \text{ alle (geschlossenen) Formeln aus } Ax_B \text{ erfüllt und} \end{array}$$

- *$\mathcal{S} \models Ax_B$*

Wir schreiben in diesem Fall $B \simeq_{\mathcal{S},\beta} (\Phi_B[e], cnt_B[e], Ax_B)$ und lassen die Indizes weg, sofern die betrachtete Struktur und Variablenbelegung aus dem Kontext klar sind. ◁

Bemerkung (Notationelle Konvention). Wir wollen im folgenden aus Gründen der Lesbarkeit bei der Verwendung des Funktionssymbols cnt_B aus einer prädikativen Beschreibung einer Multimenge B die folgende Konvention anwenden:

Seien e die ausgezeichnete Variable aus $\Phi_B[e]$ und $p_1, \ldots, p_n$ die freien Variablen aus $\Phi_B[e]$, welche von e verschieden sind. Wir notieren dann in Termen statt der ausführlichen Angabe $cnt_B(e, p_1, \ldots, p_n)$ kurz $cnt_B[e]$.

Diese Schreibweise rechtfertig sich vor dem Hintergrund, daß wir eine prädikativen Beschreibung immer bezgl. einer bestimmten Variablenbelegung β betrachten, die gewissermassen die Werte der Parameter $p_1, \ldots, p_n$ und damit die eigentlich beschriebene Multimenge B festlegt. Beim späteren Anwenden dieser Beschreibungen betrachten wir feste Multimengen B und sind damit eigentlich nur an der ausgezeichneten Variable e interessiert. Diese Notation spiegelt diesen Aspekt sehr schön wieder. □

Die drei Anforderungen besagen, daß das Prädikat $\Phi_B[e]$ bzgl. der betrachteten Struktur $\mathcal{S}$ und Variablenbelegung β die Trägermenge exakt beschreibt, das die Axiome Ax_B gerade und ausschließlich die Semantik der Anzahlfunktion definieren und das die Anzahlfunktion exakt durch das ausgezeichnete Funktionssymbol $cnt_B[e]$ dargestellt wird.

Für Sequenzen gehen wir analog wie im Falle von Multimengen vor und definieren

Definition 20 (Mengenartige Charakterisierung einer Sequenz)
Sei S eine Sequenz mit Elementen aus einer Grundgesamtheit U.

*Das Paar $(M_S, places_S)$ heißt **mengenartige Charakterisierung einer Sequenz** S falls gilt:*

- *$places_S : U \to 2^{\mathbf{N}}$ ist eine Abbildung, die jedem Element $e \in U$ die Menge der Positionen der Vorkommen von e in S zuordnet und*
- *$M_S \subseteq U$ ist eine Menge, die genau die Elemente $e \in U$ enthält, die in S vorkommen, d.h. für alle $e \in U$ gilt:*

$$e \in M_S \textbf{ gdw. } places_S(e) \neq \emptyset$$

*Wir bezeichnen M_S als **Trägermenge** von S und $places_S$ als **Positionsfunktion** von S.*

◁

Bemerkung (Eindeutigkeit). Auch hier gilt, daß sich jede Sequenz S (mit Elementen aus einer Grundgesamtheit U) mengenartig charakterisieren läßt und diese Charakterisierung eindeutig ist. □

Diese semantische Form der Beschreibung von Sequenzen wollen wir nun in der analoger Weise formalisieren, wie wir es für Multimengen getan haben:

Definition 21 (Prädikative Beschreibung einer Sequenz)
Sei Σ eine Signatur, $\mathcal{S} = (M, I)$ eine Σ-Struktur und β ein Variablenbelegung in $\mathcal{S}$. Sei S eine Sequenz über dem Universum T und $(M_S, places_S)$ die mengenartige Charakterisierung von S.

Sei $\Phi_S[e]$ ein Σ-Prädikat über dem Universum T, $p_1, \ldots, p_n$ die freien Variablen aus $\Phi_S[e]$, die von e verschieden sind, und $T_1, \ldots, T_n$ die entsprechenden Sorten.

Außerdem sei $plcs_S$:$T \times T_1 \times \ldots \times T_n \to Set_{\text{INTEGER}}$ ein Funktionssymbol in Σ und $Ax_S \subseteq For_\Sigma^{DL}$ eine Menge von geschlossenen **DL***-Formeln über der Signatur Σ.*

Das Tripel $(\Phi_S[e], plcs_S, Ax_S)$ heißt ***prädikative Beschreibung der Sequenz*** *S (bzgl. der Σ-Struktur $\mathcal{S}$ und der Variablenbelegung β) genau dann wenn*

- *$\Phi_S[e]$ ist die prädikative Beschreibung der Trägermenge M_S bzgl. $\mathcal{S}$ und β, d.h. $M_S \simeq_{\mathcal{S},\beta} \Phi_S[e]$ und*
- *für alle Σ-Strukturen $\mathcal{S}' = (M', I')$ mit $M' = M$ gilt:*

$$\begin{array}{c} \text{Für alle } w \in T \text{ gilt: } val_{I',\beta_e^w}(plcs_S(e, p_1, \ldots, p_n)) = places_S(w) \\ \textbf{gdw.} \\ \mathcal{S}' \models Ax_S, \text{ also } \mathcal{S}' \text{ alle (geschlossenen) Formeln aus } Ax_S \text{ erfüllt und} \end{array}$$

- *$\mathcal{S} \models Ax_S$*

Wir schreiben in diesem Fall $S \simeq_{\mathcal{S},\beta} (\Phi_S[e], plcs_S[e], Ax_S)$ und lassen die Indizes weg, sofern die betrachtete Struktur und Variablenbelegung aus dem Kontext klar sind. ◁

Bemerkung (Notationelle Konvention). Wir wollen im folgenden auch für prädikative Beschreibungen von Sequenzen aus Gründen der Lesbarkeit bei der Verwendung des Funktionssymbols $plcs_S$ eine Kurznotation anwenden:

Seien e die ausgezeichnete Variable aus $\Phi_S[e]$ und $p_1, \ldots, p_n$ die freien Variablen aus $\Phi_S[e]$, welche von e verschieden sind. Wir notieren dann in Termen statt der ausführlichen Angabe $plcs_S(e, p_1, \ldots, p_n)$ kurz $plcs_S[e]$. □

3.2.1.2 Grundbereichserweiterung prädikativer Darstellungen

Bevor wir zu den Anwendungen dieser Darstellungen für OCL-Kollektionen übergehen, benötigen wir zunächst noch einen technischen Begriff: Die *Grundbereichserweiterung* einer prädikativen Darstellung einer Kollektion.

Eine prädikative Darstellung einer Kollektion bezieht sich immer auf ein Universum T. Seien C_1 und C_2 zwei Kollektionen über den Universen T_1 und T_2, die durch durch prädikative Beschreibungen dargestellt sind.

Kombiniert man nun diese Kollektionen – beispielsweise durch Vereinigung – zu einer Kollektion C, so besitzt diese Kollektion im allgemeinen den kleinsten gemeinsamen Obertypen[3] als Typen. Möchte man nun auch die Kollektion C selbst durch eine geeignete Kombination der prädikativen Beschreibungen von C_1 und C_2 prädiktiv beschreiben, so ist es erforderlich, das Universum, auf dem die Prädikate zu C_1 und C_2 arbeiten (d.h. gewissermaßen die Sicht der Prädikate auf den Grundbereich) zu *erweitern* bzw. zu verallgemeinern, um die Ergebniskollektion durch ein Prädikat zu beschreiben.

[3] Man beachte die Bemerkung auf Seite 63.

Die Ausgangssituation ist also wie folgt: Gegeben sei ein prädikative Beschreibung einer Kollektion C mit der ausgezeichneten Variable $e{:}T$. Erwünscht ist nun eine prädikative Beschreibung der *gleichen* Kollektion C, die eine gegebene Variable $e'{:}T'$ eines Obertyps T' von T als ausgezeichneten Variable verwendet.

Im folgenden sei gesichert, daß die neue Variable e' in keiner Form – d.h. nicht frei und nicht gebunden – in der prädikativen Beschreibung von C vorkommt.

Für den Fall das T und T' übereinstimmen – also keine wirkliche Erweiterung des Grundbereichs vorgenommen wird – erhalten wir die neue prädikative Beschreibung durch einfache syntaktische Ersetzung der freien Vorkommen von e in dem Prädikat $\Phi_C[e]$, das die Trägermenge von C beschreibt, durch e' ersetzt. Alle anderen (möglichen) Teile der Beschreibung bleiben unverändert. Das neue Prädikat bezeichnen wir durch $\Phi_C[e']$ und die möglichen anderen „neuen" Beschreibungsbestandteile Teile durch $cnt_C[e']$ bzw. $plcs_C[e']$ und $Ax[e']$.

Eine prädikative Beschreibung für C enthält immer ein Prädikat $\Phi_C[e]$ über dem Universum T.

Wie erzeugen wir daraus nun ein Prädikat $\Phi_C[e']$ über dem Universum T'? Ein naheliegender Ansatz besteht in der einfachen syntaktische Ersetzung aller freien Vorkommen von e durch e', also der Formel

$$\Phi_C[e'] \equiv \Phi_C[e]\{e/e'\}$$

Es gibt jedoch eine *einzige* Situation, in der dieses Vorgehen *nicht* funktioniert: Enthält die Formel $\Phi_C[e]$ ein freies Vorkommen von e, auf das eine Eigenschaft (Funktion, Attribut oder Methode) angewendet wird, die nicht für ein entsprechendes Argument e' des Typs T' definiert ist, so ist eine einfache syntaktische Ersetzung nicht möglich, da keine syntaktisch zulässige Formel entsteht. Wir behelfen und in diesem Fall durch eine Formel, die eine Art *Typecast* anwendet:

$$\Phi_C[e'] \equiv \dot{\exists} e : T\,(e' \doteq e \wedge \Phi_C[e])$$

Diese Formel ist syntaktisch zulässig und beschreibt die gleiche Trägermenge. Wir wollen – sofern das möglich ist – die einfache syntaktische Ersetzung verwenden.

Wir bezeichen die so erzeugte Formel durch $\Phi_C[e']$.

Bei prädikativen Darstellungen von Multimengen und Sequenzen haben wir außerdem ein neues Funktionsymbol $cnt_B[e']$ bzw. $plcs_S[e']$ anzugeben.

Seien $p_1{:}T_1, \ldots, p_n{:}T_n$ die Parameter des Symbols $cnt_B[e]$ bzw.$plcs_S[e]$, die von $e{:}T$ verschieden sind.

Dazu verwenden wir ein neues Funktionssymbol

$$\begin{aligned} & cnt_B{:}T' \times T_1 \times \ldots \times T_n \to \textsc{Integer} \\ bzw.\quad & plcs_S{:}T' \times T_1 \times \ldots \times T_n \to Set_{\textsc{Integer}} \end{aligned}$$

Wir bezeichnen dies neue Symbol durch $cnt_B[e']$ bzw. $plcs_S[e']$.

Bleiben schließlich die Axiome Ax_C, welche das alte Funktionssymbol $cnt_B[e]$ bzw. $plcs_S[e]$ definieren, entsprechend anzupassen:

Wir definieren das neue Funktionsymbol $cnt_B[e']$ über zwei neue Axiome einfach unter Rückgriff auf das alte Symbol $cnt_B[e]$:

$$\begin{aligned} &\dot{\forall} e'{:}T' \dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n\,((\dot{\exists} e{:}T\ (e' \doteq e)) \to cnt_B[e'] \doteq cnt_B[e]) \\ &\dot{\forall} e'{:}T' \dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n\,(\neg(\dot{\exists} e{:}T\ (e' \doteq e)) \to cnt_B[e'] \doteq 0) \end{aligned}$$

Man beachte, daß wir Gebrauch von unserer Kurznotation $cnt_B[e]$ für den Term $cnt_B(e, p_1, \ldots, p_n)$ gemacht haben.

Analog verwenden wir für das Symbol $plcs_S[e']$:

$$\dot{\forall} e'{:}T' \dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, ((\dot{\exists} e{:}T \; (e' \doteq e)) \rightarrow plcs_S[e'] \doteq plcs_S[e])$$
$$\dot{\forall} e'{:}T' \dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, (\neg(\dot{\exists} e{:}T \; (e' \doteq e)) \rightarrow plcs_S[e'] \doteq emptySet_{\text{INTEGER}})$$

Die in der erweiterten Beschreibung notwendige Axiomenmenge besteht nun aus der Ax_C und diesen neu erzeugten Axiomen. Wir bezeichnen diese neue Axiomenmenge durch $Ax_C[e']$.

Insgesamt erhalten wir als erweiterte prädikative Beschreibung:

$\Phi_S[e']$	für Mengen S
$(\Phi_B[e'], cnt_B[e'], Ax_B[e'])$	für Multimengen B
$(\Phi_S[e'], plcs_S[e'], Ax_S[e'])$	für Sequenzen S

Bemerkung (Notationelle Konvention). Wenn wir also im folgenden Teil von einer prädikative Beschreibung $(\Phi_C[e], symb_C[e], Ax_C[e])$ ausgehend die Ausdrücke $\Phi_C[e']$, $symb_C[e']$, $Ax_C[e']$ (mit $e'{:}T'$) betrachten, so meinen wir jeweils die auf das Universum T' erweiterte Komponente der Beschreibung! □

3.2.2 Anwendung der prädikativen Darstellung für Mengen

In diesem Abschnitt untersuchen wir die Möglichkeit der Charakterisierung von Mengen, welche durch OCL-Ausdrücke spezifiziert sind, durch Prädikate. Wir interessieren uns insbesondere dafür, wie die prädikativen Beschreibung von Mengen, welche durch Aggregation mehrerer Kollektionen entstehen, von den prädikativen Beschreibungen dieser aggregierten Kollektionen abhängen.

Wir betrachten dazu, wie Mengen in OCL überhaupt entstehen, und wie eine prädikative Beschreibung für diese Mengen in den einzelnen Fällen generiert werden können.

Sei deshalb im folgenden D eine beliebige, feste Instanziierung von $\mathcal{D}$ und β eine beliebige, feste Variablenbelegung in D. Sei Σ die Signatur, die der Übersetzung insgesamt zur Verfügung steht. Sei $\mathcal{S}_D$ eine beliebige zu D korrespondierende Σ-Struktur und β_L eine zu β korrespondierende Variablenbelegung in $\mathcal{S}_D$. Alle in diesem Abschnitt betrachteten prädikativen Beschreibungen beziehen sich auf die Struktur $\mathcal{S}_D$ und die Variablenbelegung β_L, wir verzichten daher auf die explizite Erwähnung in den einzelnen Fällen.

Im folgenden bezeichnen `s1` und `s2` OCL-Ausdrücke aus $OCLExp_{\mathcal{D}}$ mit dem Ergebnistyp `Set(T1)` und `Set(T2)`. Die Auswertung eines OCL-Ausdrucks `e` bezüglich des Snapshots D von $\mathcal{D}$ und der Variablenbelegung β in D wollen wir durch $\langle \texttt{e} \rangle_{D,\beta}$ angeben.

Seien S_1 bzw. S_2 Mengen von Elementen aus $\mathcal{S}_D$, die zu den OCL-Mengen $\langle \texttt{s1} \rangle_{D,\beta}$ bzw. $\langle \texttt{s2} \rangle_{D,\beta}$ korrepsondieren. Für die S_i gelte: $S_i \simeq \Phi_i[e_i]$, d.h. die Mengen S_i seien durch die Prädikate $\Phi_i[e_i]$ (bzgl. $\mathcal{S}_D$ und β_L) beschrieben.

o bezeichne eine Instanz des OCL-Typs T'. c bezeichne eine Kollektion C über dem Elementtyp T'', wobei gelte, daß $C \simeq \Phi_C[e]$ (falls C eine Menge ist) bzw. $C \simeq (\Phi_C[e], cnt_C[e], Ax_C)$ (falls C eine Multimenge ist) oder $C \simeq (\Phi_C[e], plcs_C[e], Ax_C)$ (falls C eine Sequenz ist).

Entstehung von Mengen in OCL. In OCL gibt es folgende Möglichkeiten, eine Menge S zu erzeugen:

1. **Explizite Definition.** Mengen können in OCL über eine explizite Definition erzeugt werden. Dabei werden die Elemente der Menge explizit angegeben.

   Eine solche Definition hat die Form:

   `Set{e-1, e-2,...,e-n}` oder `Set{e-lower .. e-upper}`.

   Im ersten Fall bezeichnen die OCL-Ausdrücke `e-i` die entsprechenden Elemente, wohingegen im zweiten Fall alle Elemente aus dem angegebenen Intervall die Menge bilden.[4]

   Im ersten Fall können durch die OCL-Ausdrücke Elemente *beliebiger* OCL-Typen bezeichnet werden. Kommt dabei ein OCL-Ausdruck vor, der ein Element eines Kollektionstypen bezeichnet, so muß ein *Flattening* der Ergebnismenge vorgenommen werden, da OCL keine geschachtelten Kollektionen kennt.

   Seien nun OBdA. die Typen von `e-1`, ...,`e-k` keine Kollektionstypen und $\lceil$`e-1`$\rceil, \ldots, \lceil$`e-k`$\rceil$ die Terme, die die zugehörigen Elemente bezeichnen. Die Typen von `e-(k+1)`, ...,`e-n` entsprechen hingegen jeweils einem Kollektionstypen und $\Phi_{k+1}[e_{k+1}], \ldots, \Phi_n[e_n]$ seien die Prädikate, die die zugehörigen Kollektionen beschreiben. Sei außerdem e' eine neue Variable der Sorte T', die dem OCL-Typ zugeordnet ist, der den speziellsten mit allen Elementtypen kompatiblen Typen[5] darstellt.

   Dann läßt sich die Ergebnismenge $\langle$`Set{e-1, e-2,...,e-n}`$\rangle_{D,\beta}$ mit Hilfe des folgenden Prädikats $\Phi[e']$ (bzgl. $\mathcal{S}_D$ und β_L) beschreiben:

   $$\Phi[e'] \equiv \; e' \doteq \lceil\texttt{e-1}\rceil \vee \ldots \vee e' \doteq \lceil\texttt{e-k}\rceil \vee \Phi_{k+1}[e'] \vee \ldots \vee \Phi_n[e']$$

   Im zweiten Fall können durch die OCL-Ausdrücke Integer-Werte als Grenzen des Intervalls angegeben werden.

   Sei $\lceil$`e-lower`$\rceil$ bzw. $\lceil$`e-upper`$\rceil$ der Term, der das Element beschreibt, welches durch `e-lower` bzw. `e-upper` bezeichnet ist und e eine neue Variable der Sorte INTEGER.

   Die Ergebnismenge $\langle$`Set{e-lower .. e-upper}`$\rangle_{D,\beta}$ läßt sich dann durch folgendes Prädikat $\Phi[e]$ (bzgl. $\mathcal{S}_D$ und β_L) beschreiben:

   $$\Phi[e] \equiv \lceil\texttt{e-lower}\rceil \leq e \wedge e \leq \lceil\texttt{e-upper}\rceil$$

   Beispiel 45
   Die Menge S, die durch den OCL-Ausdruck `Set {1, 2, 8}` beschrieben wird, kann durch das Prädikat $\Phi[e] \equiv e \doteq 1 \vee e \doteq 2 \vee e \doteq 8$ dargestellt werden[6].

[4] Die OCL-Grammatik in [Obj99a] erlaubt tatsächlich nur diese beiden Formen. Eine Mischform ist demnach in OCL nicht zulässig, obwohl sicherlich denkbar. Unser Ansatz läßt sich aber auch auf eine solche Mischform übertragen.

[5] Man beachte die Bemerkung auf Seite 63.

[6] Wir haben hier der besseren Lesbarkeit wegen den Term über dem ADT INTEGER, der die ganze Zahl n darstellt, wieder durch n bezeichnet.

2. **Navigation über eine Assoziation mit einer Multiplizität größer als eins.**

   Seien a eine binäre Assoziation zwischen den Klassen T_0 und T_1 aus $\mathcal{D}$ und r_i $(i = 0, 1)$ die entsprechend der in Abschnitt 2.3.2.1 beschriebenen Formalisierung von UML-Modellen eingeführten Funktionssymbole zur Navigation nach T_i über die Assoziation a. Der OCL-Ausdruck `o` bezeichne die Instanz, von der aus navigiert werden soll.

   Sei $e{:}T_i$ eine neue Variable.

   Dann läßt sich die Ergebnismenge der Navigation von `o` aus nach T_i durch das Prädikat $\Phi[e]$ (bzgl. $\mathcal{S}_D$ und β_L) beschreiben mit:

$$\Phi[e] \equiv e \in r_i(\lceil \texttt{o} \rceil)$$

   Bemerkung . Würde ein Relationssymbol r für die formale Darstellung der Assoziation verfügbar sein, so könnte man auch das Prädikat

$$\Phi[e] \equiv r(\lceil \texttt{o} \rceil, e)$$

   verwenden. □

3. **Operationen auf Mengen.** Die Operationen auf Mengen, welche aus Mengen wieder Mengen erzeugen, werden im nächsten Abschnitt ausführlich behandelt.

4. **Operation `asSet` auf einer Multimenge.** Wir betrachten den OCL-Ausdruck `b->asSet`, der die durch `b` beschriebene Multimenge $B = \langle \texttt{b} \rangle_{D,\beta}$ in eine Menge umwandelt.

   B werde durch die prädikative Darstellung $(\Phi_B[e], cnt_B[e], Ax_B)$ (bzgl. $\mathcal{S}_D$ und β_L) charakterisiert.

   Dann läßt sich die Ergebnismenge $\langle \texttt{b->asSet} \rangle_{D,\beta}$ durch das Prädikat

$$\Phi[e] \equiv \Phi_B[e]$$

   (bzgl. $\mathcal{S}_D$ und β_L) darstellen.

5. **Operation `asSet` auf einer Sequenz.** Wir betrachten den OCL-Ausdruck `s->asSet`, der die durch `s` beschriebene Sequenz $S = \langle \texttt{s} \rangle_{D,\beta}$ in eine Menge umwandelt.

   S werde durch die $(\Phi_S[e], plcs_S[e], Ax_S)$ prädikativ beschrieben.

   Dann läßt sich die Ergebnismenge $\langle \texttt{s->asSet} \rangle_{D,\beta}$ durch das Prädikat

$$\Phi[e] \equiv \Phi_S[e]$$

   (bzgl. $\mathcal{S}_D$ und β_L) darstellen.

6. **Operation `allInstances` auf einem OCL-Typobjekt** Bezeichne `oclType` ein Element des OCL-Typs OclType, für das die Operation `allInstances` definiert ist.

Dann läßt sich die durch `oclType->allInstances` bezeichnete Menge S durch das Prädikat $\Phi[e]$ (bzgl. $\mathcal{S}_D$ und β_L) bezeichnen mit:

$$\Phi[e] \equiv true$$

Man beachte, daß e eine neue Variable der Sorte T ist, die dem OCL-Typen, welcher durch `oclType` bezeichnet wird, zugeordnet ist.

Operationen auf Mengen, die Mengen erzeugen.

1. **Vereinigung.** $\langle$`s1->union(s2)`$\rangle_{D,\beta}$ kann durch das Prädikat $\Phi[e]$ (bzgl. $\mathcal{S}_D$ und β_L) beschrieben werden mit:

   $$\Phi[e] \equiv \Phi_1[e] \vee \Phi_2[e]$$

   wobei T der kleinste gemeinsame Obertyp von T1 und T2 und e:T eine neue Variable der entsprechenden Sorte ist.

2. **Durchschnitt.** $\langle$`s1->intersection(s2)`$\rangle_{D,\beta}$ kann durch das Prädikat $\Phi[e]$ (bzgl. $\mathcal{S}_D$ und β_L) beschrieben werden mit:

   $$\Phi[e] \equiv \Phi_1[e] \wedge \Phi_2[e]$$

   wobei T der kleinste gemeinsame Obertyp von T1 und T2 und e:T eine neue Variable der entsprechenden Sorte ist.

3. **Differenz.** $\langle$`s1 - s2`$\rangle_{D,\beta}$ kann durch das Prädikat $\Phi[e]$ (bzgl. $\mathcal{S}_D$ und β_L) beschrieben werden mit:

   $$\Phi[e] \equiv \Phi_1[e] \wedge \neg\Phi_2[e]$$

   wobei T der kleinste gemeinsame Obertyp von T1 und T2 und e:T eine neue Variable der entsprechenden Sorte ist.

4. **Hinzufügen eines Elements.** $\langle$`s1->including(o)`$\rangle_{D,\beta}$ kann durch das Prädikat $\Phi[e]$ (bzgl. $\mathcal{S}_D$ und β_L) beschrieben werden mit:

   $$\Phi[e] \equiv \Phi_1[e] \vee (e \doteq \lceil \mathtt{o} \rceil)$$

   wobei T der kleinste gemeinsame Obertyp von T1 und T' und e:T eine neue Variable der entsprechenden Sorte ist.

5. **Entfernen eines Elements.** $\langle$`s1->excluding(o)`$\rangle_{D,\beta}$ kann durch das Prädikat $\Phi[e]$ (bzgl. $\mathcal{S}_D$ und β_L) beschrieben werden mit:

   $$\Phi[e] \equiv \Phi_1[e] \wedge \neg(e \doteq \lceil \mathtt{o} \rceil)$$

   wobei T der kleinste gemeinsame Obertyp von T1 und T' und e:T eine neue Variable der entsprechenden Sorte ist.

6. **Symmetrische Differenz.** $\langle$`s1->symmetricDifference(s2)`$\rangle_{D,\beta}$
   kann durch das Prädikat $\Phi[e]$ (bzgl. $\mathcal{S}_D$ und β_L) beschrieben werden mit:

   $$\Phi[e] \equiv (\Phi_1[e] \leftrightarrow \neg\Phi_2[e])$$

   wobei T der kleinste gemeinsame Obertyp von T1 und T2 und e:T eine neue Variable der entsprechenden Sorte ist.

7. **Select.** $\langle$`s1->select(e:T | b)`$\rangle_{D,\beta}$ kann durch das Prädikat $\Phi[e']$ (bzgl. $\mathcal{S}_D$ und β_L) beschrieben werden mit:

$$\Phi[e'] \equiv (\Phi_1[e'] \wedge \lceil \texttt{b} \rceil\{e/e'\})$$

wobei e':T eine neue Variable ist.

8. **Reject.** $\langle$`s1->reject(e:T | b)`$\rangle_{D,\beta}$ kann durch das Prädikat $\Phi[e']$ (bzgl. $\mathcal{S}_D$ und β_L) beschrieben werden mit:

$$\Phi[e'] \equiv (\Phi_1[e'] \wedge (\neg \lceil \texttt{b} \rceil)\{e/e'\})$$

wobei e':T eine neue Variable ist.

Operationen auf Mengen, die boolesche Werte erzeugen.

1. **Gleichheit.**

`s1 = s2` kann durch die Formel Φ beschrieben werden mit:

$$\Phi \equiv \forall e : T(\Phi_1[e] \leftrightarrow \Phi_2[e])$$

wobei T der kleinste gemeinsame Obertyp von T1 und T2 und e:T eine neue Variable der entsprechenden Sorte ist.

2. **Includes.** `s1->includes(o)` kann durch die Formel Φ beschrieben werden mit:

$$\Phi \equiv \Phi_1(\lceil \texttt{o} \rceil)$$

3. **Excludes.** `s1->excludes(o)` kann durch die Formel Φ beschrieben werden mit:

$$\Phi \equiv \neg\Phi_1(\lceil \texttt{o} \rceil)$$

4. **IncludesAll.** `s1->includesAll(c)` kann durch die Formel Φ beschrieben werden mit:

$$\Phi \equiv \dot{\forall} e : T(\Phi_C[e] \rightarrow \Phi_1[e])$$

wobei T der kleinste gemeinsame Obertyp von T1 und T" und e:T eine neue Variable der entsprechenden Sorte ist.

5. **ExcludesAll.** `s1->excludesAll(c)` kann durch die Formel Φ beschrieben werden mit:

$$\Phi \equiv \dot{\forall} e : T(\Phi_C[e] \rightarrow \neg\Phi_1[e])$$

wobei T der kleinste gemeinsame Obertyp von T1 und T" und e:T eine neue Variable der entsprechenden Sorte ist.

6. **IsEmpty.** `s1->isEmpty` kann durch die Formel Φ beschrieben werden mit:

$$\Phi \equiv \dot{\forall} e : T(\neg\Phi_1[e])$$

wobei e:T eine neue Variable ist.

7. **NotEmpty.** `s1->notEmpty` kann durch die Formel Φ beschrieben werden mit:

$$\Phi \equiv \dot{\exists} e : T(\Phi_1[e])$$

wobei e:T eine neue Variable ist.

8. **Exists.** `s1->exists(e:T | b)` kann durch die Formel Φ beschrieben werden mit:
$$\Phi \equiv \dot{\exists} e' : T(\Phi_1[e'] \wedge \lceil \mathtt{b} \rceil \{e/e'\})$$
wobei e':T eine neue Variable ist.

9. **ForAll.** `s1->forAll(e:T | b)` kann durch die Formel Φ beschrieben werden mit :
$$\Phi \equiv \dot{\forall} e' : T(\Phi_1[e'] \wedge \lceil \mathtt{b} \rceil \{e/e'\})$$
wobei e':T eine neue Variable ist.

3.2.3 Anwendung der prädikativen Darstellung für Multimengen

Wir prüfen nun die Möglichkeit der Beschreibung von Multimengen, die durch OCL-Ausdrücke spezifiziert sind, durch Prädikate. Wir interessieren uns wieder insbesondere dafür, wie die prädikativen Beschreibung von Multimengen, welche durch Aggregation mehrerer Kollektionen entstehen, von den prädikativen Beschreibungen dieser aggregierten Kollektionen abhängen.

Wir betrachten dazu, wie Multimengen in OCL überhaupt entstehen, und wie eine prädikative Beschreibung für dieser Multimengen in den einzelnen Fällen generiert werden können.

Im folgenden bezeichnen `b1` und `b2` OCL-Ausdrücke aus $OCLExp_{\mathcal{D}}$ mit dem Ergebnistyp `Bag(T1)` und `Bag(T2)`. Die Auswertung eines OCL-Ausdrucks `e` bezüglich des Snapshots D von $\mathcal{D}$ und der Variablenbelegung β in D wollen wir durch $\langle \mathtt{e} \rangle_{D,\beta}$ angeben.

Seien B_1 bzw. B_2 Multimengen von Elementen aus $\mathcal{S}_D$, die zu den Multimengen $\langle \mathtt{s1} \rangle_{D,\beta}$ bzw. $\langle \mathtt{s2} \rangle_{D,\beta}$ korrepsondieren. Für B_i gelte: $B_i \simeq (\Phi_{B_i}[e_i], cnt_{B_i}[e_i], Ax_{B_i})$, d.h. die B_i seien durch die prädikativen Beschreibungen $(\Phi_{B_i}[e_i], cnt_{B_i}[e_i], Ax_{B_i})$ (bzgl. $\mathcal{S}_D$ und β_L) dargestellt.

`o` bezeichne eine Instanz des OCL-Typs `T'`. `c` bezeichne eine Kollektion C über dem Elementtyp `T''`, wobei gelte, daß $C \simeq \Phi_C[e]$ (falls C eine Menge ist) bzw. $C \simeq (\Phi_C[e], cnt_C[e], Ax_C)$ (falls C eine Multimenge ist) oder $C \simeq (\Phi_C[e], plcs_C[e], Ax_C)$ (falls C eine Sequenz ist).

Die Menge der Axiome, die bei der Behandlung des OCL-Ausrucks `exp` erzeugt werden, wollen wir im folgenden durch $Ax_{\mathtt{exp}}$ bezeichnen.

Entstehung von Multimengen in OCL. In OCL gibt es folgende Möglichkeiten, eine Multimenge B zu erzeugen:

1. **Explizite Definition.** Auch Multimengen können in OCL über eine explizite Definition erzeugt werden. Dabei werden die Elemente der Multimenge explizit angegeben. Eine solche Definition hat die Form: `Bag{e-1, e-2, ...,e-n}` . Hierbei bezeichnen die OCL-Ausdrücke `e-i` die entsprechenden Elemente der Multimenge. Die OCL-Ausdrücke können Elemente *beliebiger* OCL-Sorten bezeichnet werden. Kommt dabei ein OCL-Ausdruck vor, der ein Element eines Kollektionstypen bezeichnet, so muß auch hier wieder ein *Flattening* der Ergebnismenge vorgenommen werden.

Seien nun OBdA. die Typen von `e-1`, ...,`e-k` keine Kollektionstypen und $\lceil$`e-1`$\rceil, \ldots, \lceil$`e-k`$\rceil$ die Terme, die die zugehörigen Elemente bezeichnen. Die Typen von `exp-(k+1)`, ...,`exp-n` entsprechen hingegen einem Kollektionstypen, wobei $\langle$`exp-i`$\rangle_{D,\beta}$ durch $(\Phi_i[e_i], cnt_i[e_i], Ax_i),\ i = k+1, \ldots, n$ beschrieben sei. Sei außerdem e eine neue Variable der Sorte T', die dem OCL-Typ zugeordnet ist, der den speziellsten mit allen Elementtypen kompatiblen Typen darstellt. Seien $p_1, \ldots, p_n$ die freien Variablen aus allen `e-i`, die von den ausgezeichneten Variablen e_i verschieden sind und $T_1, \ldots, T_n$ die zugehörigen Typen.

Seien $cnt_B, cnt_1, \ldots, cnt_k{:}T \times T_1 \times \ldots \times T_n \to$ INTEGER neue Funktionssymbole.

Die Ergebnismultimenge $B = \langle$`Bag{e-1, e-2, ...,e-n}`$\rangle_{D,\beta}$ läßt sich dann durch das Tripel $(\Phi_B[e], cnt_B[e], Ax_B)$ (bzgl. $\mathcal{S}_D$ und β_L) prädikativ beschreiben:

$$\Phi_B[e] \equiv e \doteq \lceil\texttt{e-1}\rceil \vee \ldots \vee e \doteq \lceil\texttt{e-k}\rceil \vee \Phi_{k+1}[e] \vee \ldots \vee \Phi_n[e]$$

und

$$\begin{aligned} Ax_B \equiv\ & \{Cl_{\dot\forall}\,(cnt_B[e] \doteq cnt_1[e] + cnt_2[e] + \ldots + cnt_n[e])\} \cup \\ & \{Cl_{\dot\forall}\,\dot\forall e{:}T\,((e \doteq \lceil\texttt{e-i}\rceil \to cnt_i[e] \doteq 1) \wedge \\ & \qquad (\neg(e \doteq \lceil\texttt{e-i}\rceil) \to cnt_i[e] \doteq 0)) \mid i = 1, \ldots, k\} \cup \\ & Ax_{\texttt{e-1}} \cup \ldots \cup Ax_{\texttt{e-k}} \cup Ax_{k+1}[e] \cup \ldots \cup Ax_n[e] \end{aligned}$$

Bemerkung (Punktierter Allabschluß).

Wir bezeichnen mit $Cl_{\dot\forall}\,(\Phi)$ den *punktierten Allabschluß* der Formel Φ, d.h. seien $p_1, \ldots, p_n$ die freien Variablen aus Φ und $T_1, \ldots, T_n$ die entsprechenden Sorten, dann ist

$$Cl_{\dot\forall}\,(\Phi) \equiv \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n(\Phi)$$

Man beachte, daß die Formel $Cl_{\dot\forall}\,(\Phi)$ somit abgeschlossen ist.

Bei dem gerade beschriebenen Vorgehen werden die Terme $\lceil$`e-1`$\rceil$,..., $\lceil$`e-k`$\rceil$ durch das Einführen der neuen Funktionssymbole $cnt_1, \ldots cnt_k$ sozusagen implizit zu Einermengen umgewandelt. □

Bemerkung (Alternative Axiomatisierung von cnt_B). Als Alternative zur oben angegebenen Axiomatisierung des Symbols cnt_B zur Formalisierung der Anzahlfunktion der Ergebnismultimenge B könnte man auch einfach eine **DL**-Formel benutzen, die durch ein einfaches Java-Programm die entsprechende Summe bildet und durch eine Reihe von `if`-Anweisungen das Einführen der Symbole $cnt_1, \ldots cnt_k$ vermeidet. □

Der Fall einer Multimengen-Definition über ein Intervall ist einfach zu handhaben:

$B = \langle$`Bag{e-lower .. e-upper}`$\rangle_{D,\beta}$ läßt sich auf einfache Weise durch das Tripel $(\Phi_B[e], cnt_B, Ax_B)$ prädikativ beschreiben:

$$\Phi_B[e] \equiv \lceil\texttt{e-lower}\rceil \leq e \wedge e \leq \lceil\texttt{e-upper}\rceil$$

und

$$\begin{aligned} Ax_B \equiv\ & \{Cl_{\dot\forall}\,\dot\forall e{:}T\,((\Phi_B[e] \to cnt_B[e] \doteq 1) \wedge (\neg\Phi_B[e] \to cnt_B[e] \doteq 0))\} \\ & \cup Ax_{\texttt{e-lower}} \cup Ax_{\texttt{e-upper}} \end{aligned}$$

2. **Mehrfache Navigation über eine Assoziation mit einer Multiplizität größer eins.** Eine Navigation über mehrere Assoziationen (in $\mathcal{D}$) mit einer Multiplizität größer 1 entspricht einer `collect`-Operation und wird deshalb genauso wie in Punkt 7 im nächsten Abschnitt behandelt.

3. **Operationen auf Multimengen.** Die Operationen auf Multimengen, die aus Multimengen wieder Multimengen erzeugen, werden im nächsten Abschnitt ausführlich behandelt.

4. **Operation `asBag` auf einer Menge.** Sei M die Menge, die als Ergebnis der Auswertung des OCL-Ausdrucks `s` entsteht, also $M = \langle \texttt{s} \rangle_{D,\beta}$. M lasse sich durch das Prädikat $\Phi_M[e]$ darstellen.

   Seien $p_1, \ldots, p_n$ die freien Variablen aus $\Phi_M[e]$, die von e verschieden sind, und $T_1, \ldots, T_n$ die entsprechenden Sorten.

   Sei $cnt_B{:}T \times T_1 \times \ldots \times T_n \to$ INTEGER ein neues Funktionssymbol.

   Dann läßt sich $B = \langle \texttt{s->asBag} \rangle_{D,\beta}$ prädikativ beschreiben durch das Tripel $(\Phi_B[e], cnt_B[e], Ax_B)$ mit:

   $$\Phi_B[e] \equiv \Phi_M[e]$$

   und

   $$Ax_B \equiv \{Cl_{\dot\forall}\, \dot\forall e : T((\Phi_M[e] \to cnt_B[e] \doteq 1) \wedge (\neg\Phi_M[e] \to cnt_B[e] \doteq 0))\}$$

5. **Operation `asBag` auf einer Sequenz.** Sei S die Sequenz, die als Ergebnis der Auswertung des OCL-Ausdrucks `s` entsteht, also $S = \langle \texttt{s} \rangle_{D,\beta}$. Sei S durch die prädikative Beschreibung $(\Phi_S[e], plcs_S, Ax_S)$ repräsentiert.

   Seien $p_1, \ldots, p_n$ die freien Variablen aus $\Phi_S[e]$, die von e verschieden sind, und $T_1, \ldots, T_n$ die entsprechenden Sorten.

   Sei $cnt_B{:}T \times T_1 \times \ldots \times T_n \to$ INTEGER ein neues Funktionssymbol.

   Dann läßt sich $B = \langle \texttt{s->asBag} \rangle_{D,\beta}$ prädikativ beschreiben durch das Tripel $(\Phi_B[e], cnt_B, Ax_B)$ mit:

   $$\Phi_B[e] \equiv \Phi_S[e]$$

   und

   $$Ax_B \equiv \{Cl_{\dot\forall}\, \dot\forall e : T(cnt_B[e] \doteq size(plcs_S[e]))\}$$

Operationen auf Multimengen, die Multimengen erzeugen.

1. **Vereinigung.**

   Wir betrachten den Ausdruck `b1->union(b2)`.

   Sei `T` der kleinste gemeinsame Obertyp von `T1` und `T2` und $e{:}T$ eine neue Variable der entsprechenden Sorte ist.

   Seien $p_1, \ldots, p_n$ die Menge der freien Variablen aus $\Phi_{B_1}[e]$, die von e_1 verschieden sind, und der freien Variablen aus $\Phi_{B_2}[e]$, die von e_2 verschieden sind, sowie $T_1, \ldots, T_n$ die entsprechenden Sorten.

   Sei $cnt_B{:}T \times T_1 \times \ldots \times T_n \to$ INTEGER ein neues Funktionssymbol.

   $B = \langle \texttt{b1->union(b2)} \rangle_{D,\beta}$ kann durch $(\Phi_B[e], cnt_B[e], Ax_B)$ prädikativ beschrieben werden mit:

   $$\Phi_B[e] \equiv \Phi_{B_1}[e] \vee \Phi_{B_2}[e]$$

und

$$Ax_B \equiv \{Cl_{\dot\forall}\, \dot\forall e : T\,(cnt_B[e] \doteq cnt_{B_1}[e] + cnt_{B_2}[e])\} \cup Ax_{B_1}[e] \cup Ax_{B_2}[e]$$

2. **Durchschnitt.**

   Wir betrachten den Ausdruck `b1->intersection(b2)`.

   Sei T der kleinste gemeinsame Obertyp von T1 und T2 und $e{:}T$ eine neue Variable der entsprechenden Sorte ist.

   Seien $p_1, \ldots, p_n$ die Menge der freien Variablen aus $\Phi_{B_1}[e]$, die von e_1 verschieden sind, und der freien Variablen aus $\Phi_{B_2}[e]$, die von e_2 verschieden sind, sowie $T_1, \ldots, T_n$ die entsprechenden Sorten.

   Sei $cnt_B{:}T \times T_1 \times \ldots \times T_n \to$ INTEGER ein neues Funktionssymbol.

   $B = \langle$`b1->intersection(b2)`$\rangle_{D,\beta}$ kann durch $(\Phi_B[e], cnt_B[e], Ax_B)$ prädikativ beschrieben werden mit:

$$\Phi_B[e] \equiv \Phi_{B_1}[e] \wedge \Phi_{B_2}[e]$$

   und

$$Ax_B \equiv \{Cl_{\dot\forall}\, \dot\forall e : T\,(cnt_B[e] \doteq min(cnt_{B_1}[e], cnt_{B_2}[e]))\} \cup Ax_{B_1}[e] \cup Ax_{B_2}[e]$$

3. **Hinzufügen eines Elements.**

   Wir betrachten den Ausdruck `b1->including(o)`.

   Sei T der kleinste gemeinsame Obertyp von T1 und T' und $e{:}T$ eine neue Variable der entsprechenden Sorte ist.

   Seien $p_1, \ldots, p_n$ die Menge der freien Variablen aus $\Phi_{B_1}[e]$, die von e_1 verschieden sind, und der freien Variablen aus $\lceil$o$\rceil$, sowie $T_1, \ldots, T_n$ die entsprechenden Sorten.

   Sei $cnt_B{:}T \times T_1 \times \ldots \times T_n \to$ INTEGER ein neues Funktionssymbol.

   $B = \langle$`b1->including(o)`$\rangle_{D,\beta}$ kann durch $(\Phi_B[e], cnt_B[e], Ax_B)$ prädikativ beschrieben werden mit:

$$\Phi_B[e] \equiv \Phi_{B_1}[e] \vee (e \doteq \lceil \mathsf{o} \rceil)$$

   und

$$\begin{aligned} Ax_B \equiv \{Cl_{\dot\forall}\, \dot\forall e : T\,((\neg(e \doteq \lceil \mathsf{o} \rceil) \to cnt_B[e] \doteq cnt_{B_1}[e]) \wedge \\ (e \doteq \lceil \mathsf{o} \rceil \to cnt_B[e] \doteq cnt_{B_1}[e] + 1))\} \\ \cup\, Ax_{B_1}[e] \cup Ax_{\mathsf{o}} \end{aligned}$$

4. **Entfernen eines Elements.**

   Wir betrachten den Ausdruck `b1->excluding(o)`.

   Sei T der kleinste gemeinsame Obertyp von T1 und T' und $e{:}T$ eine neue Variable der entsprechenden Sorte ist.

   Seien $p_1, \ldots, p_n$ die Menge der freien Variablen aus $\Phi_{B_1}[e]$, die von e_1 verschieden sind, und der freien Variablen aus $\lceil$o$\rceil$, sowie $T_1, \ldots, T_n$ die entsprechenden Sorten.

   Sei $cnt_B{:}T \times T_1 \times \ldots \times T_n \to$ INTEGER ein neues Funktionssymbol.

$B = \langle \texttt{b1->excluding(o)} \rangle_{D,\beta}$ kann durch $(\Phi_B[e], cnt_B[e], Ax_B)$ prädikativ beschrieben werden mit:

$$\Phi_B[e] \equiv \Phi_{B_1}[e] \wedge \neg(e \doteq \lceil \texttt{o} \rceil)$$

und

$$\begin{aligned} Ax_B \equiv \{Cl_{\dot\forall}\, \dot\forall e : T\, ((\neg(e \doteq \lceil \texttt{o} \rceil) \rightarrow cnt_B[e] \doteq cnt_{B_1}[e]) \wedge \\ (e \doteq \lceil \texttt{o} \rceil \rightarrow cnt_B[e] \doteq 0))\} \\ \cup\, Ax_{B_1}[e] \cup Ax_{\texttt{o}} \end{aligned}$$

5. **Select.**

   Wir betrachten den Ausdruck `b1->select(e:T | b)`, wobei `b` ein boolescher OCL-Ausdruck ist, der durch die Formel $\lceil \texttt{b} \rceil$ beschrieben wird.

   Sei e':T eine neue Variable.

   Seien $p_1, \ldots, p_n$ die Menge der freien Variablen aus $\Phi_{B_1}[e_1]$, die von e_1 verschieden sind, und der freien Variablen aus $\lceil \texttt{b} \rceil$, die von e verschieden sind, sowie $T_1, \ldots, T_n$ die entsprechenden Sorten.

   Sei cnt_B:$T \times T_1 \times \ldots \times T_n \rightarrow$ INTEGER ein neues Funktionssymbol.

   $B = \langle \texttt{b1->select(e:T | b)} \rangle_{D,\beta}$ kann durch $(\Phi_B[e'], cnt_B[e'], Ax_B)$ prädikativ beschrieben werden mit:

$$\Phi_B[e'] \equiv \Phi_{B_1}[e'] \wedge \lceil \texttt{b} \rceil\{e/e'\}$$

   und

$$\begin{aligned} Ax_B \equiv \{Cl_{\dot\forall}\, \dot\forall e' : T\, ((\lceil \texttt{b} \rceil\{e/e'\} \rightarrow cnt_B[e'] \doteq cnt_{B_1}[e']) \wedge \\ (\neg \lceil \texttt{b} \rceil\{e/e'\} \rightarrow cnt_B[e'] \doteq 0))\} \\ \cup\, Ax_{B_1} \cup Ax_{\texttt{b}} \end{aligned}$$

6. **Reject.**

   Wir betrachten den Ausdruck `b1->reject(e:T | b)`, wobei `b` ein boolescher OCL-Ausdruck ist, der durch die Formel $\lceil \texttt{b} \rceil$ beschrieben wird.

   Sei e':T eine neue Variable.

   Seien $p_1, \ldots, p_n$ die Menge der freien Variablen aus $\Phi_{B_1}[e_1]$, die von e_1 verschieden sind, und der freien Variablen aus $\lceil \texttt{b} \rceil$, die von e verschieden sind, sowie $T_1, \ldots, T_n$ die entsprechenden Sorten.

   Sei cnt_B:$T \times T_1 \times \ldots \times T_n \rightarrow$ INTEGER ein neues Funktionssymbol.

   $B = \langle \texttt{b1->reject(e:T | b)} \rangle_{D,\beta}$ kann durch $(\Phi_B[e'], cnt_B[e'], Ax_B)$ prädikativ beschrieben werden mit:

$$\Phi_B[e'] \equiv \Phi_{B_1}[e'] \wedge \neg \lceil \texttt{b} \rceil\{e/e'\}$$

   und

$$\begin{aligned} Ax_B \equiv \{Cl_{\dot\forall}\, \dot\forall e' : T\, ((\neg \lceil \texttt{b} \rceil\{e/e'\} \rightarrow cnt_B[e'] \doteq cnt_{B_1}[e']) \wedge \\ (\lceil \texttt{b} \rceil\{e/e'\} \rightarrow cnt_B[e'] \doteq 0))\} \\ \cup\, Ax_{B_1} \cup Ax_{\texttt{b}} \end{aligned}$$

7. **Collect.**

   Sei `exp` ein OCL-Ausdruck *beliebigen* Typs T'.

   Seien $p_1, \ldots, p_n$ die Menge der freien Variablen aus $\Phi_{B_1}[e_1]$, die von e_1 verschieden sind, und der freien Variablen aus $\lceil$`exp`$\rceil$, die von e verschieden sind, sowie $T_1, \ldots, T_n$ die entsprechenden Sorten.

   Da die Multimenge $\langle$`b1->collect(e:T | exp)`$\rangle_{D,\beta}$ über die `collect`-Operation durch Vereinigung der Ergebnisse der Auswertung des Ausdrucks `exp` auf den einzelnen Elementen der Multimenge $\langle$`b1`$\rangle_{D,\beta}$ erzeugt wird und der Typ des Ausdrucks `exp` auch ein Kollektionstyp sein kann, muß bei Auswertung dieser Operation möglicherweise ein *Flattening* vorgenommen werden. Wir unterscheiden deshalb folgende Fälle:

   (a) `exp` **hat *keinen* Kollektionstyp.**

   Wollen wir $B = \langle$`b1->collect(e:T | exp)`$\rangle_{D,\beta}$ prädikativ durch ein Tripel $(\Phi_B[h], cnt_B[h], Ax_B)$ beschreiben, so ist das für die Trägermenge noch sehr einfach möglich:

$$\Phi_B[h] \equiv \dot{\exists} e' : T\,(\Phi_{B_1}[e'] \wedge h \doteq \lceil \texttt{exp} \rceil \{e/e'\})$$

   wobei e' eine neue Variable der Sorte T und h eine neue Variable der Sorte T' ist.

   Die Beschreibung der neuen Anzahlfunktion $count_B$ dagegen, ist erheblich schwieriger: Die Anzahl der Vorkommen eines Elements h des Typs T' in der Multimenge B ergibt sich durch Summation der Anzahl der Vorkommen aller vermittelnden Elemente e aus der Multimenge B_1, d.h.

$$count_B(h) = \sum_{e' \in T : e' \in B_1 \wedge h \doteq exp(e')} count_{B_1}(e')$$

   Unsere Axiomenmenge zur Definition des zugehörigen Symbols cnt_B müßte nun diese Formeln in der **DL** formalisieren, was unter alleiniger Verwendung rein prädikatenlogischer Ausdrucksmittel nicht mehr möglich ist. Wir könnten aber ein Programm P benutzten, welches über einer Schleife die angegebene Summe berechnet, zum Beispiel für den Fall von Modelltypen T das Programm $P_T(h) \equiv$

```
int sum = 0;
if (T.lastCreatedObj != null) {
    T current = T.firstObj;
    boolean finished = false;
    do {
        if(test_B(current, h, p_1, ..., p_n)) {
            sum = sum + cnt_B1[current];
        }
        finished = (current == T.lastCreatedObj);
        current = current.nextObj;
    } while (!finished)
}
```

wobei $test_B{:}T \times T' \times T_1 \times \ldots \times T_n \to \texttt{boolean}$ ein neues Funktionssymbol ist, das den Test für die Summenbildung darstellt und durch das Axiom

$$Ax_{Test} \equiv Cl_{\dot\forall}\, \dot\forall e'{:}T \dot\forall h{:}T'\, (test_B(e', h, p_1, \ldots, p_n) \doteq \texttt{true} \leftrightarrow (\Phi_{B_1}[e'] \wedge (h \doteq \lceil \texttt{exp} \rceil \{e/e'\})))$$

definiert wird.

Als Axiomenmenge zur Definition des Symbols cnt_B ergäbe sich dann

$$Ax_B \equiv \{Cl_{\dot\forall}\, \dot\forall h : T'\, \forall i : \textsc{Integer}\, (\langle P_T(h)\rangle (i \doteq \texttt{sum}) \to cnt_B[h] \doteq i)\} \cup Ax_{B_1} \cup Ax_{\texttt{exp}} \cup Ax_{Test}$$

Sollte der `collect`-Ausdruck auf einer Menge `s` operieren, statt auf einer Multimenge `b1`, so muß lediglich der Term $cnt_{B_1}[\texttt{current}]$ im Programm $P_T(h)$ durch den Ausdruck `1` ersetzt werden, sowie die Formel $\Phi_{B_1}[e']$ in Ax_{Test} durch das zur Menge korrespondierende Prädikat $\Phi_S[e']$.

Bemerkung (Terminierung von $P_T(h)$). Das Programm $P_T(h)$ terminiert, da zu jedem Zeitpunkt im System die Extension eines Modelltypen `T` nur *endlich* viele Instanzen umfassen kann. □

(b) `exp` **hat *einen* Kollektionstyp.**

In diesem Fall können wir prinzipiell wie für den ersten Fall (für Basistypen) vorgehen, müssen jedoch das Flattening beachten, d.h. wollen wir $B = \langle$`b1->collect(e:T | exp)`$\rangle_{D,\beta}$ durch ein Tripel $(\Phi_B[h], cnt_B[h], Ax_B)$ prädikativ beschrieben, so ist gilt nun für die Trägermenge

$$\Phi_B[h] \equiv \dot\exists e' : T\, (\Phi_{B_1}[e'] \wedge h \in \lceil \texttt{exp} \rceil \{e/e'\})$$

Die zu beschreibende Anzahlfunktion $count_B$ ist in diesem Fall etwas komplizierter, als im ersten Fall: Die Anzahl der Vorkommen eines Elements h des Typs T' in der Multimenge B ergibt sich durch Summation der Produkte aus der Anzahl der Vorkommen aller vermittelnden Elemente e aus der Multimenge B_1 und der Anzahl der Vorkommen des Elements h in der Auswertung des Ausdrucks `exp` bezüglich des vermittelnden Elements e, d.h. also:

Sei $C(e) = \langle \texttt{exp} \rangle_{D,\beta}$ die Kollektion, die durch Auswertung des Ausdrucks `exp` bezüglich des Kontextelements e entsteht. Betrachten wir die zugehörige mengenartige Charakterisierung, dann entspricht die Anzahlfunktion der Ergebnismultimenge B gerade

$$count_B(h) = \sum_{e' \in T : e' \in B_1 \wedge h \doteq h \in C(e')} count_{B_1}(e') * occ_{C(e')}(h)$$

wobei für die Anzahl der Vorkommen von h in der Kollektion $C(e)$ gilt[7]:

[7] Man beachte, das das Summationskriterium sichert, daß das Element h mindestens einmal in der Kollektion $C(e)$ enthalten ist.

$$occ_{C(e')}(h) = \begin{cases} 1 & \text{falls } \texttt{exp} \text{ hat Typ } \mathsf{Set(T')} \\ count_{C(e')}(h) & \text{falls } \texttt{exp} \text{ hat Typ } \mathsf{Bag(T')} \\ |places_{C(e')}(h)| & \text{falls } \texttt{exp} \text{ hat Typ } \mathsf{Sequence(T')} \end{cases}$$

Unsere Axiomenmenge zur Definition des zugehörigen Symbols cnt_B müßte nun wieder diese Formeln in der **DL** formalisieren, wozu wir genau wie im Fall 7a vorgehen und lediglich das Programm $P_T(h)$ und das Axiom Ax_{Test} für den Test geeignet modifizieren.
Wir wollen das exemplarisch für den Fall vorführen, bei dem `exp` den Typ Sequence(T') besitzt.

Es gelte $C(e) \simeq (\Phi_{C(e)}[h'], plcs_{C(e)}[h'], Ax_{C(e)})$. Dann lautet das modifizierte Testaxiom

$$Ax_{Test} \equiv Cl_{\dot\forall}\, \dot\forall e'{:}T \dot\forall h{:}T'\ (test_B(e', h, p_1, \ldots, p_n) \doteq \texttt{true} \leftrightarrow (\Phi_{B_1}[e'] \wedge \Phi_{C(e)}[h]\{e/e'\})$$

und das entsprechende angepaßte Programm $P'_T(h) \equiv$

```
int sum = 0;
if (T.lastCreatedObj != null) {
    T current = T.firstObj;
    boolean finished = false;
    do {
        if(test_B(current, h, p_1, ..., p_n)) {
            sum = sum +
            cnt_B1[current] * size((plcs_C(e)[h]){e/current});
        }
        finished = (current == T.lastCreatedObj);
        current = current.nextObj;
    } while (!finished)
}
```

Sollte der `collect`-Ausdruck auf einer Menge `s` operieren, statt auf einer Multimenge `b1`, so muß lediglich wieder im Programm $P'_T(h)$ der Term cnt_{B_1}[current] durch den Ausdruck `1` ersetzt werden, sowie die Formel $\Phi_{B_1}[e']$ im Testaxiom durch zur Menge korrespondierende Prädikat $\Phi_S[e']$.

Bemerkung (Terminierung von $P'_T(h)$). Das Programm $P'_T(h)$ terminiert, da zu jedem Zeitpunkt im System die Extension eines Modelltypen T nur *endlich* viele Instanzen umfassen kann. □

Bemerkung (Komplexität der Darstellung). Wie man sieht, ist die prädikative Behandlung der `collect`-Operation zwar prinzipiell möglich, doch recht komplex. Die entstehenden Axiome entbehren einer gewissen Eleganz und sind nicht besonders leicht zu verstehen. Auch das Arbeiten mit diesen Formeln beim Beweisen dürfte erheblich komplizierter sein, als mit rein prädikatenlogischen Formeln. Die Situation wird zusätzlich ungünstiger, wenn man sich klar macht,

daß die `collect`-Operation eine der wichtigsten Operationen in OCL ist und implizit in Form von mehrfachen Navigationen durch ein UML-Modell $\mathcal{D}$ recht häufig vorkommt!

Eine alleinige Behandlung von `collect` nach dem oben beschriebenen Verfahren erachten wir deshalb als nicht sinnvoll! □

Operationen auf Multimengen, die boolesche Werte erzeugen.
Diese Operationen verwenden ausschließlich Informationen, die durch die Trägermenge der betrachteten Multimenge verkörpert wird und können deshalb genau auf dieselbe Weise behandelt werden, wie im Falle von Mengen in Abschnitt 3.2.2.

3.2.4 Anwendung der prädikativen Darstellung für Sequenzen

Wir wollen uns in diesem Abschnitt sehr kurz halten und lediglich andeuten, daß eine rein prädikative Behandlung von Sequenzen sehr schnell an die Grenzen der Nützlichkeit stößt. Das Problem dabei liegt nicht in der Beschreibung der Trägermenge einer aus einfacheren Sequenzen aggregierten Sequenz – ihre Beschreibung stimmt mit den Beschreibungen der Trägermengen im Falle von Mengen oder Multimengen für die Operationen, die allen Kollektionsausprägungen gemein sind, überein –, sondern in der Komplexität der Axiomatisierung der Positionsfunktion der aggregierten Sequenz, die durch die sich verändernden Positionen der Elemente aus den einfacheren Sequenzen ergeben.

Seien S_1 bzw. S_2 Sequenzen von Elementen aus $\mathcal{S}_D$, die zu den OCL-Sequenzen $\langle \texttt{s1} \rangle_{D,\beta}$ bzw. $\langle \texttt{s2} \rangle_{D,\beta}$ korrepsondieren.

Für die S_i gelte: $S_i \simeq (\Phi_{S_i}[e_i], plcs_{S_i}[e_i], Ax_{S_i})$, d.h. die Sequenzen S_i seien durch die prädikativen Beschreibungen $(\Phi_{S_i}[e_i], plcs_{S_i}[e_i], Ax_{S_i})$ (bzgl. $\mathcal{S}_D$ und β_L) dargestellt.

`o` bezeichne eine Instanz des OCL-Typs `T'`. `c` bezeichne eine Kollektion C über dem Elementtyp `T''`, wobei gelte, daß $C \simeq \Phi_C[e]$ (falls C eine Menge ist) bzw. $C \simeq (\Phi_C[e], cnt_C[e], Ax_C)$ (falls C eine Multimenge ist) oder $C \simeq (\Phi_C[e], plcs_C[e], Ax_C)$ (falls C eine Sequenz ist).

Die Menge der Axiome, die bei der Behandlung des OCL-Ausrucks `exp` erzeugt werden, wollen wir im folgenden durch $Ax_{\texttt{exp}}$ bezeichnen.

union. Die Vereinigung $S = \langle \texttt{s1->union(s2)} \rangle_{D,\beta}$ zweier Sequenzen S_1, S_2 entspricht in OCL gerade der Konkatenation aus der beiden Sequenzen, wobei die Elemente aus S_1 vor den Elementen aus S_2 stehen.

Die Trägermenge läßt sich wie bei Mengen oder Multimengen durch das Prädikat

$$\Phi_S[e] \equiv \Phi_{S_1}[e] \vee \Phi_{S_2}[e]$$

formalisieren, wobei $e{:}T$ eine neue Variable der Sorte T ist, die dem kleinsten gemeinsamen Obertypen von `T1` und `T1` entspricht.

Seien $p_1, \ldots, p_n$ die freien Variablen aus $\Phi_S[e]$, die von e verschieden sind, sowie $T_1, \ldots, T_n$ die entsprechenden Sorten.

Bei der Axiomatisierung der neuen Positionsfunktion benötigen wir die Berechnung der Größe der Sequenz S_1, sowie eine Indexverschiebung bei den Positionen der Elemente aus S_2 um diese Größe. Die neue Positionsfunktion $plcs_S$ läßt sich dann durch eine elementweise Vereinigung der beiden Positionsfunktionen $plcs_{S_1}$ und $plcs_{S_2}$ berechnen.

Bei der gewählten Definition für eine prädikative Darstellung einer Sequenz ist die Berechnung der Größe einer Sequenz nicht besonders einfach möglich. Wir benötigen eine Programm P_S^{size}, welches alle existierenden Elemente e des Universums T durchgeht und die entsprechende Anzahl der Vorkommen von e in S aufsummiert.

Ein solches Programm wäre zum Beispiel $P_S^{size} \equiv$

```
int sum = 0;
if (T.lastCreatedObj != null) {
    T current = T.firstObj;
    boolean finished = false;
    do {
        sum = sum + size(plcs_S[current]);
        finished = (current == T.lastCreatedObj);
        current = current.nextObj;
    } while (!finished)
}
```

Man beachte wieder, daß die Formel $\Phi_S[e]$ die freien Variablen $p_1, \ldots, p_n$ beinhaltet und somit für jede Belegung dieser Variablen eine *andere* (feste) Menge S betrachtet wird (und wie es im zugehörigen OCL-Ausdruck mit seinen freien Variablen auch der Fall ist).

Für die Formalisierung der Größe der Sequenz S benutzen wir dann eine neues Funkitionssymbol $size_S{:}T_1 \times \ldots \times T_n \to \textsc{Integer}$ in Σ ein, das durch die folgende Axiomenmenge definiert wird

$$Ax_S^{size} \equiv \{\dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n\, (\langle P_S^{size}\rangle (size_S(p_1, \ldots, p_n) \doteq \texttt{sum}))\}$$

Für die Berechnung der Positionsverschiebungen benutzten wir ebenfalls ein neues Funktionssymbol:

Sei $moveIndex{:}Set_{\textsc{Integer}} \times \textsc{Integer} \to Set_{\textsc{Integer}}$ eine Funktion, welche die Indizes aus dem ersten Argumentwert um den zweiten Argumentwert verschiebt und diese modifizierte Indexmenge zurückliefert. Um diese Intention formal auszudrücken benutzen wir die Axiomenmenge

$$Ax_{moveIndex} \equiv \{\forall s{:}Set_{\textsc{Integer}}\, \forall i, n{:}\textsc{Integer}\, (i \in s \leftrightarrow (i+n) \in moveIndex(s, n))\}$$

Nun können wir das Symbol $plcs_S$ für die Darstellung der Positionsfunktion von S mit der folgenden Formelmenge axiomatisieren

$$Ax_S \equiv \{Cl_{\dot\forall} \dot\forall e : T\, (plcs_S[e] \doteq plcs_{S_1}[e] \cup moveIndex(plcs_{S_2}[e], size_{S_1}(p_1, \ldots, p_n)))\} \cup Ax_{S_1}[e] \cup Ax_{S_2}[e] \cup Ax_{S_1}^{size} \cup Ax_{moveIndex}$$

Damit erhalten wir insgesamt $S \simeq (\Phi_S[e], plcs_S[e], Ax_S)$.

append, prepend, including. Für die prädikative Darstellung der Sequenz $S = \langle$`s1->append(o)`$\rangle_{D,\beta}$ können wir analog zum Fall des Operators `union` vorgehen.

e:T eine neue Variable der Sorte T ist, die dem kleinsten gemeinsamen Obertypen von T1 und T' entspricht.

Wir beschreiben die Trägermenge von S durch das Prädikat

$$\Phi_S[e] \equiv \Phi_{S_1}[e] \vee (e \doteq \lceil \mathtt{o} \rceil)$$

Seien $p_1, \ldots, p_n$ die freien Variablen aus $\Phi_S[e]$, die von e verschieden sind, sowie $T_1, \ldots, T_n$ die entsprechenden Sorten.

Sei $plcs_S$:$T \times T_1 \times \ldots \times T_n \to Set_{\text{INTEGER}}$ eine neues Funktionssymbol aus Σ, welches wir durch die Axiomenmenge

$$\begin{aligned} Ax_S \equiv \{Cl_{\dot\forall} \dot\forall e : T\, ((e \doteq \lceil \mathtt{o} \rceil \to plcs_S[e] \doteq plcs_{S_1}[e] \cup \{size_{S_1}(p_1, \ldots, p_n) + 1\}) \wedge \\ (\neg(e \doteq \lceil \mathtt{o} \rceil) \to plcs_S[e] \doteq plcs_{S_1}[e]))\} \\ \cup\, Ax_{S_1}[e] \cup Ax^{size}_{S_1} \cup Ax_{\mathtt{o}} \end{aligned}$$

definieren. Dann gilt $S \simeq (\Phi_S[e], plcs_S[e], Ax_S)$.
Der Operator `including` hat in OCL für Sequenzen dieselbe Semantik wie der Operator `append` und wird deshalb genauso behandelt.

Für die Behandlung des Operators `prepend` verwenden wir lediglich eine andere Axiomenmenge

$$\begin{aligned} Ax_S \equiv \{Cl_{\dot\forall} \dot\forall e : T\, ((e \doteq \lceil \mathtt{o} \rceil \to plcs_S[e] \doteq \{1\} \cup moveIndex(plcs_{S_1}[e], 1)) \wedge \\ (\neg(e \doteq \lceil \mathtt{o} \rceil) \to plcs_S[e] \doteq moveIndex(plcs_{S_1}[e], 1)))\} \\ \cup\, Ax_{S_1} \cup Ax_{moveIndex} \cup Ax_{\mathtt{o}} \end{aligned}$$

Waren diese einfachen Operationen noch recht gut, wenn auch nicht besonders elegant zu beschreiben, so ist das zum Beispiel für die Operatoren `excluding`, `select`, `reject` oder `collect` nicht mehr so einfach – wenn überhaupt – möglich, da die Indexverschiebungen nicht mehr so homogen zu beschreiben sind, wie zum Beispiel bei `union`.

Wir erachten diese formale Darstellung durch prädikative Beschreibungen um Falle von Sequenzen daher im allgemeinen als ungeeignet, da die Axiomenmengen zu den zugehörigen Beschreibung in den meisten Fällen zu komplex und damit zu schwer zu verstehen sind.

3.2.5 Wechsel zwischen den verschiedenen Darstellungsformen für Kollektionen.

Wie wir in der obigen Behandlung der prädikativen Darstellung von OCL-Kollektionen in der **DL** sehen konnten, gibt es einige Eigenschaften[8], die sich sehr schön über die prädikative Darstellung beschreiben lassen. Andererseits gibt es andere Eigenschaften, die sich nicht besonders gut durch eine prädikative Beschreibung fassen lassen und deren Behandlung durch eine termbasierte Darstellung der Kollektion sehr viel einfacher erscheint. Diese Überlegung motiviert die Frage, ob und wie es möglich ist, zwischen den verschiedenen möglichen Beschreibungsformen für Kollektion zu wechseln.

[8]Wir meinen hier die auf den OCL-Typen definierten Operationen; diese werden in [Obj99a] als sogenannte *properties* bezeichnet.

3.2.5.1 Wechsel von einer termbasierten Darstellung in eine prädikative Darstellung

Mengen. Sei M eine Menge über dem Universum T, die durch den Term t über dem ADT Set_T repräsentiert wird. Sei e eine Variable der Sorte T, die nicht frei in t vorkommt. Dann liefert die Formel

$$\Phi_M[e] \equiv e \in t$$

eine prädikative Beschreibung der Menge M, also $M \simeq \Phi_M[e]$.

Multimengen. Sei B eine Multimenge über dem Universum T, die durch den Term t über dem ADT Bag_T repräsentiert wird. Sei e eine Variable der Sorte T, die nicht frei in t vorkommt. Seien $p_1, \ldots, p_n$ die freien Variablen aus t und $T_1, \ldots, T_n$ die entsprechenden Sorten. Dann liefert die Formel

$$\Phi_B[e] \equiv e \in t$$

eine prädikative Beschreibung der Trägermenge von B. Sei $cnt_B{:}T \times T_1 \times \ldots \times T_n \to$ INTEGER ein neues Funktionssymbol in Σ, welches zur Formalisierung der Anzahlfunktion von B benutzt werden soll. Dann axiomatisiert die Formelmenge $Ax_B \subseteq For_\Sigma^{DL}$ mit

$$Ax_B \equiv \{\dot\forall e{:}T \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n\, (cnt_b[e] \doteq count(t, e))\}$$

das neue Funktionssymbol cnt_B in gewünschter Weise.

Wir erhalten somit insgesamt $B \simeq (\Phi_B[e], cnt_B[e], Ax_B)$.

Sequenzen. Sei S eine Sequenz über dem Universum T, die durch den Term t über dem ADT $Sequence_T$ repräsentiert wird. Sei e eine Variable der Sorte T, die nicht frei in t vorkommt. Seien $p_1, \ldots, p_n$ die freien Variablen aus t und $T_1, \ldots, T_n$ die entsprechenden Sorten. Dann liefert die Formel

$$\Phi_S[e] \equiv e \in t$$

eine prädikative Beschreibung der Trägermenge von S. Sei $plcs_S{:}T \times T_1 \times \ldots \times T_n \to Set_{\text{INTEGER}}$ ein neues Funktionssymbol in Σ, welches zur Formalisierung der Positionsfunktion von S benutzt werden soll. Dann axiomatisiert die Formelmenge $Ax_S \subseteq For_\Sigma^{DL}$ mit

$$Ax_S \equiv \{\dot\forall e{:}T \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n \forall i{:}Integer\, (i \in plcs_S[e] \leftrightarrow (1 \le i \le size(t) \wedge e \doteq at(t, i)))\}$$

das neue Funktionssymbol $plcs_S$ in gewünschter Weise.

Wir erhalten somit insgesamt $S \simeq (\Phi_S[e], plcs_S[e], Ax_S)$.

3.2.5.2 Wechsel von einer prädikativen Darstellung in eine termbasierte Darstellung

Mengen. Seien M eine Menge über dem Universum T, die durch das Prädikat $\Phi_M[e]$ beschrieben wird, $v_1, \ldots, v_n$ die freien Variablen in $\Phi_M[e]$, die von e verschieden sind und T_i $(i = 1, \ldots, n)$ die entsprechenden Sorten dieser Variablen. Sei f_M eine neues n-stelliges Funktionssymbol mit den Parametersorten $T_1, \ldots, T_n$ und der Ergebnissorte

Set_T in Σ. Dann liefert der Term $t \equiv f_M(v_1, \ldots, v_n)$ zusammen mit der Axiomenmenge

$$Ax \equiv \{\dot{\forall} e{:}T \dot{\forall} v_1{:}T_1 \ldots \dot{\forall} v_n{:}T_n \, (e \in f_M(v_1, \ldots, v_n) \leftrightarrow \Phi_M[e])\} \cup Ax_M$$

eine termbasierte Beschreibung der Menge M.

Multimengen. Sei B eine Multimenge über dem Universum T, die durch das Tripel $B \simeq (\Phi_B[e], cnt_B, Ax_B)$ prädikativ beschrieben wird, $v_1, \ldots, v_n$ die freien Variablen in $\Phi_B[e]$, die von e verschieden sind und T_i $(i = 1, \ldots, n)$ die entsprechenden Sorten dieser Variablen. Sei f_B eine neues n-stelliges Funktionssymbol mit den Parametersorten $T_1, \ldots, T_n$ und der Ergebnissorte Bag_T in Σ^*. Dann liefert der Term $t \equiv f_B(v_1, \ldots, v_n)$ zusammen mit der Axiomenmenge

$$Ax \equiv \{\dot{\forall} e{:}T \dot{\forall} v_1{:}T_1 \ldots \dot{\forall} v_n{:}T_n \, (count(f_B(v_1, \ldots, v_n), e) \doteq cnt_B(e, v_1, \ldots, v_n))\} \cup Ax_B$$

eine termbasierte Beschreibung der Multimenge B.

Sequenzen. Sei S eine Sequenz über dem Universum T, die durch das Tripel $S \simeq (\Phi_S[e], plcs_S, Ax_S)$ prädikativ beschrieben wird, $v_1, \ldots, v_n$ die freien Variablen in $\Phi_S[e]$, die von e verschieden sind und T_i $(i = 1, \ldots, n)$ die entsprechenden Sorten dieser Variablen. Sei f_S eine neues n-stelliges Funktionssymbol mit den Parametersorten $T_1, \ldots, T_n$ und der Ergebnissorte $Sequence_T$ in Σ^*. Dann liefert der Term $t \equiv f_S(v_1, \ldots, v_n)$ zusammen mit der Axiomenmenge

$$Ax \equiv \{\dot{\forall} e{:}T \dot{\forall} v_1{:}T_1 \ldots \dot{\forall} v_n{:}T_n \, (e \doteq at(f_S(v_1, \ldots, v_n), i) \leftrightarrow i \in plcs_S(e, v_1, \ldots, v_n))\} \cup Ax_S$$

eine termbasierte Beschreibung der Sequenz S.

3.2.5.3 Resultat und Diskussion

Wir haben nun die Anwendbarkeit einer prädikativen Beschreibung für die verschiedenen Ausprägungen von Kollektionen in OCL untersucht und können feststellen, daß diese Technik unterschiedlich erfolgreich einsetzbar ist:

Für Mengen scheint eine solche Darstellung sehr erfolgversprechend zu sein. Es gibt lediglich einige wenige Operationen, die sich einer prädikativen Beschreibung entziehen, beispielsweise die `size`-Operation, die die Anzahl der Elemente einer Kollektion berechnet.

Allgemein kann man sagen, daß es sich dabei um Operationen handelt, die Eigenschaften der *Kollektion* an sich wiederspiegeln – wie eben die Anzahl der Elemente –, wohingegen die prädikative Behandlung für solche Operationen gut funktioniert, die lediglich mit Eigenschaften zu tun haben, welche die *einzelnen Elementen* der Kollektion betreffen, zum Beispiel die Eigenschaft, eine ganze Zahl größer als 14 zu sein.

Desweiteren kann man beobachten, daß das Konzept in weiten Teilen auch auf Multimengen übertragbar ist, wobei nicht alle Eigenschaften in ähnlich einfacher Weise, wie es für Mengen der Fall ist, behandelt werden können.

Für Sequenzen schließlich stoßen wir – zumindest bei der von uns gewählten Definition einer prädikativen Beschreibung von Sequenzen – sehr schnell an die Grenzen der Nützlichkeit dieser Technik. Wir sind sogar der Überzeugung, daß auch für andere Definitionen einer prädikativen Darstellung von Sequenzen ähnliche Phänomene auftreten würden.

Das Kernproblem, welches bei einer prädikativen Beschreibung von Multimengen und vor allem von Sequenzen zu Tage tritt, ist gerade die Notwendigkeit, *neue* Symbole und damit ebenfalls *zusätzliche* Axiome einführen zu müssen, die zur formalen Beschreibung der Veränderung von bestimmten Charakteristika der Kollektion, auf die eine Operation angewendet wurde, dienen.

Für Multimengen entspricht dieses sich verändernde Charakteristikum gerade der Anzahlfunktion, die angibt, wie oft ein Element in einer Kollektion vorkommt, wohingegen bei Sequenzen die verschiedenen Positionen dieser Vorkommen gemeint sind.

Wir wollten ja gerade durch die Anwendung einer neuen Darstellungsform für Kollektionen vermeiden, daß solche Axiome eingeführt werden, insofern ist diese Beobachtung an sich recht unerfreulich, da unsere ursprüngliches Problem durch die prädikative Darstellungstechnik *alleine* nicht gelöst wird.

Eine rein prädikative Beschreibung von Multimengen und vor allem Sequenzen scheint im allgemeinen *nicht* sinnvoll zu sein.

Doch ist deswegen unsere ganze Untersuchung wertlos?

Nun, so pessimistisch würden wir unsere Ergebnisse nicht deuten! Das diese Beschreibungstechnik sich nicht nahtlos auf alle Kollektionsarten ausdehnen läßt, war prinzipiell zu erwarten; aus den Ergebnissen aber zu schließen, daß eine solche Darstellung in *jedem* Fall ungeeignet ist, wäre hingegen ebenfalls maßlos übertrieben.

Wir müssen lediglich nach einem geeigneten – und hoffentlich recht großen – Fragment der Sprache OCL suchen, für das wir unsere Prädikate gewinnbringend einsetzen können.

Zumindest für Mengen scheint das in vielen Fällen gut möglich zu sein, möglicherweise sind jedoch in bestimmten Situationen auch für Multimengen und Sequenzen Erfolge zu erzielen.

Wie wir in den Abschnitten 3.2.5.1 sowie 3.2.5.2 dargelegt haben, kann unsere neue Darstellungsform von Kollektionen mit derjenigen kombiniert werden, die in der Basisabbildung verwendet wird. Damit ist gewährleistet, daß wir eine Optimierung für ein Fragment von OCL, die auf unserer prädikativen Darstellungstechnik beruht, problemlos an die Basisabbildung anbinden können. Die Darstellung von Kollektionen mittels Formeln braucht also nicht in allen Fällen zu funktionieren, um sie trotzdem einsetzen zu können. Dieses Ergebnis sollte uns etwas versöhnlich stimmen.

Wir werden im Abschnitt 3.3 ein solches Fragment angeben, für welches die prädikative Beschreibung sehr gut funktioniert.

3.3 Eine einfache Heuristik für die Nutzung prädikativer Darstellungen

3.3.1 Grundidee der Heuristik

Wie wir im vorangehenden Abschnitt versucht haben darzulegen, ist die Beschreibung von Multimengen und Sequenzen durch Prädikate problematisch, da häufig neue Axiome eingeführt werden müssen, was wir ja gerade vermeiden wollen.

Für Mengen hingegen sind die aus der Abbildung entstehenden Formeln oftmals besser lesbar, als bei der termbasierten Darstellung, die in der Basisabbildung verwendet wird. Man denke beispielsweise an die Behandlung des `select`-Operators.

Wir wollen daher grundsätzlich für die Übersetzung jeder Operation auf Mengen die prädikative Darstellung für die beteiligten Mengen nutzen, *sofern* die Operation das zuläßt und nicht andere gewichtige Gründe dagegen sprechen.

Können wir die Prädikate eventuell auch zur Beschreibung von Multimengen und Sequenzen einsetzen, *ohne* neue Axiome einzuführen? Vor allem im Falle von Multimengen wäre das wünschenswert, da Multimengen – neben Mengen – die am häufigsten vorkommende Kollektionsart in OCL ist und damit ähnlich gute Ergebnisse bei der prädikativen Darstellung wie bei Mengen unsere Abbildung insgesamt *erheblich* verbessern würden!

Wenn man unsere Untersuchung über die prädikative Darstellung von Multimengen und Sequenzen nocheinmal genauer anschaut, so kann man die folgende interessante Beobachtung machen:

Die neu eingeführten Axiome beschreiben immer nur die Veränderungen der Aspekte von Multimengen bzw. Sequenzen, die diese von Mengen abheben, also die Anzahl von Vorkommen eines Elements bzw. die Positionen der entsprechenden Vorkommen.

Betrachtet man jedoch wie die Veränderungen der Trägermengen in die Prädikate eingehen, so stellt man fest, daß zum einen diese Veränderungen *ohne* die Einführung neuer Axiome formal zu beschreiben sind, und zum anderen, daß diese Veränderungen der Trägermengen für *alle* drei Ausprägungen von Kollektionen in OCL die gleichen sind!

Das heißt also, wenn man nur die Trägermengen einer Multimenge bzw. einer Sequenz betrachtet, so läßt sich die prädikative Darstellungstechnik *genauso gut* anwenden, wie bei Mengen. Im Prinzip bedeutet eine solche Betrachtung ja gerade, daß man die entsprechende Multimenge oder Sequenz nur (d.h. „vergröbert") als *Menge* betrachtet und alle sonstigen Informationen, wie die Anzahl oder genauen Positionen der Vorkommen, unter den Tisch fallen läßt.

Aber ist eine solche Betrachtung von Multimengen und Sequenzen überhaupt sinnvoll? Rein intuitiv würde man so eine verkürzte Sichtweise für puren Unsinn halten, denn wozu verwendet ein Modellierer überhaupt eine Multimenge oder Sequenz, wenn er eigentlich nur eine Menge betrachtet?

Dieser Meinung stimmen wir prinzipiell zu, aber unsere Erfahrung in der Modellierung mit UML/OCL belehrt uns eines besseren:

Das angeführte Argument basiert auf der Annahme, daß ein Modellierer sich bei der Formulierung von Constraints *immer bewußt ist* (bzw. daß es einen wichtigen Grund dafür gibt), daß er an einer bestimmten Stelle im Constraint eine Multimenge oder eine Sequenz benutzt.

Diese Annahme trifft bei *vielen* Constraint jedoch gar nicht zu! Multimengen bzw. Sequenzen entstehen sehr oft *implizit* durch Navigation im UML-Modell $\mathcal{D}$, *ohne* daß sich der Modellierer dieser Tatsache bewußt ist, oder von dieser Tatsache bzw. den speziellen Eigenschaften dieser Kollektionen *wirklichen* Gebrauch im Constraint macht. Im Gegenteil, recht häufig sammelt ein Modellierer durch verschiedene Navigationen im Modell Objekte in einer Kollektion zusammen und verarbeitet diese Kollektion anschließend, in dem er bestimmte Eigenschaften der Elemente dieser Kollektion prüft. Aufgrund der Typisierung der OCL-Ausdrücke arbeitet der Modellierer dann im allgemeinen zwar mit einer Multimenge, er benutzt jedoch die Anzahl der Vorkommen eines Elements bzw. die genauen Positionen der Vorkommen möglicherweise in keiner Weise.

In einer solchen Situation verwendet der Modellierer tatsächlich nur die Trägermenge der Multimenge oder Sequenz. Unsere Hoffnungen sind also nicht völlig unberechtigt.

Ein Beispiel soll die Diskussion verdeutlichen:

Beispiel 46
Wir wollen in unserem Beispielmodell aus Abbildung 2.1 ausdrücken, daß jeder Angestellte der Firma c, der über 30 Jahre alt ist, ausschließlich für dieses Unternehmen arbeitet und formulieren dazu den Ausdruck

```
c.employees->select(e| e.age > 30)->collect(e|
   e.employer)->forAll(cmp| cmp = c)
```

Der enthaltene `collect`-Ausdruck beschreibt aufgrund des OCL- Typsystems eine Multimenge von Unternehmen. Die konkrete Anzahl der Vorkommen der Unternehmen wird hingegen vom Modellierer gar nicht verwendet, da der `forAll`-Operator auf diese Multimenge angewendet wird, um zu prüfen, ob jedes Unternehmen in der Multimenge gerade der Firma c entspricht.

Wir können also unter diesen Umständen für Multimengen und Sequenzen eine prädikative Darstellung[9] in einem beliebigen Teilausdruck des Constraints verwenden und uns *nur* auf die Trägermenge konzentrieren – alle anderen Informationen spielen keine Rolle.

Leider müssen wir bei dieser vergröberten Betrachtung noch etwas vorsichtiger und restriktiver sein, wenn wir eine prädikative Darstellung für einen Teilausdrucks aus einem solchen Constraint anwenden wollen:

Wie wir bereits festgestellt haben, ist die rein prädikative Repräsentation von Multimengen bzw. Sequenzen *nicht* sinnvoll, da sich manche Operationen auf diesen Kollektionen nicht so einfach darstellen lassen, wenn die zu verarbeitende Kollektion in Form eines Prädikats beschrieben ist. Man denke etwa an einen OCL-Ausdruck der Form `b->size`, wobei `b` eine Multimenge beschreibt, die durch das Prädikat $\Phi_b[e]$ beschrieben wurde und die `size`-Operation zu übersetzen ist.

Wir wollen in solchen Fällen weiterhin die *termbasierte* Darstellung aus unserer Basisabbildung verwenden und daher die Basisabbildung mit einer Heuristik kombinieren, die unsere Idee der Nutzung von Prädikate in den oben geschilderten Situationen implementiert.

Betrachten wir nun das folgende Beispiel, um die Problematik bei einer solchen Kombination zu veranschaulichen:

Beispiel 47
Gegeben seinen folgende OCL-Ausdrücke, die jeweils Mengen von Konten beschreiben:

$$s_1 \equiv \texttt{aBank.customers.accounts->asSet}$$
$$s_2 \equiv \texttt{aBank.employees.accounts->asSet}.$$

Wäre nun s_1 bzw. s_2 durch das Prädikat $\Phi_1(aAccount, aBank)$ bzw. $\Phi_2(aAccount, aBank)$ dargestellt, dann würde beispielsweise die Übersetzung des Durchschnitts der beiden Mengen $s_3 \equiv$

```
aBank.customers.accounts->asSet->intersection(
  aBank.employees.accounts->asSet)
```

[9] Wir verstehen in einer solchen Situation unter einer *prädikativen Darstellung* einer Multimenge bzw. Sequenz nur noch das Prädikat, welches die zugehörige Trägermenge charakterisiert!

einfach durch die Konjunktion

$$\Phi_3(aAccount, aBank) \equiv \Phi_1(aAccount, aBank) \wedge \Phi_2(aAccount, aBank)$$

der beiden Prädikate darzustellen sein.

Wird der Ausdruck s_3 jedoch im Kontext einer `size`-Operation in einem OCL-Ausdruck, beispielsweise $s_4 \equiv$

```
aBank.customers.accounts->asSet->intersection(
  aBank.employees.accounts->asSet)->size
```

benutzt, so ist die prädikative Darstellung von s_3 ungünstig. Wir müßten an dieser Stelle die prädikative Beschreibung $\Phi_3(aAccount, aBank)$ in eine termbasierte Darstellung $t_3(aBank)$ wandeln, wofür ein neuer Bezeichner (in Form eines neuen Symbols in Σ^*) und damit ein zusätzliches Axiom zur Definition dieses Bezeichners einzuführen wäre:

$$\dot{\forall} a{:}Account \dot{\forall} b{:}Bank\ (a \in t_3(b) \leftrightarrow \Phi_3(a, b))$$

Damit ergäbe sich als Übersetzung von s_4:

$$\lceil s_4 \rceil \equiv size(t_3(aBank))$$

Wie dieses Beispiel sehr schön zeigt, könnte man unter diesem Umständen schon bei der Übersetzung der Teilausdrücke s_1, s_2 auf eine termbasierte Darstellung ausweichen, um die oben notwendige Repräsentationswandlung und damit die Erzeugung des zusätzlichen Axioms zu vermeiden.

Wir müssen also gegebenenfalls zwischen den einzelnen Darstellungsformen – der prädikativen und der termbasierten – wandeln. Da wir an dieser Stelle unter einer prädikativen Darstellung einer Kollektion nur noch das Prädikat verstehen, welches die Trägermenge der Kollektion bezeichnet[10], ist – im Gegensatz zu den Ergebnissen der Abschnitte 3.2.5.1 und 3.2.5.2 – eine Wandlung nicht mehr ohne weiteres möglich:

Aus einer termbasierten Darstellung eine prädikative zu erzeugen ist nachwievor ohne weiteres möglich. Aus einem Prädikat jedoch eine termbasierte Repräsentation zu generieren, scheitert hingegen aufgrund der ausschließlichen Betrachtung der Trägermenge für Multimengen und Sequenzen gänzlich: Die entsprechende Multimenge kann *nicht* mehr exakt angegeben werden, da wir die Anzahl der Vorkommen bzw. die entsprechenden Positionen der Vorkommen aus dem Prädikat der Trägermenge alleine nicht mehr rekonstruieren können. Für Mengen ist eine solche Wandlung – wie Beispiel 47 veranschaulichte – zwar prinzipiell möglich, hat aber den Nachteil, daß wieder ein neues Symbol (und damit ein neues Axiom) für den Term, der die Menge anschließend beschreiben soll, eingeführt werden muß. Man sollte solche Wandlungen also auch für den Fall von Mengen nicht zu oft durchführen und weitestgehend vermeiden.

Für unsere Heuristik wollen wir sogar restriktiver sein und eine solche Wandlung von Prädikaten nach Termen ganz verbieten.

Die Heuristik muß also bei der Entscheidung, ob ein gewisser Teilausdruck e' des Constraints unter Verwendung von Prädikaten übersetzt werden soll, alle Teilausdrücke e des Constraints untersuchen, die den Ausdruck e' enthalten, und prüfen, ob einer dieser Vorfahren e termbasiert übersetzt werden muß und eine prädikative Abbildung von e' zu einer Wandlung des zugehörigen Prädikates in einen Term führen

[10] Andernfalls hätten wir im Vergleich zu den Resultaten aus Abschnitt 3.2.5.3 nichts gewonnen.

würde. Ist das für einen Vorfahren e der Fall, so wird e termbasiert übersetzt, ansonsten durch ein Prädikat.

Bei dieser Analyse des zu übersetzenden Constraints gilt es schließlich noch einen Sonderfall zu beachten: Der Constraint könnte einen (oder mehrere) `let`-Ausdrücke enthalten, beispielsweise `let a:T = exp in e`. Da `let`-Ausdrücke als Abkürzungen zu verstehen sind, die durch einfache syntaktische Ersetzung aufgelöst werden können, muß bei der Festlegung der Behandlung des Ausdrucks `exp`, der durch die Variable `a` im Ausdruck `e` abgekürzt wird, alle Teilausdrücke e' in e analysiert werden, die die Variable a benutzen.

Wir entscheiden uns der Einfachheit halber im folgenden dafür, einen abgekürzten Ausdruck `exp` für *alle* Vorkommen von a *gleich* zu behandeln – eine Entscheidung, die manchmal sicherlich schlechtere Ergebnisse liefert, als eine *differenzierte* Behandlung der einzelnen Vorkommen. Sollte sich das für die Anwendung der Formeln tatsächlich als erheblicher Nachteil herausstellen, so kann man die Heuristik entsprechend anpassen:

Erfordert ein einziges der Vorkommen von a eine termbasierte Darstellung von a, so wird für alle Vorkommen von a eine termbasierte Darstellung gewählt. Andernfalls kommt eine prädikative Repräsentation von a zur Anwendung.

Wir wollen diese Gedanken nun formal fassen, was insbesondere für ein detailliertes Verständnis sowie eine Implementierung dieser Ideen unabdingbar ist.

3.3.2 Formalisierung der Grundidee

Die Kernidee der Heuristik ist, die Übersetzung eines Teilausdrucks e' (eines gegebenen OCL-Ausdrucks e), der einen *beliebigen* Kollektionstypen aufweist, durch ein Prädikat darzustellen, wenn die durch e' beschriebene Kollektion im Kontext des Ausdrucks e als *Menge* verwendet wird, d.h. die Anzahl der Vorkommen eines Elementes oder die Position eines Vorkommens innerhalb der Kollektion *nicht* benötigt wird *und gleichzeitig* diese prädikative Darstellung die Übersetzung der Ausdrücke verbessert, die e' enthalten.

Aber wie können wir nun entscheiden, ob ein Kollektionsausdruck e' „als Menge" verwendet wird und für welche OCL-Operationen eine prädikative Darstellung der Übersetzung eines Argumentausdrucks Vorteile bringt?

Um diese informell beschriebenen Kriterien formal zu präzisieren, wollen wir zunächst den Begriff des Pfadausdrucks einführen:

Definition 22 (Pfadausdruck)
Ein **Pfadausdruck** p *ist ein OCL-Ausdruck, der in der OCL-Grammatik G aus dem Nichterminalsymbol* `postfixExpression` *erzeugt wird, also*

$$p \in \mathcal{L}_{PathExpressions} := \{w \in OCLExp_D \mid \texttt{postfixExpression} \vdash_G^* w\}$$

◁

Pfadausdrücke sind somit alle OCL-Ausdrücke, die Instanzen oder Kollektionen von Instanzen eines *beliebigen* OCL-Typs (oder Eigenschaften von diesen) beschreiben.

Im allgemeinen besteht ein Pfadausdruck p =`e?f1?...?fN`[11] aus einer Reihe von Schritten `fk`, wobei `e` das Startelement des Pfadausdrucks beschreibt – in der Regel wird das das Kontextelement `self` oder ein Literal sein – und jeder Schritt `fk` die Anwendung einer Eigenschaft[12] (des entsprechenden OCL Typs) auf die Instanz oder

[11] Das Symbol `?` steht dabei für das Symbol `.` oder `->`.
[12] In OCL Terminologie spricht man von einem *feature call*.

Kollektion darstellt, die durch den Pfadausdruck `e?f1?...?fk-1` beschrieben wird.

Dabei erzeugt jeder Schritt aus einer Instanz oder einer Kollektion wieder eine Instanz oder Kollektion eines beliebigen OCL-Typs, zum Beispiel durch Navigation über eine Assoziation im UML-Modell $\mathcal{D}$ oder die Anwendung einer Filters mittels `select`.

Man kann die Menge der Pfadausdrücke in gewissem Sinne als *atomare Bausteine* für den Aufbau beliebiger OCL-Ausdrücken betrachten, da diese im wesentlichen durch eine Mengen von Pfadausdrücken Eigenschaften von Kollektionen oder einzelnen Instanzen aus dem Snapshot D beschreiben und diese Eigenschaften durch Aggregation der Pfadausdrücke zu einem Constraint verarbeiten.

Insbesondere wird *jede* Kollektion in OCL durch einen Pfadausdruck beschrieben. Da wir für diese Kollektionen eine neue Darstellung (durch Formeln) erzeugen wollen, liegt es nahe, Pfadausdrücke für die formale Definition unserer Heuristik zu verwenden.

Wie oben dargelegt wurde, wirkt die Struktur eines OCL-Ausdrucks auf die Übersetzung aller Teilsausdrücke ein. Wir wollen das nun etwas formaler Beschreiben:

Definition 23 (Für eine prädikative Übersetzung sprechen)
Ein Ausdruck e **spricht für eine prädikative Übersetzung** *des Pfadausdrucks p, genau dann wenn p ein Teilausdruck von e ist und die Form $e \equiv e'\Delta e''$ hat, wobei $\Delta \in \{=, <>, -\}$ und e', e'' einem Mengentypen angehören.* ◁

Wir drücken mit dieser Definition lediglich aus, daß für die Übersetzung der Operatoren `=`, `<>`, `-` auf Mengen, es vorteilhaft wäre, daß die zugehörigen Argumentterme durch Prädikate beschrieben wären. Man könnte auch sagen, daß der Ausdruck e „ein Argument" für die prädikative Darstellung der Übersetzung von p darstellt.

Nun wollen wir durch den Begriff *kritischer Operator* alle OCL-Operatoren charakterisieren, die eine prädikative Beschreibung unmöglich machen:

Definition 24 (Kritischer Operator)
Ein Eigenschaft eines OCL-Kollektionstyps heißt **kritischer Operator**, *wenn sie* `at`, `first`, `last`, `subSequence` *oder* `iterate` *entspricht.* ◁

Kritische Operatoren sind damit alle Eigenschaften von OCL-Kollektionen, die unbedingten Gebrauch von der Anzahl oder der Reihenfolge der Vorkommen der Elemente in einer Kollektion machen.

Wie wir aber oben gleichermaßen erläutert haben, gibt es Ausdrücke, für die eine termbasierte Darstellung der Argumente unbedingt erforderlich ist. Wir definieren daher:

Definition 25 (Eine termbasierte Übersetzung fordern)
Ein Pfadausdruck p_1 **fordert eine termbasierte Übersetzung** *für den Pfadausdruck p_2, genau dann wenn p_2 ein Teilausdruck von p_1 ist, p_2 einen Kollektionstypen besitzt und zudem der Typ von p_1 nicht* Boolean *und kein Kollektionstyp ist oder p_1 einen Schritt enthält, der einen kritischen Operator anwendet.* ◁

Eine prädikative Darstellung für p_1 wäre in diesem Falle nicht möglich und damit wäre eine prädikative Darstellung für p_2 ungünstig oder sogar unzulässig.

Da in OCL jede Kollektion durch einen Pfadausdruck (oder ein Präfix davon) beschrieben wird und wir für Kollektionen entscheiden wollen, wann wir sie durch Prädikate oder durch einen Term beschreiben, müssen wir die syntaktische Struktur des Pfadausdrucks und die einzelnen auftretenden Typen analysieren:

Eine prädikative Darstellung für ein Vorkommen eines Pfadausdrucks p ist in auf jeden Fall dann sinnvoll, wenn der p ein boolescher OCL-Ausdruck ist, der keinen Schritt enthält, welcher einen kritischen Operator anwendet.

Falls p einen Kollektionstypen besitzt, dann hängt die geeignete Darstellung von dem Kontext ab, in dem der Pfadausdruck p steht: Gibt es einen Elternausdruck p' der für eine prädikative Darstellung von p spricht und für den gilt, daß kein Teilausdruck p'' von p', der das betrachtete Vorkommen von p enthält, eine termbasierte Übersetzung fordert, dann ist eine prädikative Darstellung für dieses Vorkommen von p nützlich. Auf jeden Fall darf p auch hier keinen Schritt enthalten, der einen kritischen Operator anwendet, damit eine prädikative Darstellung für p benutzt werden kann.

Man beachte an diesem Vorgehen, daß sich das Vermeiden einer Darstellungswandlung von Prädikaten zu Termen in einer gewissen Dominanz der Forderungen nach termbasierter Behandlung von p (durch einen Teilausdruck p'') gegenüber dem Argument für eine prädikative Darstellung von p (durch einen übergeordneten Teilausdruck p') niederschlagen.

Insgesamt ergibt sich damit die folgende formale Definition unserer Heuristik:

Definition 26 (Heuristik *Kollektionen als Mengen*)
Sei e der OCL-Ausdruck, der insgesamt zu übersetzten ist.

Ein Vorkommen[13] *eines Pfadausdrucks p (in e), der aus dem Startelement s und den Schritten $f_1 \ldots f_N$ besteht, wird prädikativ übersetzt, wenn die folgenden Bedingungen erfüllt sind:*

1. *Jeder Schritt $f_1 \ldots f_N$ im Pfadausdruck p enthält keine Anwendung eines kritischen Operators, d.h.* `at`, `first`, `last`, `subSequence` *oder* `iterate`, *und*

2. *Der Pfadausdruck p hat den OCL-Typen* Boolean *oder – sofern p eine Kollektion beschreibt – es gibt einen Teilausdruck e' von e, der dieses Vorkommen von p enthält und für eine prädikative Übersetzung von p spricht und zusätzlich gibt es keinen Teilausdruck e'' von e', der dieses Vorkommen von p enthält und eine termbasierte Übersetzung von p fordert.*

In allen anderen Fällen wird die Standard-Darstellung der Basisabbildung bei die Übersetzung verwendet.

Für die Entscheidung der Behandlung von Termen, die durch `let`*-Ausdrücke abgekürzt werden, muß die Definition des Begriffs* „ist Teilausdruck von" *etwas verallgemeinert werden:*

Sei `let a:T = exp in e` *ein* `let`*-Ausdruck im zu übersetzenden Constraint. Dann betrachtet man den abgekürzten Term* `exp` *als Teilausdruck aller Terme, die die zugehörige Abkürzung* `a` *enthalten. Das entspricht prinzipiell der Betrachtung als syntaktische Ersetzung!*

[13] Man beachte, daß *verschiedene* Vorkommen desselben Pfadausdrucks p *unterschiedlich* übersetzt werden können.

Momentan gilt dabei die folgende vereinfachende Einschränkung: Alle Vorkommen von `a` *müssen nach folgenden Prinzip gleich behandelt werden: Erfordert ein einziges der Vorkommen von a eine termbasierte Darstellung von a, so wird für alle Vorkommen von a eine termbasierte Darstellung gewählt. Andernfalls kommt eine prädikative Repräsentation von a zur Anwendung.* ◁

3.4 Ergebnisse der Anwendung der Heuristik

Wir werden nun versuchen, die Wirkungsweise unserer Heuristik an einem komplexeren Anwendungsbeispiel zu demonstrieren und mit den Ergebnissen der reinen Basisabbildung zu vergleichen.

Dazu verwenden wir die Constraints aus Abschnitt 2.4 und übersetzen sie diesmal mit einer optimierten Variante der Basisabbildung, die gerade die in Abschnitt 3.3 beschriebene Heuristik *Kollektionen als Mengen* verwendet.

Bei der Darstellung folgen wir dem Schema aus Abschnitt 2.4, d.h. wir geben für jedes Beispiel zuerst eine natürlichesprachliche Anforderung an die modellierte Miniwelt an, die anschließend in Form eines OCL-Constraints formalisiert wird. Danach geben wir die Ergebnisse der Übersetzung durch die optimierte Variante an. Um den direkten Vergleich mit der reinen Basisabbildung zu erleichtern, folgt schließlich die Formel, die durch die reine Basisabbildung generiert wird und bereits in 2.4 angegeben wurde.

Die Übersetzung wird hier ebenfalls nicht wie bisher unter Verwendung von mathematischen Symbolen aufgeschrieben, sondern über eine rein textuelle Darstellung der erzeugten Formeln festgehalten, die von einer Implementierung beispielsweise auf dem Bildschirm ausgegeben würde. Auch an dieser Stelle haben wir haben zur Generierung der Ergebnisse unsere eigene Implementierung herangezogen. Zur besseren Lesbarkeit wurde der Text formatiert und die Variablenbezeichnungen vereinfacht. Die punktierten Quantoren $\dot{\forall}, \dot{\exists}$ wurden noch nicht durch eine entsprechende Bereinigung expandiert und sind durch `all'` bzw. `ex'` notiert. Wir verzichten an dieser Stelle außerdem auf die Darstellung der freien Variablen des Constraints durch Programmvariable. Für die Junktoren $\wedge, \vee, \rightarrow, \leftrightarrow$ schreiben wir im folgenden wieder `&`,`|`,`->` bzw. `<->`. Die Enthaltensein-Relation $\in$ wird durch das Symbol `contains` verkörpert. Die weiteren Korrespondenzen zwischen den Symbolen in der bisherigen Notation und der rein textuellen Darstellung sollten im einzelnen ebenfalls ohne Probleme klar werden.

3.4.1 Invarianten

Beispiel 48
Natürlichsprachliche Anforderung:

> Ein Prüfer wird von keinem Referenten geprüft, der von diesem Prüfer bewertet wurde.

OCL-Constraint C:

```
context Referent inv:
  self.pruefling->notEmpty implies
    not Referent.allInstances->exists(r|
        r.pruefling->includes(self) and
        self.pruefling->includes(r))
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:Referent.(
 ex' r0:Referent.contains(self.pruefling,r0) ->
   !ex' r:Referent.(
       contains(r.pruefling,self) &
       contains(self.pruefling,r)
   )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:

```
all' self:Referent.(
  all' r1:Referent.contains(allInstancesOfReferent,r1) ->
    (ex' r0:Referent.contains(self.pruefling,r0) ->
       !ex' r:Referent.(
           contains(allInstancesOfReferent,r) &
           contains(r.pruefling,self) &
           contains(self.prruefling,r)
        )
    )
)
```

Beispiel 49
Natürlichsprachliche Anforderung:

> Die Prüfungsergebnisse eines Prüflings müssen sich mit der Zeit immer weiter verbessern, oder, falls es 30 Punkte oder mehr beträgt, nicht mehr unter 30 Punkte fallen.

OCL-Constraint C:

```
context Referent inv:
  self.pruefungsErgebnis[pruefer]->forAll(pe1,pe2|
      pe1.datum.liegtVor(pe2.datum) implies
         (pe1.bewertung < pe2.bewertung or
          pe2.bewertung >= 30))
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:Referent.(
  all' pe1:PruefungsErgebnis.(
     contains(self.pruefungsErgebnis[pruefer],pe1) ->
        all' pe2:PruefungsErgebnis.(
          contains(self.pruefungsErgebnis[pruefer],pe2) ->
             pe1.datum.liegtVor(pe2.datum) ->
                ( pe1.bewertung < pe2.bewertung |
                  pe2.bewertung >= #30
                )
        )
  )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:

```
all' self:Referent.(
  all' self:Referent.let-0(self) =
         pruefungsErgebnis[pruefer](self) ->
    all' pe1:PruefungsErgebnis.(
      contains(let-0(self),pe1) ->
         all' pe2:PruefungsErgebnis.(
           contains(let-0(self),pe2) ->
              pe1.datum.liegtVor(pe2.datum) ->
                ( pe2.bewertung < pe1.bewertung |
                  pe2.bewertung >= #30
                )
         )
    )
)
```

Beispiel 50
Natürlichsprachliche Anforderung:

Ein Referent hält ausschließlich Vorträge über seine Fachgebieten.

OCL-Constraint C:

```
context Referent inv:
  self.vortragsEreignis->collect(ve| ve.vortrag)->forAll(v|
    self.fachgebiete->intersection(v.fachgebiete)->notEmpty)
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:Referent.(
   all' v:Vortrag.(
      ex' ve:VortragsEreignis.(
        contains(self.vortragsEreignis,ve) &
        v = ve.vortrag
      )  -> ex' zk:Zeichenkette.(
              contains(self.fachgebiete,zk) &
              contains(v.fachgebiete,zk)
            )
   )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:

```
all' self:Referent.(
 (  all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.
      collect-0(insert(s,ve)) =
          insert(collect-0(remove(s,ve)),ve.vortrag)
  & collect-0(Set_Of_VortragsEreignis::emptySet) =
          Bag_Of_Vortrag::emptyBag  ) ->
      all' v:Vortrag.(
        contains(collect-0(self.vortragsEreignis),v) ->
          ex' zk:Zeichenkette.
            contains(
              intersection(self.fachgebiete,v.fachgebiete)),zk)
)
```

Beispiel 51
Natürlichsprachliche Anforderung:

> Die von einem Vortragsverzeichnis verwalteten Vorträge sind bezüglich der Vortragsqualität absteigend geordnet.

OCL-Constraint C:

```
context VortragsVerzeichnis inv:
 Sequence{1 .. self.vortrag->size}->forAll(i,j| j >= i
  implies
    self.vortrag->at(i).qualitaet() >=
    self.vortrag->at(j).qualitaet())
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:VortragsVerzeichnis.(
  all' i:Integer.(
    (#1 <= i) & (i <= size(self.vortrag)) ->
        all j:Integer.(
          (#1 <= j) & (j <= size(self.vortrag)) ->
            (j >= i) ->
                ( at(self.vortrag,i).qualitaet() >=
                  at(self.vortrag,j).qualitaet() )
           )
  )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:

```
all' self:VortragsVerzeichnis.(
  (
    ( all' self:VortragsVerzeichnis.all i:Integer,j:Integer.(
       (#1 <= i) & (i <= size(sequence-1(self))) &
       (#1 <= j) & (j <= size(sequence-1(self))) & (j >= i) ->
           at(sequence-1(self),i) <= at(sequence-1(self),j))
    ) & (
      all' self:VortragsVerzeichnis.all i:Integer.(
        count(sequence-1(self),i) <= #1)
    ) & (
      all' self:VortragsVerzeichnis.(let-0(self) = sequence-1(self))
    ) & (
      all i:Integer.all' self:VortragsVerzeichnis.(
        contains(sequence-1(self),i)
        <-> ((#1 <= i) & (i <= size(self.vortrag)))
      )
    )
   ) -> all i:Integer.(
          contains(let-0(self),i) ->
           all j:Integer.(
             contains(let-0(self),j) ->
               (j >= i) ->
                ( at(self.vortrag,i).qualitaet() >=
                  at(self.vortrag,j).qualitaet() )
           )
   )
)
```

Beispiel 52
Natürlichsprachliche Anforderung:

> Alle Vorträge werden von dem gleichen Vortragsverzeichnis verwaltet und dieses Verzeichnis entspricht genau dem Verzeichnis, welches durch die statische Variable `DAS_VERZEICHNIS` in VortragsVerzeichnis zugreifbar ist.

OCL-Constraint C:

```
context VortragsVerzeichnis inv:
 let alleVerz:Set(VortragsVerzeichnis) =
       Vortrag.allInstances->collect(v|
                        v.vortragsVerzeichnis)->asSet in
   alleVerz = Set{VortragsVerzeichnis.DAS_VERZEICHNIS}
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:VortragsVerzeichnis.(
  all' vv:VortragsVerzeichnis.(
     ex' v:Vortrag.(vv = v.vortragsVerzeichnis)
     <-> (vv = VortragsVerzeichnis.DAS_VERZEICHNIS)
  )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:

```
(
   (
     all' v:Vortrag.contains(allInstancesOfVortrag,v)
   ) & (
     all s:Set_Of_Vortrag.all' v:Vortrag.
        collect-1(insert(s,v)) =
           insert(collect-1(remove(s,v)),v.vortragsVerzeichnis)
      & collect-1(Set_Of_Vortrag::emptySet) =
           Bag_Of_VortragsVerzeichnis::emptyBag
   ) & (
     all' vv:VortragsVerzeichnis.(
       contains(collect-1(allInstancesOfVortrag),vv)
       <-> contains(asSet-1,vv)
     )
  ) & (
     alleVerz = asSet-1
  )
) -> all' vv:VortragsVerzeichnis.(
        contains(alleVerz,vv)
        <-> contains(
              insert(Set_Of_VortragsVerzeichnis::emptySet,
                     VortragsVerzeichnis.DAS_VERZEICHNIS)
             ,vv)
      )
```

Beispiel 53
Natürlichsprachliche Anforderung:

> Wir wollen eine alternative OCL-Formulierung zum vorangehenden Beispiel bieten:
>
> Alle Vorträge werden von dem gleichen Vortragsverzeichnis verwaltet und dieses Verzeichnis entspricht genau dem Verzeichnis, welches durch die statische Variable `DAS VERZEICHNIS` in VortragsVerzeichnis zugreifbar ist.
>
> Man beachte, wie sich die Anwendung der `size` Operation auf `alleVerz` auf die Übersetzung dieser Abkürzung auswirkt.

OCL-Constraint C:

```
context VortragsVerzeichnis inv:
 let alleVerz:Set(VortragsVerzeichnis) =
       Vortrag.allInstances->collect(v|
                              v.vortragsVerzeichnis)->asSet in
   alleVerz->size = 1 and
   alleVerz->forAll(v|
                   v = VortragsVerzeichnis.DAS_VERZEICHNIS)
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
(
   (
     all' v:Vortrag.contains(allInstancesOfVortrag,v)
   ) & (
     all s:Set_Of_Vortrag.all' v:Vortrag.
        collect-1(insert(s,v)) =
           insert(collect-1(remove(s,v)),v.vortragsVerzeichnis)
      & collect-1(Set_Of_Vortrag::emptySet) =
           Bag_Of_VortragsVerzeichnis::emptyBag
   ) & (
     all' vv:VortragsVerzeichnis.(
       contains(collect-1(allInstancesOfVortrag),vv)
       <-> contains(asSet-1,vv)
     )
  ) & (
     alleVerz = asSet-1
  )
) -> size(alleVerz) = #1 &
     all' vv:VortragsVerzeichnis.(
        contains(alleVerz,vv) ->
            vv = VortragsVerzeichnis.DAS_VERZEICHNIS
      )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:
Die optimierte Übersetzung entspricht in diesem Fall genau der Basis-Übersetzung.

Beispiel 54
Natürlichsprachliche Anforderung:

> Alle Folien, die ein Referent überhaupt verwendet, wurden auch von diesem Referenten erstellt.

OCL-Constraint C:

```
context Referent inv:
 self.vortragsEreignis->select(ve|
     ve.oclIsKindOf(FolienVortragsEreignis))
 ->collect(ve| ve.vortrag.oclAsType(FolienVortrag))
 ->collect(fv| fv.vortragsFolien)->forAll(f|f.autor = self)
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:Referent.(
  all' f:Folie.(
     ex' fv:FolienVortrag.(
       ex' ve:VortragsEreignis.(
          contains(self.vortragsEreignis,ve) &
          ex' fve:FolienVortragsEreignis.(fve = ve) &
          fv = asType-Vortrag-FolienVortrag(ve.vortrag)
       ) &  contains(fv.vortragsFolien,f)
     ) -> f.autor = self
  )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:

```
all' self:Referent.(
 (
    collect-0(Set_Of_VortragsEreignis::emptySet) =
          Bag_Of_FolienVortrag::emptyBag
  & collect-1(Bag_Of_FolienVortrag::emptyBag) =
          Bag_Of_Folie::emptyBag
  & all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.
       collect-0(insert(s,ve)) =
          insert(collect-0(remove(s,ve)),
                 asType-Vortrag-FolienVortrag(ve.vortrag))
  & all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.(
        (ex fve:FolienVortragsEreignis.(fve = ve)) ->
            select-0(insert(s,ve)) =
               insert(select-0(s),ve)
    )
  & all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.(
        (!ex fve:FolienVortragsEreignis.(fve = ve)) ->
            select-0(insert(s,ve)) =
                select-0(s)
    )
  & select-0(Set_Of_VortragsEreignis::emptySet) =
         Set_Of_VortragsEreignis::emptySet
  & all b:Bag_Of_FolienVortrag.all' fv:FolienVortrag.(
      collect-1(insert(b,fv)) =
```

```
          union(collect-1(b),fv.vortragsFolien)
    )
  & all' fv:FolienVortrag.(
      asType-Vortrag-FolienVortrag(fv) = fv
    )
 ) -> all' f:Folie.(
        contains(
          collect-1(collect-0(select-0(self.vortragsEreignis)))
          ,f) -> f.autor = self
      )
)
```

Beispiel 55
Natürlichsprachliche Anforderung:

> Verwandte Vorträge haben mindestens zwei Schlüsselworte gemein.

OCL-Constraint C:

```
context Vortrag inv:
 self.verwandteVortraege->forAll(v|
   v.schluesselworte->intersection(
         self.schluesselworte)->size >= 2)
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:Vortrag.all' v:Vortrag.(
  contains(self.verwandteVortraege,v) ->
       size(intersection(v.schluesselworte,
                         self.schluesselworte)) >= #2
)
```

Basis-Übersetzung Th_C des OCL-Constraint:
In diesem Fall entspricht die Basisübersetzung ebenfalls genau der optimierten Abbildung.

Beispiel 56
Natürlichsprachliche Anforderung:

> Jede Vortragsreihe umfaßt mindestens zwei Folienvorträge, einen Vortrag mit dem Schlüsselwort „Eroeffnungsvortrag' und keine weitere Vortragsreihe.

OCL-Constraint C:

```
context VortragsReihe inv:
 self.vortragsEreignis->select(ve|
    ve.oclIsKindOf(FolienVortragsEreignis))->size >= 2 and
 self.vortragsEreignis->exists(ve|
    ve.vortrag.schluesselworte->exists(zk|
      zk.wert = 'Eroeffnungsvortrag')) and
 self.vortragsEreignis->select(ve|
    ve.oclIsKindOf(VortragsReihe))->isEmpty
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:VortragsReihe.(
 (   all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.(
       (ex fve:FolienVortragsEreignis.(fve = ve)) ->
          select-0(insert(s,ve)) =
              insert(select-0(s),ve)
     )
   & select-0(Set_Of_VortragsEreignis::emptySet) =
        Set_Of_VortragsEreignis::emptySet
   & all' ve:VortragsEreignis.all s:Set_Of_VortragsEreignis.(
       (!ex fve:FolienVortragsEreignis.(fve = ve)) ->
           select-0(insert(s,ve)) = select-0(s)
     )
 ) -> (
        (size(select-0(self.vortragsEreignis)) >= #2) &
        ex' ve:VortragsEreignis.(
          contains(self.vortragsEreignis,ve) &
          ex' zk:Zeichenkette.(
            contains(ve.vortrag.schluesselworte,zk) &
            zk.wert = 'Eroeffnungsvortrag'
          )
        ) &
        all' ve:VortragsEreignis.
          !( contains(self.vortragsEreignis,ve) &
             ex' vr:VortragsReihe.(vr = ve) )
      )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:

```
all' self:VortragsReihe.(
 (   all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.(
       (ex fve:FolienVortragsEreignis.(fve = ve)) ->
          select-0(insert(s,ve)) =
              insert(select-0(s),ve)
     )
   & select-0(Set_Of_VortragsEreignis::emptySet) =
        Set_Of_VortragsEreignis::emptySet
   & all' ve:VortragsEreignis.all s:Set_Of_VortragsEreignis.(
       (!ex vr:VortragsReihe.(vr = ve)) ->
           select-1(insert(s,ve)) = select-1(s)
     )
   & all' ve:VortragsEreignis.all s:Set_Of_VortragsEreignis.(
       (!ex fve:FolienVortragsEreignis.(fve = ve)) ->
           select-0(insert(s,ve)) = select-0(s)
     )
   & select-1(Set_Of_VortragsEreignis::emptySet) =
        Set_Of_VortragsEreignis::emptySet
   & all s:Set_Of_VortragsEreignis.all' ve:VortragsEreignis.(
       (ex vr:VortragsReihe.(vr = ve)) ->
          select-1(insert(s,ve)) =
             insert(select-1(s),ve)
     )
 ) -> (
```

```
        (size(select-0(self.vortragsEreignis)) >= #2) &
        ex ve:VortragsEreignis.(
          contains(self.vortragsEreignis,ve) &
          ex zk:Zeichenkette.(
            contains(ve.vortrag.schluesselworte,zk) &
            zk.wert = 'Eroeffnungsvortrag'
          )
        ) &
        all ve:VortragsEreignis.
          !contains(select-1(self.vortragsEreignis),ve)
)
```

3.4.2 Vor-/Nachbedingungen

Beispiel 57
Natürlichsprachliche Anforderung:

> Vor dem Aufruf der Methode beginnen in FolienVortragsEreignis muß gelten, daß der Status des Ereignisses „ruhend" entspricht. Nach dem Aufruf der Methode gilt das Ereignis als „begonnen" und es wird die Folie mit der Foliennummer 1 gezeigt.
>
> Man beachte die Behandlung der 0..1-Assoziation im Teilausdruck self.aktuelleFolie.

OCL-Constraint C:

```
context FolienVortragsEreignis::beginnen():void
 pre: self.status = Status.RUHEND
 post: self.status = Status.BEGONNEN
 post: self.aktuelleFolie->notEmpty and
       self.aktuelleFolie.folienInfo->exists(fi|
         fi.folienVortrag = self.vortrag and fi.position = 1)
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:FolienVortragsEreignis.(
  self.status = Status.RUHEND ->
   <{
      self.beginnen ();
   }>( (self.status = Status.BEGONNEN) &
        ex' f:Folie.(f =  self.aktuelleFolie) &
        ex' fi:FolienInfo.(
           contains(self.aktuelleFolie.folienInfo,fi) &
           fi.folienVortrag = self.vortrag &
           fi.position = #1
        )
     )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:

```
all' self:FolienVortragsEreignis.(
  self.status = Status.RUHEND ->
   <{
      self.beginnen ();
```

```
   }>( (self.status = Status.BEGONNEN) &
        ex' f:Folie.(
           contains(
             insert(Set_Of_Folie::emptySet,
                    self.aktuelleFolie)
            ,f)
        ) &
        ex' fi:FolienInfo.(
           contains(self.aktuelleFolie.folienInfo,fi) &
           fi.folienVortrag = self.vortrag &
           fi.position = #1
        )
     )
)
```

Beispiel 58
Natürlichsprachliche Anforderung:

> Die Methode `qualitaet` liefert im Falle von Folienvorträgen einen umso größeren Wert, je näher der mittlere Füllgrad der Folien des Vortrags an 0.5 liegen.

OCL-Constraint C:

```
context Vortrag::qualitaet():Integer
post:
   let avgFuellgrad(fv: FolienVortrag): Real =
       (fv.vortragsFolien->collect(f| f.fuellgrad)->sum /
        fv.vortragsFolien->size) in
   self.oclIsKindOf(FolienVortrag) implies
     FolienVortrag.allInstances->forAll(fv|
       ((avgFuellgrad(self) - 0.5).abs() <=
        (avgFuellgrad(fv) - 0.5).abs())
          implies result >= fv.qualitaet())
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' self:Vortrag.(
  true ->
   <{
      Integer result = self.qualitaet ();
   }>(
 (collect-0(Set_Of_Folie::emptySet) =
          Bag_Of_Integer::emptyBag
  & all s:Set_Of_Folie.all' f:Folie.(
       collect-0(insert(s,f)) =
          insert(collect-0(remove(s,f)),f.fuellgrad)
    )
  & all' fv:FolienVortrag.(
       avgFuellgrad(fv) =
         ( sum(collect-0(fv.vortragsFolien)) /
           size(fv.vortragsFolien) )
    )
 ) -> ex' fv:FolienVortrag.(fv = self) ->
         all' fv:FolienVortrag.(
```

```
                (abs(avgFuellgrad(self) - #0.5) <=
                (abs(avgFuellgrad(fv) - #0.5)) ) ->
                     result >= fv.qualitaet()
          )
       )
)
```

Basis-Übersetzung Th_C des OCL-Constraint:

```
all' self:Vortrag.(
  true ->
   <{
      Integer result = self.qualitaet ();
   }>(
 (collect-0(Set_Of_Folie::emptySet) =
          Bag_Of_Integer::emptyBag
  & all s:Set_Of_Folie.all' f:Folie.(
       collect-0(insert(s,f)) =
          insert(collect-0(remove(s,f)),f.fuellgrad)
    )
  & all' fv:FolienVortrag.(
       avgFuellgrad(fv) =
         ( sum(collect-0(fv.vortragsFolien)) /
           size(fv.vortragsFolien) )
    )
  & all' fv:FolienVortrag.
        contains(allInstancesOfFolienVortrag,fv)
 ) -> ex' fv:FolienVortrag.(fv = self) ->
         all' fv:FolienVortrag.(
            contains(allInstancesOfFolienVortrag,fv) ->
               ( (abs(avgFuellgrad(self) - #0.5) <=
                 (abs(avgFuellgrad(fv) - #0.5)) ) ->
                     result >= fv.qualitaet())
         )
      )
)
```

Beispiel 59
Natürlichsprachliche Anforderung:

> Nach dem Aufruf der Methode `folieEinfuegen` in FolienVortrag wurde die übergebene Folie in die Menge der Vortragsfolien als letzte Folie eingefügt, sofern die Folie nicht schon vorhanden war.
>
> Man beachte die Behandlung von `oclIsNew`, sowie dem `@pre`-Operator. Außerdem entstehen bei der termbasierten Übersetzung des `select`-Ausdrucks *zusätzliche* Parameter!

OCL-Constraint C:

```
context FolienVortrag::folieEinfuegen(f:Folie):void
 pre: self.vortragsFolien->excludes(f)
 post: FolienInfo.allInstances->select(fi|
         fi.oclIsNew and
         fi.folienVortrag = self and
         fi.vortragsFolien = f and
```

```
        fi.position = self.vortragsFolien@pre->size + 1
       )->size = 1
```

Optimierte Übersetzung Th_C^{Opt} des OCL-Constraint:

```
all' f:Folie.all' self:FolienVortrag.(
 ( !contains(self.vortragsFolien,f) &
   all' fi:FolienInfo.(fi.created@pre <-> fi.created) &
   all' fv:FolienVortrag.(
      fv.vortragsFolien@pre = fv.vortragsFolien
   )
 ) -> <{
        self.folieEinfuegen (f);
       }>(
 (
  all' self:FolienVortrag.all s:Set_Of_FolienInfo.
      all' f:Folie.all' fi:FolienInfo.(
       (
         (created(fi) & !created@pre(fi)) &
         fi.folienVortrag = self &
         fi.vortragsFolien = f &
         di.position = size(self.vortragsFolien@pre) + #1
       ) ->
          select-0(insert(s,fi),f,self) =
            insert(select-0(s,f,self),fi)
  ) &
  all' fi:FolienInfo.
      contains(Set_Of_FolienInfo::allInstancesOfFolienInfo,fi) &
  all' self:FolienVortrag.all s:Set_Of_FolienInfo.
      all' f:Folie.all' fi:FolienInfo.(
       !(
         (created(fi) & !created@pre(fi)) &
         fi.folienVortrag = self &
         fi.vortragsFolien = f &
         di.position = size(self.vortragsFolien@pre) + #1
       ) ->
          select-0(insert(s,fi),f,self) =
            select-0(s,f,self)
  ) &
  all' self:FolienVortrag.all' f:Folie.(
      select-0(Set_Of_FolienInfo::emptySet,f,self) =
           Set_Of_FolienInfo::emptySet
  )
 ) -> size(select-0(allInstancesOfFolienInfo,f,self)) = #1))
```

Basis-Übersetzung Th_C des OCL-Constraint: Durch die Anwendung der `size`-Operation auf den `select`-Ausdruck kann unsere Heuristik nicht greifen und somit entspricht die optimierte Variante genau der Basisabbildung. Wäre beispielsweise ein boolescher Operator auf den `select`-Ausdruck angewendet worden, so wäre unsere Heuristik anwendbar und würde eine starke Vereinfachung herbeiführen.

3.5 Resultat und Ausblick auf weitere Optimierungen

Das Anwendungsbeispiel bestätigt zunächst unsere Hoffnung, eine mächtige Heuristik entwickelt zu haben, die in vielen Situationen Vereinfachungen bringt: Unter den oben angeführten zwölf Beispiels-Constraints waren lediglich in drei Fällen keine Verbesserung zu erzielen, d.h. bei 75% aller Beispielsconstraints konnten einfachere Formeln erzeugt werden. Natürlich ist die obige Stichprobe zu klein, um wirklich aussagekräftige Zahlenwerte zu liefern, doch eingeschlagene Weg scheint zumindest geeignet zu sein, um eine leistungsfähige Übersetzung zu erhalten.

Die vorgestellte Heuristik *Kollektionen als Mengen* kann sicherlich noch in vielerlei Hinsicht verbessert bzw. durch neue Heuristiken ergänzt werden, um eine optimale Abbildung für die Anwendung zu erhalten.

Wir wollen im folgenden nur einige wenige Beispiele anführen und gegebenenfalls kurz erläutern:

- *Relationssymbole.*

  Es wäre beispielsweise denkbar, Relationssymbole zur formalen Darstellung von Assoziationen aus dem UML-Modell $\mathcal{D}$ zu verwenden. Diese Formalisierung wiederum ist sicherlich nicht in allen Situationen vorteilhaft, deshalb wäre zu untersuchen, wann genau diese Darstellung zu Verbesserungen in den Ergebnissen der Übersetzung führt, und eine entsprechende Heuristik zu formulieren.

- *`let`-Ausdrücke.*

  Wie schon angedeutet, werden `let`-Ausdrücke in der vorgestellten Heuristik immer gleich behandelt, was im allgemeinen zu suboptimalen Ergebnissen führen wird. Eine differenzierte Behandlung des abgekürzten Ausdrucks mit Hinblick auf die einzelnen Vorkommen wäre wünschenswert.

  Desweiteren werden `let`-Ausdrücke bei Anwendung der Basisabbildung durch entsprechende, neue Funktionssymbole let_k in Σ^* dargestellt und durch zusätzliche Axiome geeignet axiomatisiert. Ebenso könnte man diese Abkürzungen einfach wie bei einer simplen syntaktische Ersetzung behandeln, was in manchen Situationen besser sein mag, da keine zusätzlichen Symbole und Axiome für das `let`-Konstrukt an sich eingeführt werden; bei recht komplexen abgekürzten OCL-Ausdrücken könnte andererseits dieser Vorteil schnell wieder durch die größere syntaktische Ähnlichkeit zwischen den erzeugten Termen und dem ursprünglichen OCL-Ausdruck bei Verwendung der neuen Symbole let_k wett gemacht werden, da der Modellierer in der Regel ja nicht ohne Grund eine Abkürzung für einen komplexen OCL-Ausdruck in der Formulierung der gegebenen OCL-Ausdrücke verwendet und durch die Verwendung der Abkürzung wahrscheinlich den zu übersetzenden Ausdruck besser versteht; von einem ähnlich strukturierten Übersetzungsergebnis kann man dann mit hoher Wahrscheinlichkeit ähnliches erwarten. Wieder ist eine genau, anwendungsbezogene Analyse notwendig, um eine adäquate Heuristik zu entwickeln. Die entscheidende Einflußgröße wird dabei sicherlich die Komplexität (z.B. die Länge oder Schachtelungstiefe) des abgekürzten OCL-Ausdrucks sein.

- *Gleichheitsbehandlung.*

  In der Basisabbildung wird bei Vergleichen zwischen Mengen bzw. Multimengen statt eines Gleichungsterms eine allquantifizierte Formel erzeugt. Unter

bestimmten Umständen wäre beim Beweisen mit den generierten Formeln eine Darstellung mittels eines Gleichheitssymbols günstiger. Wann genau, welche Behandlung zu wählen ist, könnte wieder – nach einer anwendungsbezogenen Analyse – durch eine neue Heuristik festgelegt werden.

- *Behandlung von* `iterate`*.*

  Die in der Basisabbildung vorgestellte Übersetzung des `iterate`-Operators ist recht komplex, was mit der Mächtigkeit des Operators an sich zusammenhängt. Es ist jedoch durchaus denkbar, unter *speziellen* Umständen – d.h. beispielsweise für ganz bestimmte Berechnungsvorschriften `exp` – Vereinfachungen herbeiführen zu können. Solche Situationen könnten durch eine spezielle Heuristik für die Behandlung des `iterate`-Operators beschrieben werden.

Wir wollen an dieser Stelle nochmals betonen, daß es neben der Lesbarkeit durchaus andere Einflußfaktoren geben kann, die die Güte der Übersetzung für die konkrete Anwendung beeinflussen können. Um diesen Einflußgrößen Rechnung zu tragen, könnte man ebenfalls neue Heuristiken an die Basisabbildung anbinden. Denkbar wären zum Beispiel

- *Heuristiken, die die automatische Beweissuche in besonderer Form unterstützen.*

  Diese Heuristiken könnten vorhandenes Wissen über ein in der Anwendung benutztes Deduktionssystem verwenden, um Formeln zu generieren, mit denen dieses System möglichst effizient arbeiten kann. Das könnte zum Beispiel dadurch erreicht werden, daß die Übersetzung nicht die volle Zielsprache, z.B. Prädikatenlogik erster Stufe, verwendet, sondern in vielen Fällen einen (möglicherweise stark) eingeschränkten Teil dieser Sprache, z.B. Horn-Formeln oder ähnliches. Das Ziel wäre es also, in möglichst vielen, praktische relevanten Fällen bestimmte Sprachkonstrukte der Zielsprache zu vermeiden, um die Arbeit des Deduktionssystems effizienter zu gestalten.

  Genausogut wäre stattdessen denkbar, daß man erwartet, in Beweisen sehr oft mit Induktionen über generierte Datentypen zu arbeiten. Einen neue Heuristik könnte dann für die Generierung von Formeln sorgen, die für die Anwendung in Induktionsbeweisen gut geeignet sind.

- *Heuristiken, die für die Verwendung besonderer Ausdrucksmittel der Zielsprache sorgen.*

  Die Basisübersetzung, wie auch die vorgestellte Heuristik, arbeiten fast ausschließlich mit prädikatenlogischen Ausdrucksmitteln. Die in der Anwendung verwendete Zielsprache bietet aber möglicherweise sehr mächtige und spezifische Sprachkonstrukte an, die von der Übersetzung bisher ungenutzt blieben. Für das KeY-System mit einer Dynamischen Logik für Java als Zielsprache bestünde zum Beispiel die Möglichkeit, Programme für die Beschreibung der iterierenden Operatoren aus OCL (wie `select`, `collect` oder `iterate`) zu verwenden, was unter bestimmten Umständen zu Verbesserungen der Übersetzung führen kann. Eine entsprechende Heuristik könnte geeignetes Wissen in die Übersetzung miteinbringen.

Bemerkung (Interpretation von Heuristiken). Dieses Beispiel birgt außerdem eine *neue* Interpretation der Bedeutung von Heuristiken mit Hinblick auf das Gesamtverfahren:

Heuristiken können nun nicht nur als Mittel zur Verbesserung der Übersetzungsqualität verwendet werden, sondern können gleichermaßen zur Anpassung des Gesamtverfahrens an eine *spezielle* Zielsprache dienen. Dabei stellt die Basisabbildung durch die Nutzung einer *universellen* Sprache – der Prädikatenlogik erster Stufe – insbesondere die breite Übertragbarkeit des Kerns des Verfahrens sicher, wobei durch Anbindung neuer Heuristiken das Gesamtverfahren auf eine *spezielle* Zielsprache spezialisiert werden kann. □

Der Entwurf von Heuristiken kann ein sehr breites Spektrum von Schwierigkeitsgraden abdecken: So sind viele der oben aufgeführten Ideen sicherlich recht leicht in Form von Heuristiken zu implementieren; wie aus der exemplarischen Entwicklung einer Heuristik zur prädikativen Darstellung von Kollektionen aber auch klar geworden sein sollte, kann der Entwurf neuer, wirkungsvoller Heuristiken durchaus recht aufwendig und schwierig sein.

Man sollte sich also bewußt sein, daß die Technik der Heuristiken im allgemeinen keine Vereinfachung der Entwurfsaufgabe an sich erzielt, jedoch zu einem verständlicheren, besser wartbaren, hoch flexiblen und sehr gut anpaßbaren Verfahren führt.

Schlußendlich gilt das Prinzip der Empirie: Was für eine konkrete Anwendung wirklich benötigt wird, läßt sich im Detail schwer voraussagen und wird erst durch die empirischen Erfahrungen bei der praktischen Arbeit mit Übersetzung in der konkreten Anwendung richtig klar – kurz gesagt: Probieren geht über studieren!

Die Technik der Heuristiken bietet dafür aber eine ideale Grundlage, da eine *iterative* Entwicklung der Übersetzung für eine spezielle Anwendung sehr gut unterstützt wird.

3.6 Darstellung von `iterate` durch Programme

Bei der Entwicklung der Basisabbildung haben wir den `iterate`-Operator – angewendet auf eine Kollektion `c` – durch ein neues Funktionssymbol $iterate_E$ dargestellt, welches im wesentlichen eine $\lceil \texttt{c} \rceil$ Termdarstellung der Kollektion als Eingabe verwendete, über die zu iterieren war. Das neue Funktionssymbol $iterate_E$ wurde dabei prinzipiell durch zwei Formeln induktiv (über den Aufbau von Kollektionstermen) definiert.

Die nach diesem Verfahren entstehende Beschreibung ist daher recht komplex und schwer zu verstehen. Insbesondere wird die einem Modellierer vertraute, anschauliche Vorstellung eines schrittweisen, programmartigen Durchlaufs durch die Kollektion nicht so klar dargestellt. Dafür ist das Verfahren aber für alle hier betrachteten Logiksprachen anwendbar.

Wir wollen nun die **DL** als spezielle Zielsprache für das KeY-Projekt heranziehen, um diesem natürlichen, algorithmischen Charakter des `iterate`-Operator in direkter Form gerecht zu werden: Wir versuchen den `iterate`-Operator durch ein Programm in der **DL** zu beschreiben.

Da ein Software-Ingenieur oder Programmierer, der mit UML/OCL arbeitet, auch mit Programmiersprachen vertraut sein wird, könnte eine solche Darstellung des Operators sehr viel klarer und verständlicher sein, als die induktive Definition in der Basisabbildung.

Wie wir schon einmal angedeutet haben, erhalten wir durch ein *neues* Verfahren zur Übersetzung des `iterate`-Operators automatisch *neue, alternative* Übersetzungen für eine Fülle anderer Operatoren aus OCL, wie beispielsweise `select` oder `collect`.

Entwickelt man dann noch eine geeignete Heuristik für Anwendung dieser alternativen Verfahren, so gewinnt man schließlich für eine Übersetzung von UML/OCL insgesamt viele Spielräume damit Optionen für Optimierungen in *bestimmten* Situationen. Der Weg zu einem sehr mächtigen Übersetzungverfahren für die gewünschte Anwendung ist geebnet.

Wir werden uns aus zeitlichen Gründen im folgenden auf die Entwicklung und Diskussion einer programmartigen Beschreibung von `iterate` beschränken und auf die Formulierung einer entsprechenden Heuristik verzichten. Die Formulierung einer solchen geeigneten Heuristik ist nichttrivial und benötigt empirische Erfahrungen, die im Rahmen dieser Arbeit leider nicht mehr gewonnen werden können.

Das Ziel ist nun klar: Wir versuchen den `iterate`-Operator in *natürlicher* Weise durch ein Programm wiederzugeben.

Bevor wir jedoch den Weg zu diesem Ziel bechreiten, wollen wir uns über die Optionen klar werden, die uns für die Entwicklung einer solchen algorithmischen Beschreibung offenstehen:

Gegeben sei ein OCL-Constraint C, der einen `iterate`-Ausdruck E enthält:

$$E = \texttt{c->iterate(e:T; acc:T'= e0 | exp)}$$

Die eigentlich mit E verknüpfte und zu beschreibende algorithmische Vorschrift besteht in einer einmaligen Iteration durch die durch `c` beschriebene Kollektion. Wir werden also bezüglich dem eigentlich algorithmischen Teil wenig Spielräume haben und eine simple Schleife verwenden.

Wir erhalten somit abstrahierend betrachtet prinzipiell das folgende Programmschema $P_{iterate}$ mit den Eingabegrößen `c`, `e0` und `exp`:

```
T' acc = e0;                    // ... Initialisierung
while (c.hasMoreElements()) {   // ... Iterationsschleife
    T e = c.nextElement();
    acc = exp;              // ... Iterationschritt
}
T' result = acc
return result;                  // ... Rueckgabe
```

Diese Schablone enthält nun als offene und noch auszufüllende Lücken:

- den Zugriff auf die Elemente e in der logischen Beschreibung der Kollektion C, d.h. `c.hasMoreElements()` und `c.nextElement()`
- die Berechnungen von `e0` und `exp` in der Logik
- die Rückgabe bzw. Einbindung des Iterationsergebnis in den umgebenden Kontext, also `return result`

Diese einzelnen Punkte sind im wesentlichen unabhängig voneinander, weshalb wir sie im folgenden getrennt untersuchen wollen.

Bemerkung (Notation). Ein Programm P enthält im allgemeinen bestimmte, ausgezeichnete freie Programmvariablen $\texttt{v1}, \ldots, \texttt{vN}$ zur Darstellung der Eingabewerte und produziert einen Ergebniswert in einer Ausgabevariablen `result`. Wir wollen zur Verdeutlichung daher ein solches Programm durch $P(\texttt{v1}, \ldots, \texttt{vN}) \rightarrow \texttt{result}$ bezeichnen. Das oben angegebene Programmschema bezeichnen wir daher durch $P_{iterate}(\texttt{c}, \texttt{e0}, \texttt{exp}) \rightarrow \texttt{result}$

□

Wie greifen wir auf die Elemente während der Iteration zu? Betrachten wir eine Kollektion C in der Logik, so hängt die Antwort auf diese Frage zum einen davon ab, *welche Art* von Kollektion wir betrachten (Mengen, Multimengen oder Sequenzen), und zum anderen, *wie* die betrachtete Kollektion in der Logik dargestellt ist.

OCL definiert die Iterationsreihenfolge nur für den Fall von Sequenzen – dort entspricht sie gerade der Reihenfolge der Elemente in der Sequenz. Für Mengen und Multimengen muß jede beliebige Reihenfolge möglich sein, da ansonsten der zu übersetzende OCL-Ausdruck E undefiniert ist; wir sind dort also *frei* in der Wahl der verwendeten Iterationsfolge.

Wir haben bisher zwei grundsätzliche Darstellungen für Kollektionen in einer Logik betrachtet: Die Darstellung durch Terme über abstrakten Datentypen und die prädikative Beschreibung durch Formeln.

Für Multimengen und Sequenzen scheidet eine prädikative Darstellung deshalb für unsere Zwecke aus, da die *elementbezogene* Betrachtung (d.h. die Betrachtung eines Elements aus einem Universum T) durch diese Darstellungen *nicht gut* zu der *vorkommenbezogenen* Betrachtung (d.h. der Betrachtung eines Vorkommens eines Elements in einer Iterationsfolge) durch den `iterate`-Operator passt. D.h. wir nehmen für Multimengen und Sequenzen an, daß die Kollektion durch einen Term $\lceil \texttt{c} \rceil$ dargestellt ist[14]. Für Mengen jedoch stimmen aufgrund der beliebigen Wahl der betrachteten Iterationsfolge die *elementbezogene* und *vorkommenbezogenen* Sicht überein und beide Varianten erscheinen sinnvoll.

Kollektionsdarstellung für `c`. In diesen Fällen verwenden wir zur Iteration die von der Termdarstellung $\lceil \texttt{c} \rceil$ der Kollektion `c` induzierten Reihenfolge. Sei OBdA. $Sequence_T$ die Sorte des Kollektionsterms $\lceil \texttt{c} \rceil$.

Wir gestalten die Schleifenberechnung dann derart, daß wir eine (neue) lokale Programmvariable `remainingPart` der Sorte $Sequence_T$ verwenden, um den noch zu iterierenden Teil der Kollektion `c` zu speichern. Diese Variable wir im Initialisierungsteil durch die Anweisung $Sequence_T$ `remainingPart = c` mit dem Wert der Eingabevariablen `c` initialisiert, die die zu iterierenden Kollektion darstellt.

[14] Man beachte, daß entsprechend den Ausführungen aus Abschnitt 3.2.5 aus der prädikativen Beschreibung einer Kollektion *immer* ein solcher Term generiert werden kann, d.h. diese Annahme stellt sogar *keine* wirkliche Einschränkung dar!

Die Schleifenbedingung `c.hasMoreElements()` entspricht dann prinzipiell der Formel $\texttt{remainingPart} \doteq emptySequence_T$ und wird durch ein Funktionssymbol

$$hasMoreElements{:}Sequence_T \to \texttt{boolean}$$

dargestellt, das durch das Axiom

$$\dot{\forall} s{:}Sequence_T\, (hasMoreElements(s) \doteq \texttt{false} \leftrightarrow s \doteq emptySequence_T)$$

definiert wird.

Man beachte, daß diese Formel *nicht spezifisch* für den zu übersetzenden Ausdruck E ist und somit nur einmal generiert werden muß.

Der Iterationschritt `c.nextElement()` besteht hingegen aus zwei Teilen: Zunächst müssen wir das nächste Element in der Iterationsreihenfolge bestimmten und anschließend unseren Iterationszustand, d.h. die Variable `remainingPart`, anpassen.

Zur Ermittlung des nächsten Elements dient ein Funktionssymbol

$$nextElement{:}Sequence_T \to T$$

das durch

$$\forall s{:}Sequence_T \forall s'{:}Sequence_T \dot{\forall} e{:}T\, (s \doteq insert(s', e) \to nextElement(s) \doteq e)$$

definiert wird, also das nächste Element entsprechend der übergebenen Termdarstellung zurückgibt, wenn es ein solches gibt (und einen beliebigen Wert ansonsten).

Bemerkung (Verwendung eines Epsilon-Terms). Man könnte an dieser Stelle auch auf die explizte Verwendung eines Funktionssymbols *nextElement* verzichten und einen (punktierten) Epsilon-Term der Form

$$\dot{\epsilon} e.(\exists s'{:}Sequence_T(s \doteq insert(s', e)))$$

verwenden. Wir betrachten das dann entstehende Programm aber als weniger gut verständlich und bevorzugen daher die Nutzung eines expliziten Funktionssymbols *nextElement*. □

Für die Anpassung des Iterationszustand verwenden wir ein Funktionssymbol

$$remainingElements{:}Sequence_T \to Sequence_T$$

in Verbindung mit dem Axiom

$$\forall s{:}Sequence_T \forall s'{:}Sequence_T \dot{\forall} e{:}T\, (s \doteq insert(s', e) \to remainingElements(s) \doteq s')$$

Wir erhalten somit ein Programmschema $P_{iterate}(\texttt{b}, \texttt{e0}, \texttt{exp}) \to \texttt{result}$ der Form:

```
T' acc = e0;
Sequence_T remainingPart = c;
while (hasMoreElements(remainingPart)) {
    T e = nextElement(remainingPart);
    acc = exp;
    remaingPart = remainingElements(remainingPart);
}
T' result = acc
```

und die Axiomenmenge

$$\begin{aligned} Ax_{iterate} \equiv \{\ & \dot\forall s{:}Sequence_T\,(\ hasMoreElements(s) \doteq \texttt{false} \leftrightarrow \\ & \qquad s \doteq emptySequence_T), \\ & \forall s{:}Sequence_T \forall s'{:}Sequence_T \dot\forall e{:}T\,(\ s \doteq insert(s', e) \rightarrow \\ & \qquad nextElement(s) \doteq e), \\ & \forall s{:}Sequence_T \forall s'{:}Sequence_T \dot\forall e{:}T\,(\ s \doteq insert(s', e) \rightarrow \\ & \qquad remainingElements(s) \doteq s')\} \end{aligned}$$

Man beachte, daß die Axiomenmenge $Ax_{iterate}$ *nicht* von dem zu übersetzenden Ausdruck E abhängt.

Für Multimengen können wir völlig analog vorgehen. Im Fall von Mengen müssen wir hingegen noch bei der Aktualisierung des Iteratorzustandes beachten, daß mögliche Mehrfacheinfügungen eines Elements e in der betrachteten Termdarstellung der zu iterierenden Kollektion nur einmal betrachtet werden; wir verwenden deshalb bei der Neuberechnung des Wertes der Variablen `remainingPart` den Term

$$remove(remainingElements(\texttt{remainingPart}), \texttt{e})$$

anstelle des Terms

$$remainingElements(\texttt{remainingPart})$$

in der Zuweisung im Schleifenrumpf.

Definition 27 (Programminstanziierung)
Sei $P \equiv P(\texttt{v1}, \ldots, \texttt{vN}) \rightarrow \texttt{result}$ ein Programm mit ausgezeichenten Eingabevariablen $\texttt{v}_i$ der Sorten T_i und der Ausgabevariable `result` der Sorte T'.

Seien $p_1, \ldots, p_n$ Terme der Sorten T_i und `result'` eine Programmvariable der Sorte T'. Sei p'_i $(i = 1, \ldots, n)$ der Term, der jeweils aus dem Term p_i entsteht, indem man alle freien Vorkommen einer logische Variablen l einer Sorte T, die den Bezeichner einer lokale Variablen `l` aus P mit der Sorte T tragen, durch die entsprechende Programmvariable `l` ersetzt.

Dann verstehen wir unter der **Instanziierung von P mit $p_1, \ldots, p_n$, `result'`** *das Programm P'*

T_1 `v1 =` p'_1`;`
...
T_n `vN =` p'_n`;`
P
`T' result' = result;`

Tragen `result'` und `result` den gleichen Bezeichner, dann entfällt die letzte Programmzeile.

Wir bezeichen dieses Programm P' kurz durch $P(p_1, \ldots, p_n) \rightarrow \texttt{result'}$. ◁

Betrachten wir nun also die Terme $\lceil\texttt{c}\rceil$, $\lceil\texttt{e0}\rceil$ und $\lceil\texttt{exp}\rceil$ aus der Übersetzung der Teilausdrücke des `iterate`-Ausdrucks E, so läßt sich der durch E berechnete Wert durch das Programm

$$P_{iterate}(\lceil\texttt{c}\rceil, \lceil\texttt{e0}\rceil, \lceil\texttt{exp}\rceil) \rightarrow \texttt{result}$$

– d.h. die Instanziierung unseres Programmschemas $P_{iterate}(\texttt{c}, \texttt{e0}, \texttt{exp}) \rightarrow \texttt{result}$ mit den Termen $\lceil\texttt{c}\rceil, \lceil\texttt{e0}\rceil, \lceil\texttt{exp}\rceil$ – beschreiben.

Prädikative Darstellung für `c`. Für OCL-Ausdrücke `c` mit dem Typ T ist nicht nur eine termbasierte Beschreibung durch einen Term $\lceil \mathtt{c} \rceil$ der Sorte Set_T im Zusammenhang mit dem `iterate`-Ausdruck anwendbar, sondern – unter bestimmten Umständen – auch eine prädikative Beschreibungen der zugehörigen Menge `c` durch eine Formel $\Phi[e]$ über dem Univerum T möglich und möglichweise auch nützlich:

Ein Prädikat beschreibt eine Menge M einfach dadurch, daß die Elemente e aus dem Universum T, die in der Menge M enthalten sind, gerade die Elemente aus T sind, welche das Prädikat erfüllen.

Möchte man den `iterate`-Ausdruck E nun unter Verwendung einer solchen Beschreibung auswerten, so steht man vor dem Problem, eine (beliebige) Iterationsreihenfolge aus der Darstellung gewinnen zu müssen. Doch leider induziert die prädikative Darstellung an sich – im Gegensatz zu der termbasierten Darstellung – *keine* solche Reihenfolge. Wir brauchen also aus der **DL** selbst eine (beliebige) Ordnung über einem Universum T, um E unter Verwendung von Prädikaten überhaupt beschreiben zu können. Im allgemeinen Fall haben wir leider keine solche Ordnung, für den wichtigsten Fall von *Modelltypen* hingegen schon: Die Kontextklassen C zu einer Klassen C aus dem UML-Modell $\mathcal{D}$ bietet eine solche Ordnung in Form von Attributen zum Zugriff auf die *Liste der möglichen Objekte* der Sorte C. Insbesondere gibt es ein Anfangselement in dieser Liste, auf das man explizit zugreifen kann.

Für alle anderen Basistypen scheidet die prädikative Beschreibungsform aus, da es entweder gar keine definierte Ordnung in der **DL** gibt, oder das zugehörige Universum bzgl. einer existierenden Ordnung kein kleinstes Element besitzt.

Wir betrachten also ausschließlich den Fall von prädikativen Beschreibungen für Mengen über einem Universum T aus *Objekten*.

Sei `b` eine Programmvariable der Sorte `boolean`.

Analog zum Vorgehen im Fall der Kollektionsdarstellung der zu iterierenden Kollekton, können wir ein Programmschema $P'_{iterate}(\mathtt{b}, \mathtt{e0}, \mathtt{exp}) \rightarrow \mathtt{result}$ angeben, das – bei geeigneter Instanziierung der Größen `b`, `e0`,`exp` – den durch E berechnete Wert beschreibt:

Als Programmschema $P'_{iterate}(\mathtt{b}, \mathtt{e0}, \mathtt{exp}) \rightarrow \mathtt{result}$ benutzen wir hier:

```
T' acc = e0;
if (T.lastCreatedObj != null) {
    T e = T.firstObj;
    do {
        if(b) { // b means e is in the set
            acc = exp;
            e = e.nextObj;
        }
    } while (e == T.lastCreatedObj)
}
T' result = acc
```

Sein nun $\Phi_C[e]$ das Prädikat, das die durch `c` beschriebene Menge (über dem Modelltyp T) beschreibt. Seien $p_1, \ldots, p_n$ die Menge der freien Variablen in $\Phi_C[e]$, die sich von e unterscheiden.

Dann erzeugen wir ein neues Funktionssymbol

$$isInSet_C{:}T \times T_1, \times \ldots \times T_n \rightarrow \mathtt{boolean}$$

das gerade das Prädikat als boolescher Term beschreibt, also durch das Axiom

$$Ax_C \equiv \{\dot{\forall} e{:}T \dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \, (isInSet_C(e, p_1, \ldots, p_n) \doteq \texttt{true} \leftrightarrow \Phi_C[e])\}$$

definiert wird.

Betrachten wir nun also die Terme $\lceil$`e0`$\rceil$ und $\lceil$`exp`$\rceil$ aus der Übersetzung der Teilausdrücke des `iterate`-Ausdrucks E, so läßt sich der durch E berechnete Wert durch das Programm

$$P'_{iterate}(isInSet_C(e, p_1, \ldots, p_n), \lceil\texttt{e0}\rceil, \lceil\texttt{exp}\rceil) \rightarrow \texttt{result}$$

– d.h. die Instanziierung unseres Programmschemas $P'_{iterate}(\texttt{b}, \texttt{e0}, \texttt{exp}) \rightarrow \texttt{result}$ mit den Termen $isInSet_C(e, p_1, \ldots, p_n), \lceil\texttt{e0}\rceil, \lceil\texttt{exp}\rceil$ – und das Axiom Ax_C beschreiben.

Wir brauchen in diesem Fall also nur eine neues Funktionssymbol (und damit ein neues Axiom), das aber von der betrachteten Kollektion C abhängt. Das Pogrammschema erscheint zwar zunächst etwas komplizierter, insgesamt ist es aber einheitlicher als das Programmschema $P_{iterate}(\texttt{c}, \texttt{e0}, \texttt{exp}) \rightarrow \texttt{result}$ aus dem anderen Fall, da mehr Informationen im Programm selbst stecken und weniger Informationen in Axiomen untergebracht wurden. Beim Beweisen mit den erzeugten Formel mag das von Vorteil sein.

Man beachte desweiteren, daß sich aus einer termbasierten Darstellung einer Menge S sehr leicht eine prädikative Darstellung erzeugen läßt. Damit kann man für Mengen von Objekten das Programmschema $P'_{iterate}(\texttt{b}, \texttt{e0}, \texttt{exp}) \rightarrow \texttt{result}$ grundsätzlich immer anwenden.

Es sei noch bemerkt, daß das hier betrachtete Zusammenspiel von `iterate` und Prädikaten sehr viel Ähnlichkeit zu der semantischen Definition des des *iterate*-Operators in der sogenannten *Iterate Logic* in [Sch01a] besitzt. Diese Logiksprache enthält – im Gegensatz zu **DL** – einen expliziten Operator, der genau dem `iterate`-Operator aus OCL (auf Mengen) entspricht. Insbesondere wird dort zur Beschreibung der zu iterierenden Menge eine Formel verwendet. Die dort explizit vorhandene *lineare* Ordnung auf dem Univerum der betrachteten (endlichen) Strukturen wird hier durch die Liste der möglichen Objekte der Sorte T nachempfunden.

Wenden wir uns nun dem zweiten wesentlichen Einflußfaktor auf eine programmtechnische Darstellung von `iterate`-Ausdrücken zu:

Wie machen wir das Iterationsergebnis für andere Termen nutzbar? Der zu übersetzende `iterate`-Ausdruck E ist im allgemeinen als Teilausdruck eines anderen Ausdrucks in einen Constraint eingebettet. Wir berechen versuchen nun durch eine Programm $P_{iterate}(\lceil\texttt{c}\rceil, \lceil\texttt{e0}\rceil, \lceil\texttt{exp}\rceil) \rightarrow \texttt{result}$ den Wert des Ausdrucks E zu berechnen und in der Ausgabevariablen zu `result` zu speichern.

Es stellt sich nun noch die Frage, wie man diesen in der Ausgabevariable gespeicherten Wert einer Übersetzung eines E umfassenden Ausdrucks E' zugänglich macht.

Prinzipell gehen wir auch hier nach unserer bewährten Methode vor: Durch die Bennenungstechnik führen wir ein neues Symbol $iterate_E$ ein, daß als Parameter die Eingabewerte nimmt und den entsprechenden Ergebniswert für die Übersetzung $\lceil$`E'`$\rceil$ benennbar macht.

Seien also $p_1, \ldots, p_n$ die freien Variablen aus $\lceil$`c`$\rceil$, $\lceil$`exp`$\rceil$ und $\lceil$`e0`$\rceil$, die sich von e und acc verschieden sind, und $T_1, \ldots, T_n$ die zugehörigen Sorten.

Dann erzeugen wir ein neues Funktionssymbol $iterate_E$ in Σ^* mit der Signatur:

$$iterate_E{:}T_1 \times \ldots \times T_n \to T'$$

Wir verwenden also als Übersetzung des Ausdrucks E:

$$\lceil \texttt{E} \rceil = iterate_E(p_1, \ldots, p_n)$$

Die entsprechende Semantik des neuen Symbols $iterate_E$ als Endergebnis der Ausführung des Programms $P_{iterate}(\lceil \texttt{c} \rceil, \lceil \texttt{e0} \rceil, \lceil \texttt{exp} \rceil) \to \texttt{result}$ wird schließlich durch die folgende Axiomenmenge festgehalten:

$$\begin{aligned} Ax_E \equiv \{\ & \dot\forall p_1{:}T_1 \ldots \dot\forall p_n{:}T_n\, (\\ & \langle P_{iterate}(\lceil \texttt{c} \rceil, \lceil \texttt{e0} \rceil, \lceil \texttt{exp} \rceil) \to \texttt{result} \rangle (iterate_E(p_1, \ldots, p_n) \doteq \texttt{result})) \\ & \} \cup Ax_{iterate} \end{aligned}$$

Diese Technik ist einfach und funktioniert in jedem Fall. Außerdem läßt sich durch das neue Symbol $iterate_E$ – wie in der Basisabbildung auch – die Undefiniertheit des OCL-Ausdrucks E durch Unterspezifikation des Funktionswertes in den korrespondierenden Situationen modellieren! Die entstehende Übersetzung für den gesamten Constraint ist genauso strukturiert, wie der ursprüngliche OCL-Ausdruck, was dem Verständnis der Übersetzungen komplexer Ausdrücke zuträglich sein kann.

Wenngleich wir aus zeitlichen Gründen leider nicht mehr ausführlich darauf eingehen können, so wollen wir doch auf eine *andere Alternative* hinweisen, die in manchen Situationen möglicherweise zu Vereinfachungen führen könnte:

Da wir nun als Zielsprache **DL** betrachten und Programme somit syntaktische Bestandteile von Formeln sein können, ist hier prinzipiell auch ein anderes Vorgehen denkbar: Man verwendet zur Benennung des Ergebnis des Ausdrucks E statt des Symbols $iterate_E$ *direkt* die Ausgabevariable `result`.

Zur Verdeutlichung stellen wir uns vor, daß ein OCL-Ausdruck E', der E enthält, bisher durch einen Term $\lceil$`E'`$\rceil$ der Form

$$f_{E'}(iterate_E(p_1, \ldots, p_n))$$

beschrieben wird. Würden wir direkt die Ausgabevariable `result` statt des Funktionssymbols $iterate_E$ zur Bennenung des Wertes von E benutzen, so entstünde stattdessen eine Formel $\lceil$`E'`$\rceil$ der Form

$$\langle P_{iterate}(\lceil \texttt{c} \rceil, \lceil \texttt{e0} \rceil, \lceil \texttt{exp} \rceil) \to \texttt{result} \rangle (f(\texttt{result}))$$

und man könnte auf das Symbol $iterate_E$ und das zugehörige (komplexe) **DL**-Axiom verzichten.

Wie man aber schon an dieser groben Skizze leicht erkennt, funktioniert eine solche Vorgehensweise nicht in jedem Fall – Man muß sehr stark darauf achten, daß der notwendige *Kontext*, in dem der Ausdruck E ausgewertet wird – d.h. die Variablen $p_1, \ldots, p_n$ – nicht verfälscht werden, also das Programm gewissermaßen „an der richtigen Stelle" ausgeführt wird. Andererseits erzeugt der Diamant-Operator eine *Formel* (keinen Term!) und somit kann das Programm nur *außerhalb* von Termen ausgewertet

werden. Das heißt, daß wir im allgemeinen *nicht* immer das Programm mit der „richtigen" Belegung von $p_1, \ldots, p_n$ ausführen können, wenn ein Operator in einer Folge von OCL-Ausdrücken die Belegung eines der Parameter $p_1, \ldots, p_n$ einschränkt. Ein Beispiel:

$E' =$ `coll1->select(e' | coll2(e')->iterate(e; acc=e0 | exp))->size`

Der Kollektions-Ausdruck `coll2(e')` enthalte die durch das `select` gebundene Variable `e'`. Dann hängt der Wert des `iterate`-Ausdrucks E von der konkret betrachteten Belegung von `e'`, die aber als Iteratorvariable in ihren Werten durch den `select`-Ausdruck bestimmt wird. In diesem Fall (einer geschachtelten Konstruktion) kann daher eine *direkte* Einbindung des erzeugten Programms in der Übersetzung eines Constraints nicht funktionieren!

Es drängt sich daher die Frage auf, unter welchen Umständen eine solche *direkte* Einbindung der Übersetzung *überhaupt möglich* und wann sie außerdem *vorteilhaft* ist.

Wie im nächsten Absatz noch kurz angemerkt wird, könnte es beispielsweise für geschachtelte und hintereinandergefügte `iterate`-Ausdrücke möglich sein, die entstehenden Programme zu *einem einzigen Programm* zusammenzufügen. Bedenkt man außerdem, daß sich alle iterierenden Operatoren in OCL (also auch das `select`) durch einen `iterate`-Ausdrucken definieren lassen, so könnte man das angegebene Gegenbeispiel unter Umständen doch derart behandeln, wenn man den `select`-Ausdruck ebenfalls durch ein Programm übersetzt, welches auf der Programmdarstellung des `iterate`-Ausdrucks aufbaut.

Leider bleibt uns im Rahmen dieser Arbeit keine Zeit mehr, um dieser interessanten Frage nachzugehen.

Schließlich betrachten wir den letzten, der oben genannten Einflußfaktoren auf eine programmtechnische Darstellung von `iterate`-Ausdrücken:

Wie berechnen wir die erforderlichen Werte? Für die Darstellung der Berechnungsvorschriften betrachten wir hier grundsätzlich Terme, d.h. die Übersetzungen der Ausdrücke `e0` und `exp` sind Terme. Alle bisher angestellten Überlegungen zur Behandlung des `iterate`-Operators verwenden diese (sehr allgemeine) Betrachtungsweise.

Wir wollen doch auch an dieser Stelle darauf hinweisen, daß durch die hier besprochene Variante der Übersetzung des `iterate`-Operators selbst, sich plötzlich *neue* Möglichkeiten für diesen Aspekt in unserem einfachen Betrachtungsmodells für den `iterate`-Operator auftun, die wir bisher nicht hatten, und damit ein Spielraum für mögliche Optimierungen entsteht:

Da wir nun eine Übersetzung angeben, die die Werte von `iterate`-Ausdrücken – und somit potentiell alle anderen iterierenden Operatoren in OCL (wie `select` und `collect` etc.) – durch *Programme* berechnet, könnte man diese Darstellung bei der Formuiierung des Programms $P \equiv P($`v1`$, \ldots,$ `vN`$) \rightarrow$ `result` ausnutzen, wenn der Wert eines der Teilausdrücke `e0` und `exp` ebenfalls durch ein Programm berechnet werden.

Ein Beispiel wären ineinander geschachtelte `select`-, `collect`- oder `iterate`-Ausdrücke. Man könnte dann jeweils versuchen, die Programme zu den einzelnen durch Programmme beschriebenen Teilausdrücken *direkt* in *einem Programm* für Gesamtausdruck (der Schachtelung) zu verbinden und somit auf Einführung der Symbole zur Bennenung des Ergebniswertes (und die zugehörigen Axiome) zu verzichten.

Ist beispielsweise in dem hier betrachteten `iterate`-Ausdruck E (über eine Sequenz) die Berechnungsvorschrift durch einen Ausdruck `exp` gegeben, der selbst durch ein Programm $P' \equiv P'(\texttt{v1}, \ldots, \texttt{vN+2}) \rightarrow \texttt{result'}$ dargestellt werden kann, so könnten wir prinzipiell für den Ausdruck E ein Programm $P \equiv P(\texttt{v1}, \ldots, \texttt{vN}) \rightarrow \texttt{result}$ der Form

```
T' acc = e0;
Sequence_T remainingPart = c;
while (hasMoreElements(remainingPart)) {
    T e = nextElement(remainingPart);
    P'(v1, ..., vN+2) → result'
    acc = result';
    remaingPart = remainingElements(remainingPart);
}
T' result = acc
```

generieren. Man müßte dabei unter Umständen eine Umbenennung von lokalen Variablen vornehmen, um Namenskonflikte zu vermeiden.

Wir können uns leider auch an dieser Stelle nicht mehr vertieft mit der Thematik auseinandersetzen, wollen aber zumindest deutlich machen, an welchen Stellen bei der Darstellung von `iterate` durch Programme noch Optimierungsmöglichkeiten bestehen.

Es sollte auch deutlich geworden sein, daß die hier genannten und diskutierten drei Gesichtspunkte

- Darstellung der zu iterierenden Kollektion
- Darstellung der Berechnungen von `e0` und `exp` in der Logik
- Die Einbindung des Iterationsergebnis in den umgebenden Kontext

alle Parameter darstellen, die für die Behandlung des `iterate`-Operator durch Programme wesentlich sind.

Abschließend wollen wir nun ein Beispiel betrachten.

Beispiel 60
Wir verwenden den OCL-Ausdruck aus Beispiel 25, der dort mit der Basisabbildung behandelt wurde: $E =$

```
cmp.customers->iterate(c:Customer;
                       acc:Bag(Date) = Bag{} |
                           acc->including(c.birthDate))
```

Für die Übersetzung verwenden wir hier die Möglichkeit, die Menge zu iterierende Menge $C =$ `cmp.customers` als Prädikat darzustellen: $\Phi_{[c]} \equiv c \in cmp.customer$ (wobei $c{:}Customer$ eine neue Variable ist). Wir wollen außerdem das Berechnungsergebnis über eine Funktionssymbol zugreifbar machen.

Als Übersetzung erhalten wir für E erhalten damit:

$$\lceil \texttt{E} \rceil = iterate_E(cmp)$$

sowie folgende Axiomenge Ax_E:

$$
\begin{aligned}
Ax_E \equiv \{ & \dot{\forall} cmp{:}Customer\,(\langle Bag_{Date}\ \texttt{acc = } emptyBag_{Date}\texttt{;} \\
& \quad \texttt{if (Customer.lastCreatedObj != null) \{} \\
& \quad\quad \texttt{Customer e = Customer.firstObj;} \\
& \quad\quad \texttt{do \{} \\
& \quad\quad\quad \texttt{if(}isInSet(\texttt{e}, cmp)\texttt{) \{} \\
& \quad\quad\quad\quad \texttt{acc = } including(\texttt{acc}, \texttt{e}.birthDate)\texttt{;} \\
& \quad\quad\quad\quad \texttt{e = e.nextObj;} \\
& \quad\quad\quad \texttt{\}} \\
& \quad\quad \texttt{\} while (e == Customer.lastCreatedObj)} \\
& \quad \texttt{\}} \\
& \quad Bag_{Date}\ \texttt{result = acc}\rangle\,(iterate_E(cmp) \doteq \texttt{result}) \\
& \dot{\forall} c{:}Customer \dot{\forall} cmp{:}Company\,(isInSet_C(c, cmp) \doteq \\
& \qquad\qquad \texttt{true} \leftrightarrow c \in cmp.customer)\}
\end{aligned}
$$

Man vergleiche das Ergebnis mit der auf Seite 86 erzeugten induktiven Formulierung:

$$
\begin{aligned}
& \lceil \texttt{E} \rceil = iterate_E(cmp.customers) \\
& Ax_E \equiv \{collect_E(emptySet_{Customer}) \doteq emptyBag_{Date} \\
& \qquad \forall s{:}Set_{Customer} \dot{\forall} c{:}Customer\,(\\
& \qquad\quad collect_E(insert(s, c)) \doteq insert(collect_E(remove(s, c)), c.birthDate))\}
\end{aligned}
$$

Kapitel 4

Zusammenfassung und Ausblick

Wir wollen in diesem Kapitel eine kurze Zusammenfassung der Ergebnisse der Arbeit bieten und diesen Rückblick abschließend nutzen, um zu diskutieren, ob bzw. in wie weit wir die Ziele erreicht haben, die wir uns anfänglich gestellt haben.

Zunächst geben wir einen kurzen Überblick über das Gesamtverfahren, das in dieser Arbeit entworfen und implementiert wurde.

4.1 Zusammenfassung

4.1.1 Überblick über das Gesamtverfahren

Die Übersetzung eines OCL-Constraints C in eine Formel Th_C der Logiksprache, die von der betrachteten Anwendung verwendet wird, gliedert sich in einzelne, aufeinanderfolgende Schritte, die im wesentlichen unabhängig Teilaufgaben betreffen. Eine Übersicht bietet die Abbbildung 4.1.

Am Anfang steht als Eingabe für die Abbildung der zu übersetzende OCL-Constraint als Zeichenkette zur Verfügung; diese Zeichenkette könnte beispielsweise von einem CASE-Werkzeug erzeugt worden sein, das die Modellierungssprachen UML/OCL unterstützt. Im KeY-System wird dazu *TogetherCC* verwendet. Diese Zeichenkette wird nun durch die folgenden, aufeinander aufbauenden Stufen in eine Formel Th_C transformiert, die das Ergebnis der Übersetzung in die entsprechende Zielsprache darstellt.

1. *Parsen des Constraints C.*

   Die Zeichenketten-Darstellung wird in einen Syntaxbaum umgesetzt, der die Struktur des Constraints C wiederspiegelt und unter anderem alle notwendigen Typinformationen über Teilausdrücke in C enthält.

   Diese Stufe stellt außerdem sicher, daß der zu verarbeitende Constraint ein *gültiger* OCL-Constraint ist damit eine weitere Behandlung überhaupt sinnvoll ist.

   Wir verwenden hier den Parser der im Rahmen der Diplomarbeit von Frank Finger [Fin00] an der TU Dresden entwickelt wurde.

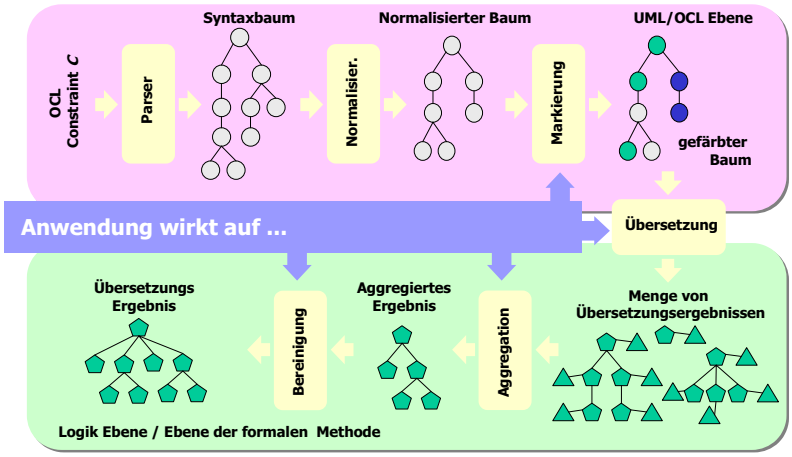


Abbildung 4.1: Überblick über das Gesamtverfahren.

2. *Normalisierung des Syntaxbaums.*

   Der erzeugte Syntaxbaum wird in einen neuen, äquivalenten Syntaxbaum transformiert, der bestimmte Einschränkungen der OCL-Grammatik genügt. Diese Stufe dient hauptsächlich der Vereinfachung der nachfolgenden Stufen, da der durch die Transformation erzeugte (äquivalente) OCL-Ausdruck C' einem eingeschränkten Teil der OCL-Grammatik angehört.

   Man so zum Beispiel sicherstellen, daß jeder iterierende Operator in OCL genau eine Iteratorvariable besitzt, oder daß das Kontextelement für die Auswertung eines Teilausdrucks immer explizit angegeben ist.

3. *Markierung des Syntaxbaums durch Heuristiken.*

   Vor der eigentlichen Übersetzung des normalisierten Constraints C' analysieren die verschiedenen Heuristiken den zugehörigen Syntaxbaum und markieren die Knoten bzw. Teilausdrücke, die durch sie behandelt werden können. Eine Übersetzungssteuerung legt anschließend fest, auf welche Art und Weise (bzw. unter Verwendung welcher Heuristik) ein Teilausdruck aus C' übersetzt werden soll; sie löst unter anderem mögliche Mehrfachmarkierungen von Teilausdrücken auf. Unmarkierte Teilausdrücke werden insbesondere durch die Basisabbildung behandelt, die sozusagen unsere Standardmethode zur Behandlung von OCL-Ausdrücken darstellt.

4. *Übersetzung des markierten Constraints C'.*

   Die einzelnen Teilausdrücke aus C' werden nun entsprechend ihrer Markierung übersetzt. Dabei entstehen *Übersetzungsergebnisse* für jeden Teilausdruck bzw.

Knoten im Syntaxbaum; diese Ergebnisse sind zunächst *keine* Formeln, sondern kapseln alle Informationen, die bei der Übersetzung des zugehörigen Teilausdrucks anfallen.

In der Übersetzung für das KeY-System bestehen diese Ergebnisse zum Beispiel aus dem Term (bzw. der Formel), der die Übersetzung des entsprechenden Teilausdrucks an sich verkörpert, sowie der Menge der zusätzlichen Axiome, die für eine geeignete Interpretation der Symbole sorgt, die während der Behandlung des Teilausdrucks neu eingeführt wurden.

In Abbildung 4.1 werden diese Zusatzinformationen – wie die gerade erwähnten Axiomenmengen – durch kleine Dreiecke angedeutet.

Im wesentlichen findet in dieser Stufe die Übersetzung von OCL-*Ausdrücken* statt.

5. *Aggregation der Übersetzungsergebnisse.*

   Nun können wir aus den Übersetzungsergebnissen zu den einzelnen Teilausdrücken aus C' die Formel der Zielsprache Th'_C generieren, welche der eigentlichen Übersetzung des Constraints C' entspricht.

   Diese Stufe konzentriert sich daher hauptsächlich um die formale Darstellung von *Zuständen* in unserer Zielsprache und die Aggregation der Informationen aus den Übersetzungsergebnissen zu einer einzelnen entsprechenden Formel.

   Für das KeY-System mit der Zielsprache **DL** werden beispielsweise – entsprechend den Ausführungen im Abschnitt 2.3.4 – aus Invarianten bzw. Vor/Nachbedingungen entsprechende Implikationen gebildet, die Axiomenmengen werden in geeigneter Weise zu konjunktiv verknüpften Prämissen dieser Implikationen umgewandelt und möglicherweise wird ein Diamant-Term, der einen Methodenaufruf formalisiert, gebildet und mit den Implikationstermen zu den Vor- und Nachbedingungen geeignet kombiniert.

   Im wesentlichen findet in dieser Stufe also die Übersetzung von OCL-*Constraints* statt.

6. *Bereinigung der Formel Th'_C.*

   In einem letzten Transformationsschritt wird die generierte Formel Th'_C bereinigt und somit eine Formel Th_C erzeugt, die die Übersetzung des Ausgangsconstraints C verkörpert.

   Diese Bereinigungen kümmern sich um die formale Darstellung bestimmter Abkürzungen, die von unserem Verfahren verwendet werden, um bestimmte zustandsabhängige Sachverhalte auszudrücken, die während der Behandlung von OCL-*Ausdrücken* anfallen; man denke beispielsweise an die punktierten Quantoren $\dot{\forall}, \dot{\exists}$, die bezüglich eines Zustands bzw. Snapshots D über die gerade existierenden Instanzen einer Sorte sprechen, sowie die Funktionssymbole, deren Bezeichner mit $@pre$ enden und die dazu dienen, über Auswertungen dieses Symbols im Vorzustand eines Methodenaufrufs zu sprechen.

   Diese Abkürzungen werden in dieser Stufe – beispielsweise durch geeignete Expansionen im Falle der punktierten Quantoren – wegtransformiert.

   Damit kümmert sich diese Stufe prinzipiell um die formale Darstellung dieser speziellen Abkürzungen in der verwendeten Zielsprache und detailliert damit sozusagen, wie man in der verwendeten Zielsprache über die Sachverhalte spricht,

für die die Abkürzungen stehen, zum Beispiel wie man die Auswertung eines Funktionssymbols im Vorzustand eines Methodenaufrufs in der Zielsprache konkret formalisiert.

Eine Separation dieser Stufe von der vorangehenden macht das ganze Verfahren wesentlich durchsichtiger, als ein monolithisches Verfahren, welches zum Beispiel alle zustandsabhängigen Aspekte eines Constraint auf einen Schlag behandelt. Sie ist jedoch schon deshalb sinnvoll, da es für die Behandlung dieser Abkürzungen mehrere Möglichkeiten mit unterschiedlichen Vor- und Nachteilen geben kann und man unter Umständen in bestimmten Situationen unterschiedliche Alternativen einsetzen möchte; für das KeY-System mir der Zielsprache **DL** wurde das für die Behandlung des `@pre`-Operators in [BBS01] ausführlich diskutiert.

Schließlich haben wir eine Formel Th_C erzeugt, die (hoffentlich) unserem Korrektheitskriterium genügt, und somit eine korrekte Formalisierung des Constraints C in der verwendeten Zielsprache darstellt.

Jede dieser einzelnen Teilaufgaben – abgesehen vom Parsen und vielleicht dem Normalisieren – kann nun eine *anwendungsspezifische* Behandlung erfordern. Aus diesem Grunde wurde das Gesamtverfahren im Sinne einer *Separation of Concerns* aus einzelnen, aufeinanderfolgenden Stufen aufgebaut, um eine optimale Anpaßbarkeit des Gesamtverfahrens an eine *spezielle* Anwendung zu gewährleisten.

Unter diesen Gesichtspunkten ergibt sich nun ebenfalls eine *neue, zusätzliche* Interpretation der Anwendung von Heuristiken: Heuristiken dienen nicht nur zur Optimierung der Übersetzungsergebnisse bezüglich eines von der konkreten Anwendung induzierten Gütekriteriums, sondern bieten auch gleichzeitig eine Möglichkeit, um den *universellen* Kern des Verfahrens, der auf Prädikatenlogik basiert und von der Basisabbildung, die in Kapitel 2 beschrieben wurde, realisiert wird, auf eine spezielle und möglicherweise sehr ausdrucksstarke Logiksprache auszuweiten, welche in der speziellen Anwendung verwendet wird. Man könnte deshalb auch sagen, daß Heuristiken – neben den Optimierungsaspekten – eine Methode darstellen, um eine spezielle Anpassung (*Customizing*) des *übertragbaren* und *anwendungsunabhängigen* Kerns des Verfahrens an die Anforderungen einer *speziellen* Anwendung vorzunehmen. Somit entsteht eine *anwendungsspezifische* und im allgemeinen *nicht mehr übertragbare* Übersetzung.

Man beachte zudem, daß diese Struktur einer Anwendung Einfluß auf das Gesamtverfahren unter Rückgriff auf Informationen aus *allen* Ebenen, mit denen die Übersetzung zu tun hat, gewährt und somit für die Anwendung alle Möglichkeiten für Optimierungen und Anpassungen des Verfahrens ihre besondere Bedürfnisse offenstehen; sowohl die UML/OCL-Ebene, als auch die Ebene der verwendeten formalen Methode können für Optimierungen und Anpassungen herangezogen werden.

4.1.2 Ergebnisse der Arbeit

Schlußendlich haben wir *keine* spezialisierte Abbildung beliebiger OCL-Constraints in Formeln der Sprache **DL** entwickelt, die nur für das KeY-Projekt von nutzen sein würde, sondern allgemeiner ein Rahmenwerk für eine Abbildung von OCL-Constraint in *beliebige* Logiksprachen[1] entworfen, daß sich leicht an eine *konkrete* Zielsprache

[1] Diese Logiksprachen müssen in irgendeiner Weise Erweiterungen einer prädikatenlogischen Sprache sein. Das sollte im allgemeinen keine besondere Einschränkung darstellen, da die meisten Formalismen, die im softwaretechnischen Umfeld verwendet werden, dieser Anforderung genügen.

anpassen läßt.

Wir haben besonderes Augenmerk auf die *Güte* der erzeugten Formel gelegt und diesem Aspekt in Form von Heuristiken in dem Verfahren Rechnung getragen.

Der ganze Prozeß der Anpassung und Optimierung des Gesamtverfahrens an eine spezielle Anwendung wurde an einem durchgehenden Beispiel – dem KeY-System – demonstriert.

Dabei wurde unter anderem eine mächtige Heuristik *Kollektionen als Mengen* entwickelt, die auf die Beschreibung von Kollektionen durch Formeln abzielt. Da diese Heuristik rein prädikatenlogische Ausdrucksmittel verwendet ist sie prinzipiell auch auf andere Zielsprachen übertragbar.

Das gesamte Verfahren wurde in Java implementiert und ist im Internet sowohl als integraler Bestandteil des KeY-Systems, als auch als eigenständige und von KeY-System unabhängige Version verfügbar (siehe `i12www.ira.uka.de/~key`).

Schließlich wurde zur Unterstützung eines iterativen Entwicklungs- und Optimierungsprozesses für eine anwendungsspezifische Übersetzung eine Testumgebung entwickelt, die es ermöglicht, in Form von XML-Dateien Bibliotheken (*Testsuites*) von einzelnen Testfällen (*Testcases*) anzulegen und eine solche Bibliothek automatisch ausführen zu lassen. Die Testfällen beschreiben dabei im wesentlichen einen zu übersetzenden OCL-Constraint, sowie Kontrollparameter, die zur Steuerung der Übersetzung dieses Constraints dienen. Der Vergleich zwischen verschiedenen Varianten einer anwendungsspezifischen Übersetzung bezüglich einer beliebig großen Menge von OCL-Constraints wird dadurch möglich. Ein Beispiel einer solchen Testsuite findet sich im Anhang.

4.2 Nicht behandelte Aspekte

Dieser Abschnitt faßt kurz zusammen, welche Aspekte – sei es theoretischer oder implementierungstechnischer Natur – in dieser *nicht* behandelt wurden und aus welchen Gründen wir uns in den einzelnen Fällen so entschieden haben.

4.2.1 Theoretischer Teil der Arbeit

Die Behandlung der folgenden Punkte wurden für den theoretischen Teil der Arbeit ausgespart.

Metaeigenschaften des Typs OclType. Das OCL-Typsystem bietet einem Modellierer mittels des Typs OclType und seiner Eigenschaften die Möglichkeit, in beschränktem Maße auf das UML-Metamodell zuzugreifen. Beispielsweise kann man so alle Methoden berechnen, die für eine Klasse definiert wurden, sowie Aussagen über alle Obertypen einer Klasse treffen. Da wir diesen Ausdrucksmitteln im Moment keine übermäßige Bedeutung für die praktische Modellierung mit UML/OCL beimessen, haben wir diesem Typen nicht weiter behandelt, um den Umfang dieser Arbeit nicht noch

weiter zu erhöhen. Eine wichtige Ausnahme wurde jedoch besprochen und behandelt: Der `allInstances`-Operator.

Der Typ OclState. Wir betrachten den Typen OclState im Augenblick als nicht relevant für den praktischen Einsatz von OCL, da die zugehörigen Eigenschaften zu rudimentär sind, als das sie wirklich nützlich erscheinen.

Undefiniertheit von OCL-Ausdrücken. OCL-Ausdrücke können zu einem undefinierten Wert ausgewertet werden. OCL basiert damit prinzipiell auf einer dreiwertigen Logik. Wir haben diese bedeutenden Aspekt aus Platzgründen ebenfalls weg gelassen, da undefinierte Ausdrücke in den meisten Fällen anscheinend durch „schlampige" Formulierungen dieser Ausdrücke enstehen, d.h. es scheint möglich zu sein, zu einem Constraint C, der undefiniert sein kann, einen äquivalenten OCL Ausdruck C' anzugeben, welcher in keinem Falle undefiniert ist. Mit anderen Worten: Man könnte sicherlich durch eine sorgfältige Formulierung eines OCL-Ausdrucks auf undefinierte verzichten.

Korrektheitsbeweis der Verfahrens. Wir haben in dieser Arbeit zwar ein Korrektheitskriterium skizziert, einen formalen Nachweis, daß die von uns angegebene Basisabbildung in diesem Sinne korrekt ist, haben wir hingegen aus Platzgründen nicht behandelt.

Für einen solchen formalen Beweis wäre unbedingt eine *anwendungsabhängige* Präzisierung des Korrektheitskriteriums notwendig.

Die Erstellung eines solchen Nachweises wäre sicherlich sehr interessant, man müßte dabei insbesondere untersuchen, wie sich das Argument für die Korrektheit der Basisabbildung über die Technik der Heuristiken auf die Korrektheit der optimierten Abbildung erweitern läßt. Für diese Erweiterung wäre dann möglicherweise ein zusätzliches „Korrektheitskriterium" notwendig, welches den durch die Heuristiken möglicherweise eingeführten neuen Darstellungen von modellierten Sachverhalten Rechnung trägt – man denke beispielsweise an die prädikativen Darstellungen.

Andere Diagrammtypen aus UML. Wir haben mit unserer Abbildung bisher nur eine einzige Diagrammart – nämlich Klassendiagramme – aus der recht reichhaltigen Modellierungssprache UML in der **DL** formalisiert. Das hängt zum einen damit zusammen, daß Klassendiagramme – im Vergleich zu den anderen Diagrammarten aus der UML – am häufigsten in der Praxis eingesetzt werden, zum anderen ist die formale Semantik von Klassendiagrammen (*Snapshots*) recht gut verstanden. Für die anderen Diagrammarten – vielleicht mit der Ausnahme von Zustandsübergangs-Diagrammen (*State diagramms*)– ist das sicherlich noch nicht in dem Maße der Fall und teilweise vielleicht auch gar nicht möglich! Es gibt erfreulicherweise eine Fülle von laufenden Arbeiten zur formalen Behandlung und Verarbeitung anderer Diagrammarten aus der UML, so daß wir vielleicht in Zukunft mit unserer Abbildung auch die Informationen formal beschreiben können, die in Diagrammen dieser Art durch den Modellierer dargestellt wurden.

4.2.2 Implementierungsteil der Arbeit

Die folgenden Aspekte wurden in der Arbeit zwar theoretisch behandelt, sind jedoch in der Implementierung bisher noch nicht bzw. nicht in der dargestellten Allgemeinheit

umgesetzt.

Generierung der Axiomenmenge $Th_{\mathcal{D}}$. Wir haben in Abschnitt 2.3.2.2 dargelegt, wie die mit dem UML-Modell $\mathcal{D}$ verbundene Semantik der einzelnen Modellierungselemente in der **DL** (bzw. einer prädikatenlogischen Sprache) formal durch eine geeignete Formelmenge $Th_{\mathcal{D}}$ beschrieben werden kann. Die Implementierung hingegen erzeugt im Augenblick diese Menge $Th_{\mathcal{D}}$ noch nicht.

Heuristiken. Heuristiken sind in der Implementierung zwar unterstützt, jedoch noch nicht in der Allgemeinheit, wie sie in dieser Arbeit dargestellt wurde.

4.3 Ausblick

4.3.1 Behandlung von der Undefiniertheit

Wir haben die Undefiniertheit von OCL-Ausdrücken in dieser Arbeit unter anderem deshalb nicht behandelt, da die Undefiniertheit anscheinend nur bei „schlampig" formulierten OCL-Ausdrücken auftritt. Trotzdem kann man eine solche Formulierung durch den Modellierer nicht verhindern und muß deshalb mit der Undefiniertheit von OCL-Ausdrücken umgehen.

Dazu gibt es verschiedene Möglichkeiten:

- **Verwendung einer mehrwertigen Logik.**

  Man verwendet statt einer zweiwertigen Logik – wie beispielsweise **DL**– eine dreiwertige Logik und kann somit – unter Anpassung des Korrektheitskriteriums – recht einfach und direkt die Undefiniertheit in das Verfahren einbauen.

  Dieser Ansatz birgt jedoch zwei entscheidende Nachteile: Zum einen sind mehrwertige Logiken recht spezialisierte Logiken, für deren deduktive Behandlung besondere Beweiser notwendig sind; insbesondere im softwaretechnischen Umfeld scheint es eher unwahrscheinlich, häufig auf derartige Beweissysteme zu stoßen. Damit wäre das entstehende Verfahren unter Rückgriff auf mehrwertige Logiken in der Regel *nicht mehr übertragbar*!

  Zum anderen ist das Beweisen in einem Deduktionssystem für mehrwertige Logiken zunächst komplizierter, da die „neuen" bzw. zusätzlichen Wahrheitswerte im Beweis an *sehr vielen* Stellen eine Rolle spielen. Andererseits haben wir mehrfach dargelegt, daß die Undefiniertheit nur in Randsituationen auftritt und sicherlich viele OCL-Ausdrücke, die von einem Modellierer formuliert werden, niemals undefiniert sind. Man würde sich somit den leichten Einbau der Behandlung der Undefiniertheit in die Übersetzung zu einem hohen Preis erkaufen, da im Beweisen somit in vielen Fällen mit einem unnötigen und an vielen Stellen auftretenden Overhead umzugehen wäre.

  Bemerkung (Reduktion einer mehrwertigen Logik). Man könnte nun versuchen, die beiden genannten Probleme zu beseitigen, indem man durch ein Reduktionsverfahren die Formeln der benutzten dreiwertigen Logik durch in einem gewissen Sinne äquivalente Formeln einer zweiwertigen Logik ersetzt. Eine solche Reduktion ist beispielsweise in [Sch01c, Abschnitt 2.6, Seite 101] für die dreiwertige Logik $\mathcal{L}_3$ beschrieben und müßte gegebenenfalls im Detail angepaßt

werden. Dieses Vorgehen wäre jedoch ebenfalls problematisch: Zunächst müßte geklärt werden, ob eine solche Anpassung des Verfahrens überhaupt möglich ist. Desweiteren wird bei dieser Reduktion die erzeugte Übersetzung in der mehrwertigen Logik in eine Normalform gebracht und jede atomare Formel in dieser Normalform durch *zwei* verschiedene Formeln ersetzt. Außerdem werden auch hier zusätzliche Axiome eingeführt. Es entsteht also eine Formel, die sich in ihrer Struktur möglicherweise sehr stark von dem ursprünglichen OCL-Ausdruck unterscheidet, was sehr ungünstig für das Verständnis der Ergebnisformel sein kann.

- **Darstellung der Undefiniertheit eines Ausdrucks durch eine Formel.**

  Sowohl der reine mehrwertige Ansatz, als auch der Reduktionsansatz betrachten zwei prinzipiell *unabhängige* Aspekte eines OCL-Ausdrucks auf einmal: Die eigentliche Bedeutung des Ausdrucks und die Eigenschaft der Undefiniertheit. Das Vermischen dieser Aspekte bei der Übersetzung macht das Ergebnis der Übersetzung komplizierter – sei es für die deduktive Behandlung durch ein Beweissystem, oder aber für das Verstehen durch einen Menschen.

  Naheliegender und eleganter wäre, diese beiden unabhängigen Sachverhalte im Verfahren zu trennen. Wir wollen nun kurz darlegen, wie eine solche Trennung vollzogen werden könnte:

  Für Anwendungen ist ein OCL-Constraint C eigentlich nur dann relevant, wenn er bezüglich eines Snapshots D eines UML-Modells $\mathcal{D}$ einen *definierten* Wert annimmt, also entweder gilt, oder falsch ist. Die Undefiniertheit entsteht im wesentlichen durch eine unsaubere Formulierung des OCL-Ausdrucks.

  Anwendungen betrachten und arbeiten daher prinzipiell mit definierten OCL-Ausdrücken. Wir könnten damit in unserem Verfahren für einen OCL-Constraint C zunächst eine Formel Th_C generieren, die die eigentliche Aussage dieses Constraints wiederspiegelt, wobei angenommen wird, daß der Constraint definiert sei. Desweiteren wird eine *zusätzliche* Formel $Undef_C$ generiert, welche genau ausdrückt, ob der gegebene OCL-Ausdruck definiert bzw. undefiniert ist.

  Wir haben dieses Verfahren im Verlaufe dieser Arbeit auch grob untersucht und sehen momentan die folgenden Vorteile:

  - Die Bedeutungsformel Th_C eines Constraints C wird einfacher, da keine Teile der Formel sich mit der Undefiniertheit des Constraints befassen, die mit der eigentlichen Aussage von C eigentlich nicht zu tun hat.
  - Für die Generierung der Bedeutungsformel Th_C können wir das bisherige Verfahren unverändert übernehmen. Das Gesamtverfahren muß lediglich in einem zusätzlichen Übersetzungslauf die Undefiniertheitsformel $Undef_C$ erzeugen.
  - Für den Modellierer wird durch die Formel $Undef_C$ klar dargelegt, unter welchen Umständen der von ihm formulierte Constraint undefiniert ist und kann somit zur Fehlerbeseitigung im Constraint C dienen.
  - Die Komposition dieser beiden Aspekte kann wiederum *anwendungsabhängig* sein. Die Anwendung hat dann bei diesem Vorgehen alle Möglichkeiten, die beiden Formeln (bzw. ähnliche Formeln aus weiteren Constraints $C_1, \ldots C_n$) entsprechend den Anforderungen der Anwendung zusammenzusetzen. Entscheidend ist hier, daß nur die Anwendung um die konkrete

Verarbeitung der generierten Formeln – beispielsweise in Form von *Beweisverpflichtungen* – weiß und selbst innerhalb einer Anwendung – zum Beispiel dem KeY-System – verschiedene Verarbeitungsmöglichkeiten für die generierten Formeln möglich sind.

So könnte man sich für den Nachweis einer Beweisverpflichtung, welche einen Constraint C verwendet, beispielsweise vorstellen, daß in einem entsprechenden Beweis zunächst zwei Beweisziele zu bearbeiten sind: Ein Beweisziel enthält die Bedeutungsformel Th_C des Constraints und verwendet damit die Formel $\neg\, \mathit{Undef}_C$ als Prämisse, wobei ein zweites Beweisziel dem Nachweis der Definiertheit des Constraints, d.h. der Formel $\neg\, \mathit{Undef}_C$, – und damit der zugehörigen Prämisse im anderen Beweisziel – dient.

- Die Erzeugung der Undefiniertheitsformel scheint keine besonderen Ausdrucksmittel verwenden zu müssen, die über die von unserer Basisabbildung benutzten Konstrukte hinausgehen, und ist damit prinzipiell auch übertragbar. Wir erhalten also insgesamt ein Verfahren, welches prinzipiell auf klassischer (zweiwertiger) Prädikatenlogik erster Stufe aufbaut und damit immer noch auf viele verschiedene Zielsprachen übertragbar ist!
- Die Undefiniertheit von Kollektionsausdrücken läßt sich anscheinend sehr gut über unsere prädikativen Beschreibungen charakterisieren, so daß die resultierenden Formeln einfacher darzustellen sein sollten, als Bedeutungsformeln beliebiger OCL-Ausdrücke.

- **Darstellung der Undefiniertheit eines Ausdrucks durch einen OCL-Ausdruck.**

  Bei der Untersuchung der Darstellung der Undefiniertheit durch eine Formel Undef_C konnten wir eine interessante Beobachtung machen: OCL bietet anscheinend alle Sprachmittel, um die Undefiniertheit eines OCL-Ausdrucks `e` in OCL selbst durch weiteren Ausdruck `undefined-e` zu beschreiben, d.h. genauer, daß gilt

  > Für jeden Snapshot D des UML-Modells $\mathcal{D}$ und
  > alle Variablenbelegungen β gilt:
  > $\langle$`undefined-e`$\rangle_{D,\beta}$ ist nicht undefiniert
  > und
  > $\langle$`undefined-e`$\rangle_{D,\beta} = \mathit{true}$ **gdw.** $\langle$`e`$\rangle_{D,\beta}$ ist undefiniert

  Wir könnten also zur Generierung der Undefiniertheitsformel Undef_C zunächst den OCL-Ausdruck `undefined-e` erzeugen und daraufhin diesen Ausdruck mit unserem Übersetzungsverfahren für beliebige OCL-Ausdrücke behandeln.

  Diese Vorgehensweise hätte – im Vergleich zu der vorangehenden Möglichkeit auf der Logikebene – folgende Vorteile:

  - Es ist keine Entwicklung einer neuen und zusätzlichen Übersetzung für Undefiniertheitsformeln notwendig, da die vorhanden Übersetzung wiederverwendet werden kann.
  - Die Wiederverwendung hält die Implementierung des Gesamtsystems einfacher und verständlicher, da keine zweite, separate Übersetzung und möglicherweise getrennte Heuristiken realisiert werden müssen. Das Gesamtverfahren ist sehr viel einheitlicher.

- Der Ausdruck `undefined-e` scheint sehr gut für unsere Optimierungen mittels prädikativer Darstellungen geeignet. Insbesondere unsere Heuristik *Kollektionen als Mengen* sehr häufig anwendbar, womit recht einfache Undefiniertheitsformeln entstehen sollten.
- Die Implementierung der Generierung des OCL-Ausdrucks `undefined-e` kann über einfache Zeichenkettenoperationen stattfinden und wird somit sehr viel einfacher sein, als eine entsprechende Implementierung der direkten Generierung von $Undef_C$.
- Die Beschreibung der Undefiniertheit eines OCL-Ausdrucks `e` in OCL selbst dürfte für den Modellierer selbst einfacher zu lesen sein, als eine äquivalente Beschreibung in einer Logiksprache, d.h. der generierte OCL-Ausdruck könnte – unabhängig von der Übersetzung – wahrscheinlich noch besser zur Fehlersuche in der Formulierung eines Constraints eingesetzt werden, als die entsprechende Undefiniertheitsformel.

Bemerkung (Undefiniertheit und Deutung durch Unterspezifikation).
Eine natürliche und für OCL gut geeignete Interpretation der Undefiniertheit besteht in der Betrachtung eines undefinierten OCL-Ausdrucks als eine *unterspezifizierte* Größe: Wir haben mit einem undefinierten Ausdruck `e` zwar einen Wert beschrieben, wissen aber überhaupt nichts über diesen Wert und können somit auch nichts über seine Eigenschaften aussagen.

Wir haben schon an verschiedenen Stellen in dieser Arbeit eine Technik angewandt, die allgemein auch dazu verwendet werden kann, eine solche Unterspezifikation in der Logik nachzubilden: Unsere Benennungstechnik.

Sei `e` ein OCL-Ausdruck, der undefiniert sein kann, und $\Phi_{undef,e}$ eine Formel, die die Situationen beschreibt, unter denen der Ausdruck `e` unbestimmt ist. Sei $\lceil \mathtt{e} \rceil$ eine Übersetzung von `e`, die sich *nicht* mit der Undefiniertheit von `e` befasst und nur die Semantik von `e` im definierten Fall wiederspiegelt.

Dann können wir eine Übersetzung von `e` durch einen Term angeben, der im definierten Falle genau die (vorhandene) Übersetzung wiederspielt und im undefinierten Fall eine *Unterspezifikation* des Übersetzungsergebnis garantiert:

Seien $p_1, \ldots, p_n$ die freien Variablen aus $\lceil \mathtt{e} \rceil$ und $\Phi_{undef,e}$. Seien $T_1, \ldots, T_n$ die entsprechenden Sorten und T die Sorte der vorhandenen Übersetzung $\lceil \mathtt{e} \rceil$.

Dann erzeugen wir ein neues Funktionssymbol

$$f{:}T_1 \times \ldots \times T_n \to T$$

und definieren seine Bedeutung durch das Axiom

$$\dot{\forall} p_1{:}T_1 \ldots \dot{\forall} p_n{:}T_n \; (\neg \Phi_{undef,e} \to f(p_1, \ldots, p_n) \doteq \lceil \mathtt{e} \rceil)$$

Als *neue Übersetzung* für `e` verwenden wir schließlich den Term $f(p_1, \ldots, p_n)$.

Dieses Vorgehen ist sehr allgemein und im Grunde losgelöst von einem Verfahren, das – möglicherweise sehr geschickt, trickreich und auf komplexe Weise – gute Darstellungen der eigentlichen – definierten – Semantik eines Ausdrucks `e` erzeugt. Die Beschreibung der Situationen, in denen ein Ausdruck `e` undefiniert ist, – also die Erzeugung der Formel $\Phi_{undef,e}$ – ist dagegen sehr viel einfacher möglich und muß unter Umständen gar nicht besonders trickreich erzeugt werden.

Die dargelegte Technik muß außerdem nicht auf alle Teilausdrücke eines zu übersetzenden Constraints C angewendet werden, sonder lediglich für diejenigen Teilausdrücke, die *selbst* einen undefinierten Wert *erzeugen* können, beispielsweise ein `oclAsType`-Ausdruck. Dabei handelt es sich über eine relativ kleine und klar zu ermittelnde Menge von Möglichkeiten. D.h. insbesondere, daß wir für alle anderen Teilausdrücke im Constraint, die selbst nur dann undefiniert sind, wenn einer ihrer Teilausdrücke undefiniert ist, eine vorhandene Übersetzung, die ausschließlich die Semantik im definierten Fall betrachtet, nicht anzupassen brauchen. Wir würden auf diese Weise bei der Übersetzung eines Constraints im allgemeinen lediglich eine geringe Zahl von neuen Symbolen und Axiomen einführen müssen; die grobe Struktur der dann erzeugten Formel würde sich daher in vielen Fällen kaum von derjenigen unterscheiden, die durch das vorhandene Verfahren produziert wird.

Insgesamt scheint diese Technik also ein interessanter und klarer Ansatzpunkt für die Behandlung der Undefiniertheit von OCL-Ausdrücken zu sein. □

4.3.2 Behandlung des Typs OclType

Der Typ OclType bietet dem Modellierer im wesentlichen beschränkten Zugriff auf Informationen aus dem UML-Metamodell.

Ist nun ein konkretes UML-Modell $\mathcal{D}$ gegeben, so kann man prinzipiell alle diese Informationen direkt aus dem Modell ablesen. Damit wäre die folgende Darstellung des Typs OclType möglich:

Wir verwenden einen ADT OclType, der alle Eigenschaften von OclType durch gleichnamige Funktionssymbole darstellt. Zudem gibt es für jeden Typen T eine gleichnamige Konstante T:OclType, die das entsprechende OCL-Typobjekt repräsentiert.

Die Semantik der einzelnen Funktionssymbole ist nun eindeutig durch das gegebene UML-Modell $\mathcal{D}$ festgelegt und kann durch eine entsprechende Formelnmenge, die durch eine rein syntaktische Analyse des Modells $\mathcal{D}$ erzeugt wird, automatisch axiomatisiert werden.

Wir führen unter anderem für die Typkonstanten $T_1, \ldots, T_n$ entsprechend unserer Beschreibung der Behandlung von Aufzählungstypen eine Formeln ein, welche definieren, daß durch die Typkonstanten jeweils unterschiedliche Typobjekte bezeichnet werden, und daß *jedes* Typobjekt durch eine dieser Konstanten bezeichnet wird.

Den Operator `allInstances` können wir dabei wie bisher behandeln und müßten deshalb kein entsprechendes Funktionssymbol vorsehen.

Betrachten wir beispielsweise das UML-Modell aus Abbildung 4.2, dann ergäbe

sich als Signatur für den ADT OclType

$$
\begin{array}{l}
A : \text{OclType} \\
B : \text{OclType} \\
C : \text{OclType} \\
\\
name{:}\text{OclType} \to \text{String} \\
attributes{:}\text{OclType} \to Set_{\text{String}} \\
associationEnds{:}\text{OclType} \to Set_{\text{String}} \\
operations{:}\text{OclType} \to Set_{\text{String}} \\
supertypes{:}\text{OclType} \to Set_{\text{OclType}} \\
allSupertypes{:}\text{OclType} \to Set_{\text{OclType}}
\end{array}
$$

und als Axiome erzeugen wir

$$
\begin{array}{l}
\dot{\forall} t{:}\text{OclType}\,(t \doteq A \vee t \doteq B \vee e \doteq C) \\
\neg A \doteq B \\
\neg A \doteq C \\
\neg B \doteq C \\
\\
name(A) \doteq' A' \\
name(B) \doteq' B' \\
name(C) \doteq' C' \\
\\
attributes(A) \doteq insert(emptySet_{\text{String}}, 'attrA') \\
attributes(B) \doteq insert(emptySet_{\text{String}}, 'attrB') \\
attributes(C) \doteq emptySet_{\text{String}} \\
\\
associationEnds(A) \doteq insert(emptySet_{\text{String}}, 'assEnd1') \\
associationEnds(B) \doteq emptySet_{\text{String}} \\
asscociationEnds(C) \doteq emptySet_{\text{String}} \\
\\
operations(A) \doteq insert(emptySet_{\text{String}}, 'opA') \\
operations(B) \doteq emptySet_{\text{String}} \\
operations(C) \doteq insert(insert(emptySet_{\text{String}}, 'opC\_1'), 'opC\_2') \\
\\
supertypes(A) \doteq emptySet_{\text{OclType}} \\
supertypes(B) \doteq insert(emptySet_{\text{OclType}}, A) \\
supertypes(C) \doteq insert(emptySet_{\text{OclType}}, B) \\
\\
allSupertypes(A) \doteq emptySet_{\text{OclType}} \\
allSupertypes(B) \doteq insert(emptySet_{\text{OclType}}, A) \\
allSupertypes(C) \doteq insert(insert(emptySet_{\text{OclType}}, B), A)
\end{array}
$$

4.3.3 Erleichterung des Verständnis der generierten Formeln

Eine der zentralen Techniken bei der Übersetzung von OCL-Operatoren durch die Basisabbildung ist die *Benennungstechnik*. Wir haben dabei zur Behandlung eines (top-level) Operators aus einem OCL-Ausdruck E ein neues Funktionsymbol op_E in Σ^* eingeführt, dessen Semantik durch eine Menge zusätzlicher Axiome formal beschrieben wurde.

Wie ausführlich im Kapitel 3 dargelegt wurde, macht aber gerade diese Technik unter Umständen das Lesen und verstehen komplexer OCL-Ausdrücke sehr schwer, da der menschliche Beweiser zunächst die Intention hinter dem neuen Symbol op_E gewissermaßen innerhalb der Übersetzung Th_C zusammensuchen muß.

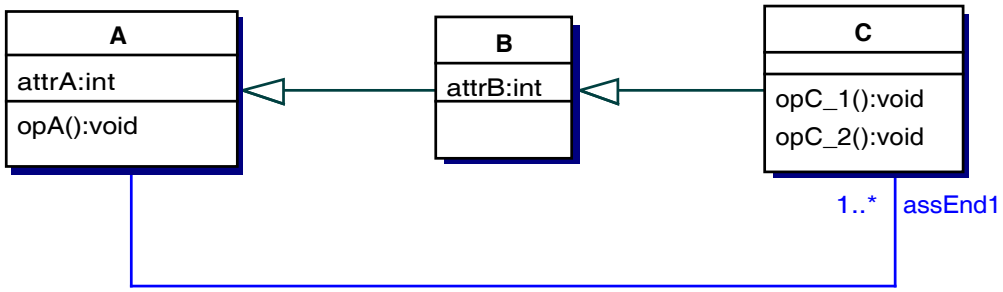


Abbildung 4.2: Ein Beispielmodell für automatische Erzeugung des ADTs OclType.

Um diesen Nachteil etwas abzuschwächen, schlagen wir bei in einer Implementierung des Verfahrens das folgende Vorgehen vor:

Die Intention eines über die Benennungstechnik eingeführten Symbols op_E eröffnet sich dem Menschen sehr leicht, wenn der OCL-Ausdruck E für dessen Behandlung das Symbol op_E überhaupt eingeführt wurde, für den Leser direkt verfügbar wäre, da man davon ausgehen kann, daß der Leser die Aussage des übersetzten Constraints C versteht.

Bei der Übersetzung eines OCL-Ausdrucks E könnte man beim Einführen eines solchen neuen Symbols, das Symbol an sich mit dem entsprechenden OCL-Ausdruck markieren, für dessen Übersetzung das Symbol eingeführt wurde.

Die Anwendung kann diese zusätzliche Information dann gewinnbringend an den menschlichen Benutzer weitergeben: Wird beispielsweise wie im KeY-System die generierte Formel am Bildschirm ausgegeben und sind die Teile der Formel durch den Mauszeiger zur weiteren Verarbeitung auszuwählen, so könnte man bei der Auswahl eines Teilterms der Formel als sogenannten *Tool-Tip* oder in einer separaten Informationszeile den OCL-Ausdruck E ausgeben, der zu der Einführung des zugehörigen top-level-Operators des Teilterms geführt hat. Bei einem $select_E$-Symbol beispielsweise wäre somit das zugehörige Filterkriterium sehr leicht erkennbar und das Verstehen der generierten Formeln durch einen menschlichen Beweiser erheblich vereinfacht.

Man kann diese Idee auch weiter verallgemeinern: Trennt man in einem konkreten Anwendungssystem – wie beispielsweise KeY– die graphische *Darstellung* eines Terms am Bildschirm von seiner *syntaktischen Struktur*, dann könnte man sogar einen Schritt weiter gehen und bei der Übersetzung eines OCL-Ausdrucks E' für die neu eingeführten Symbole op_E zu einem OCL-Teilausdruck E sofort geeignete *Pretty-Printing*-Informationen generieren, die dann später bei der Darstellung einer Formel durch einen *Pretty-Printer* genutzt und umgesetzt würden. Im einfachsten Fall wäre das beispielsweise die simple Information, daß das Symbol op_E postfix unter Verwendung der Punktnotation wie in OCL dargestellt werden soll, man könnte den Pretty-Printer aber auch veranlassen, in graphischen Darstellung eines Terms direkt Informationen anzuzeigen, die in der logischen Struktur des Terms nicht vorhanden bzw. für diese Struktur irrelevant sind. Man denke wieder an einen `select`-Ausdruck: $E =$ `c->select(e | b)` wird unter der Basisabbildung durch den Term $\lceil$`E`$\rceil = select_E(\lceil$`c`$\rceil, p_1, \ldots, p_n)$ dargestellt und insbesondere die Information über das Filterkriterium (die Implizit im Index E steckt!) ist für einen Menschen nicht direkt verfügbar, wenn man diesen Term in der gewohnten Manier graphisch darstellt. Warum sollte man den Pretty-Printer nicht anweisen, den Term in der GUI in OCL-

Notation auszugeben, also in der Form ⌈c⌉->*select*(*e*| b). Die Benutzter-Interaktionskomponente müßte dann sicherstellen, daß lediglich die relevanten freien Variablen $p_1, \ldots, p_n$ aus ⌈b⌉ und der Term ⌈c⌉ als Teilausdücke für Benutzerinteraktionen zur Verfügung stehen.

Man beachte, daß die notwendigen Darstellungsinformationen *während* der Übersetzung eines OCL-Ausdrucks generiert werden müssen, da im Nachhinein im allgemeinen keine (auf syntaktischen Analysen basierende) Verknüpfung zwischen den Funktionssymbolen aus der Übersetzung und den Teilausdrücken aus dem ursprünglichen OCL-Ausdruck mehr möglich ist, und deshalb uns eine Erwähnung dieser Idee im Rahmen dieser Arbeit durchaus sinnvoll erscheint.

4.3.4 Anpassung an den zukünftigen Standard OCL 2.0

Im kommenden Standard OCL 2.0 wird es einige grundlegende Veränderungen geben. Ohne näher darauf einzugehen, wollen wir lediglich anmerken, daß zu gegebener Zeit eine Anpassung unseres Verfahrens an diesen neuen Standard untersucht und vorgenommen werden muß. Wir sind aber recht zuversichtlich, daß sich das vorgestellte Verfahren in geeigneter Weise und ohne allzugroße Anstrengungen modifizieren läßt.

4.4 Rückblick und Schlußwort

Wir haben zu Beginn dieser Arbeit in Abschnitt 2.1 auf Seite 9 einen Katalog von Eigenschaften aufgestellt, die wir uns für eine solche Übersetzung wünschen würden. Wir wollen diesen letzten Abschnitt schließlich verwenden, um uns in einem kurzen, kritischen Rückblick zu fragen, ob bzw. in wie weit wir die von uns damals gesetzten und sicherlich ehrgeizigen Ziele erreicht haben.

- **Korrektheit.** Die Korrektheit unseres Verfahrens wurde nicht formal nachgewiesen.

  Wir haben zwar ein allgemeines, aber für einen konkreten Korrektheitsbeweis sicherlich zu allgemeines Korrektheitskriterium angegeben, d.h. in einem Korrektheitsbeweis für eine konkrete Zielsprache wird man vermutlich eine stärkere Eigenschaft der Übersetzung nachweisen, die dann das angegebene Kriterium impliziert; eine weitere Formulierung des Korrektheitskriterium, welches sich auf typisierte Prädikatenlogiken abstützt, findet sich in dem Papier [BKS01]; diese Formulierung bezieht sich aber auschließlich auf die *statische* Sicht eines Systems.

  Die Angabe eines formalen Korrektheitskriteriums wurde nach unserem Wissen bisher – mit Ausnahme unseres Papiers [BKS01] – in keiner anderen Arbeit zuvor unternommen. Auch wenn das von uns angegebene Kriterium für einen Beweis zu allgemein ist, so präzisiert es doch die intuitive Idee, was Korrektheit für eine solche Übersetzung bedeutet, und beseitige somit mögliche Unklarheiten. Wir halten eine solche formale Formulierung für ungemein wichtig, da dadurch Fehler vermieden werden, man denke beispielsweise an die von uns angemerkten Fehler in der Arbeit [HHK98].

  Wie wir außerdem versucht haben darzulegen, entsteht bei der Beschreibung von Korrektheit durch *Informationserhaltung* eine recht interessante Hierarchie von Begriffen, die einen tiefergehenden Einblick in die Natur einer solchen Abbildung bietet.

- **Erfüllung eines Gütekriteriums.** Wir haben im Rahmen dieser Arbeit sicherlich kein „perfektes" Verfahren für das KeY-Projekt entwickelt. Das war unter den gegebenen Umständen in dieser Form auch gar nicht möglich, da dazu eine große Menge von wirklich für das System relevanten Anwendungsbeispielen notwendig wäre, die zur Zeit jedoch noch nicht verfügbar ist. Wir haben uns daher darauf konzentriert, ein flexibles Verfahren zu entwickeln, daß leicht und „beliebig" verbesserbar ist.

  Das Verfahren wurde zudem so konzipiert, daß eine *inkrementelle* Verbesserung und Anpassung des Verfahrens durch die Technik der Heuristiken und unsere Testumgebung möglich ist, was dem konkreten Entwicklungsprozeß in der Praxis recht zuträglich sein dürfte.

  Die wirkliche Güte der Übersetzung für unsere spezielle Anwendung KeY wird sich erst in Zukunft durch konkrete Projekterfahrungen zeigen. Diese Erfahrung kann dann gerade dafür genutzt werden, um für häufig in den Projekten verwendeten Teilfragmenten von OCL neue, starke Heuristiken zu entwerfen und implementieren.

- **Leichte Übertragbarkeit auf ähnliche Anwendungen.** Der Kern des Verfahren, die *Basisabbildung*, basiert auf klassischer Prädikatenlogik und damit einer recht universellen Sprache, die sich in vielen formalen Methoden wiederfindet. Somit ist der Hauptteil des Verfahren im entwickelten Rahmenwerk prinzipiell in viele formale Methoden übertragbar. Man beachte, daß auch die von uns untersuchten prädikativen Beschreibungen von Kollektionen in Verbindung mit der angegenen Heuristik *Kollektionen als Mengen* als Optimierungsmaßnahme in *allen* betrachteten Zielsprachen anwendbar!

- **Zusätzlich: Die Implementierung des Verfahrens.** Gerade im Umfeld des Software-Engineering gilt der empirische Grundsatz, daß eine Methode nur dann Eingang in die industrielle Praxis finden wird, wenn sie durch Werkzeuge unterstützt wird.

  Wir haben unser Verfahren in Java implementiert und über das Internet verfügbar gemacht. Es existiert somit eine *plattform-unabhängige* und *für jeden verfügbare* Implementierung. Die vollständige Neuentwicklung einer solchen Übersetzung von OCL in andere formale Methoden im softwaretechnischen Umfeld sollte durch die flexible Architektur des Verfahrens und zugehörigen Implementierung damit *nicht mehr notwendig* sein.

  Wir hoffen, durch unsere Arbeit zumindest in geringem Maße den Nutzen von OCL für die industrielle Praxis weiter zu erhöhen und damit die Verbreitung von OCL in der Industrie zu fördern.

Finale grande

Nun, was findet der aufmerksame und interessierte Leser am Schluß dieser Arbeit – nach viel Geduld, Durchhaltevermögen und den ganzen Strapazen und Qualen, die das Lesen dieser Arbeit dem Leser sicherlich abverlangte – schließlich vor?

In den meisten Fällen wird es zunächst vermutlich eine Enttäuschung sein, da wir nur in einem einzigen Falle – dem KeY-Projekt – eine wirkliche, ausführliche Lösung angegeben haben. Für die vielen möglichen anderen Anwendungen konnten wir natürlich im Rahmen dieser Arbeit keine detaillierte Antwort geben, wenngleich wir doch einen allgemeinen Rahmen anbieten, der mit relativ geringem Aufwand zu der gewünschten Antwort führen kann.

Es obliegt also dem Leser, dem Beispiel für das KeY-Projekt folgend durch Anpassung des Rahmenwerks und das Formulieren von Heuristiken eine für seine Anwendung geeignete, *gute Antwort* zu finden.

Wir schließen deshalb mit einigen Zeilen aus [Bre64, Epilog], die diese Situation trefflich beschreiben:

„Wir stehen selbst enttäuscht und sehn betroffen
Den Vorhang zu und alle Fragen offen.
...
Verehrtes Publikum, los, such dir selbst den Schluss
Es muss ein guter da sein, muss, muss, muss!

B. Brecht

Literaturverzeichnis

[AA00] Ambrosio Toval Álvarez and José Luis Fernández Alemán. Formally modeling UML and its evolution: A holistic approach. In S. Smith and C. Talcott, editors, *Proceedings, Formal Methods for Open Object-based Distributed Systems, Standford, USA*, pages 183–206. Kluwer, 2000.

[ABB+00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.

[ABB+02] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. Proceedings of FASE at ETAPS 02, to appear, 2002.

[BBS01] Thomas Baar, Bernhard Beckert, and Peter H. Schmitt. An extension of Dynamic Logic for modelling OCL's @*pre* operator. In *Proceedings, Fourth Andrei Ershov International Conference, Perspectives of System Informatics, Novosibirsk, Russia*, LNCS. Springer, 2001.

[Bec01] Bernhard Beckert. A Dynamic Logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041. Springer, 2001.

[BGH+98] Ruth Breu, Radu Gosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Towards a precise semantics for object-oriented modeling techniques. In J. Bosch and S. Mitchell, editors, *ECOOP Workshop, Jyväskylä, Finland*, LNCS 1357, pages 205–210. Springer, 1998.

[BKS01] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the object constraint language into first-order predicate logic. Submitted, available from `http://i12www.ira.uka.de/`, 2001.

[Boo91] G. Booch. *Object-Oriented Design – with Applications.* Benjamin/Cunnings, Reedwood City, CA, 1991.

[Bre64] Bertolt Brecht. *Der gute Mensch von Sezuan.* Edition Suhrkamp, Nr.73. Suhrkamp, Ffm., 1964. ISBN – 351810073.

[BRI01] Boldsoft, Rational Software Co., and IONA. Response to the UML 2.0 OCL RfP. Initial submission, August 2001.

[DH39] P. Bernays D. Hilbert. *Grundlagen der Mathemathik*. Springer Verlag, 1939.

[DKR00a] D. Distefano, J. Katoen, and A. Rensink. Towards model checking OCL, 2000.

[DKR00b] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a Temporal Logic for Object-Based Systems. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. Kluwer Academic Publishers, 2000.

[DR96] D. L. Dill and J. Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–34, April 1996.

[Fin00] Frank Finger. Design and implementation of a modular OCL compiler. Diplomarbeit, Technische Universität Dresden, Fakultät für Informatik, 2000.

[Fra99] Robert France. A problem-oriented analysis of basic UML static modeling concepts. In *Proceedings, Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, USA*, volume 34 (10) of *ACM SIGPLAN notices*. ACM Press, 1999.

[Gie00] Martin Giese. Integriertes automatisches und interaktives Beweisen: Die Kalkülebene. Diplomarbeit, Universität Karlsruhe (TH), Fakultät für Informatik, 2000.

[GR98] Martin Gogolla and Mark Richters. On constraints and queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language: Technical Aspects and Applications*, pages 109–121. Physica-Verlag, 1998.

[HCH+98] A. Hamie, F. Civello, J. Howse, S. Kent, and M. Mitchell. Reflections on the Object Constraint Language. In *Post Workshop Proceedings of UML98*. Springer, 1998.

[HDF00] Heinrich Hußmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings, International Conference on the Unified Modeling Language (UML), York, UK*, LNCS 1939, pages 278–293. Springer, 2000.

[HHK98] Ali Hamie, John Howse, and Stuart Kent. Interpreting the Object Constraint Language. In *Proceedings, Asia Pacific Conference in Software Engineering*. IEEE Press, July 1998.

[Jac92] I. Jacobsen. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, Addison-Wesley, Wokingham, UK, 1992.

[JR91] M. Blaha J. Rumbaugh. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[JR99] Grady Booch James Rumbaugh, Ivar Jacobsen. *The Unified Modeling Language Reference Manual*. The Addison-Wesley Object Technology Series. Addison-Wesley, Reading, MA, 1999.

[KC99] Soon-Kyeong Kim and David Carrington. Formalizing the UML class diagram using Object-Z. In R. France and B. Rumpe, editors, *Proceedings, Unified Modeling Language, Fort Collins, USA*, LNCS 1723, pages 83–98. Springer, 1999.

[Kri00] P. Krishnan. Consistency checks for UML. In *Proceedings, Asia Pacific Software Engineering Conference (APSEC)*, pages 162–169, 2000.

[Obj99a] Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Object Constraint Language Specification, Version 1.3*, June 1999. Chapter 7 in OMG Unified Modeling Language Specification, Version 1.3.

[Obj99b] Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.

[Obj01] Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Object Constraint Language Specification, Version 1.4, draft*, February 2001. Chapter 6 in OMG Unified Modeling Language Specification, Version 1.4, draft.

[RCA00] G. Reggio, M. Cerioli, and E. Astesiano. An algebraic semantics of UML supporting its multiview approach. In D. Heylen, A. Nijholt, and G. Scollo, editors, *Proceedings, AMiLP 2000*, number 16 in Twente Workshop on Language Technology. University of Twente, 2000.

[RG98] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In *Proceedings, Conceptual Modeling*, LNCS 1507, pages 449–464. Springer, 1998.

[RG00] Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. In *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000*, volume 1939 of *LNCS*. Springer, 2000. USE-System web page: `http://dustbin.informatik.uni-bremen.de/projects/USE/`.

[Sch01a] Peter H. Schmitt. Iterate logic. In Peter Schroeder-Heister, Robert Stärk, and Reinhard Kahle, editors, *Proof Theory in Computer Science, Proceedings, International Seminar, Dagstuhl Castle, Germany, October 2001*, volume 2183 of *LNCS*, pages 191–201. Springer, 2001.

[Sch01b] Peter H. Schmitt. A model theoretic semantics of OCL. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 43–57. Technical Report DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, 2001.

[Sch01c] Peter H. Schmitt. Nichtklassische Logiken. Vorlesungsskriptum, Universität Karlsruhe (TH), Juli 2001.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, 1999.

Beispiel einer XML-Testsuite

Wir wollen abschließend ein kleines Beispiel für eine XML-Datei angeben, die als Testsuite für die Testumgebung unserer Entwicklungsplattform dient. Wir haben der Einfachheit halber eine abgespeckte Version der Testsuite ausgewählt, die zur Generierung der Übersetzungen der Constraints aus dem Anwendungsbeispiel (siehe Abschnitt 2.4) verwendet wurde.

Das Format für Testsuites, sowie die angegebene Testsuite an sich, wurden mit besonderem Hinblick auf Selbsterklärung entworfen, um dem Benutzer die Möglichkeit zu geben, eine verständliche Bibliothek von Tests zu entwickeln. Wir verzichten daher auf detaillierte Erläuterungen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE TESTSUITE SYSTEM "testsuite.dtd">

<TESTSUITE>
  <NAME name="Vortragswelt-Demo"/>
  <DESCRIPTION>
     Diese Testsuite dient zur illustration der Testumgebung
     in meiner Studienarbeit.

     Die Testsuite umfasst einige Constraints ueber dem
     UML-Modell 'Vortragswelt'.
     Es werden sowohl Invarianten, als auch Vor-/Nachbedingungen
     getestet.

     Die reine Basisabbildung wird verwendet, wenn die Option
     'translation-control-class' den Wert
     'de. ... .AlwaysDefaultTranslationControl' annimmt.
     Wir stattdessen der Wert 'de. ... .SimpleTranslationControl'
     verwendet oder kein Wert spezifiziert, dann wird die
     optimierte Version der Abbildung benutzt.
  </DESCRIPTION>

  <XMI-MODEL file="vortragswelt.xml"/>

  <!-- ====================  Invarianten ====================== -->
  <TESTCASE>
    <NAME name="Invariante-1-Basic"/>
    <DESCRIPTION>
       Ein Pruefer wird von keinem Referenten geprueft, der von
       diesem Pruefer bewertet wurde.
    </DESCRIPTION>
    <CONSTRAINT>
```

```
    context Referent inv:
      self.pruefling->notEmpty implies
        not Referent.allInstances->exists(r|
            r.pruefling->includes(self) and
            self.pruefling->includes(r))
 </CONSTRAINT>
  <OPTION name="constrainttype" value="inv"/>
  <OPTION name="translation-control-class"
          value="de.uka.ilkd.key.ocl.ocldltranslator.
                 AlwaysDefaultTranslationControl"/>
</TESTCASE>


<TESTCASE>
  <NAME name="Invariante-2-Optimized"/>
  <DESCRIPTION>
     Die Pruefungsergebnisse eines Prueflings muessen sich mit
     der Zeit immer weiter verbessern, oder, falls es 30 Punkte
     oder mehr betraegt, nicht mehr unter 30 Punkte fallen.
  </DESCRIPTION>
  <CONSTRAINT>
    context Referent inv:
      self.pruefungsErgebnis_pruefer->forAll(pe1,pe2|
        pe1.datum.liegtVor(pe2.datum) implies
           (pe2.bewertung > pe1.bewertung or
            pe2.bewertung >= 30))
 </CONSTRAINT>
  <OPTION name="constrainttype" value="inv"/>
  <OPTION name="translation-control-class"
          value="de.uka.ilkd.key.ocl.ocldltranslator.
                 SimpleTranslationControl"/>
</TESTCASE>

<!-- ==============  Vor- und Nachbedingungen =============== -->

<TESTCASE>
  <NAME name="Pre/Postconstraint-1-Optimized"/>
  <DESCRIPTION>
     Nach dem Aufruf der Methode gilt das Ereignis als
     ,,begonnen'' und es wird die Folie mit der Foliennummer 1
     gezeigt.

     Die Option 'diamond-form' mit dem Wert 'true' sorgt dafuer,
     dass eine DL-Formelm mittels einem Diamant-Operator gebildet
     wird.
  </DESCRIPTION>
  <CONSTRAINT>
     context FolienVortragsEreignis::beginnen()
     pre: self.status = self.status.ruhend
     post: self.status = self.status.begonnen
     post: self.aktuelleFolie->notEmpty and
           self.aktuelleFolie.folienInfo->exists(fi|
             fi.folienVortrag = self.vortrag and fi.position = 1)
  </CONSTRAINT>
```

```
    <OPTION name="constrainttype" value="pre-post"/>
    <OPTION name="diamond-form" value="true"/>
  </TESTCASE>

</TESTSUITE>
```

Man beachte, daß bei der Ausführung dieser Testsuite alle enthaltenen Informationen – insbesondere die Kommentare – in die Ergebnisdatei des Test geschrieben werden, wodurch im folgenden angeführte, sehr schön lesbare und dokumentierte Testergebnis entsteht. Es geht uns hierbei nicht darum, ein detailliertes Verständnis zu vermitteln, wir wollen vielmehr einen Eindruck vermitteln, was die unsere Testumgebung zu leisten vermag.

```
------------------------------------------------------------------------------
--------------------    Executing a test suite ....     -------------------
------------------------------------------------------------------------------


Executing test suite in file : appendix-demo.xml
Testsuite name : Vortragswelt-Demo

Description :

     Diese Testsuite dient zur illustration der Testumgebung
     in meiner Studienarbeit.

     Die Testsuite umfasst einige Constraints ueber dem
     UML-Modell 'Vortragswelt'.
     Es werden sowohl Invarianten, als auch Vor-/Nachbedingungen
     getestet.

     Die reine Basisabbildung wird verwendet, wenn die Option
     'translation-control-class' den Wert
     'de. ... .AlwaysDefaultTranslationControl' annimmt.
     Wir stattdessen der Wert 'de. ... .SimpleTranslationControl'
     verwendet oder kein Wert spezifiziert, dann wird die
     optimierte Version der Abbildung benutzt.


XMI model file : vortragswelt.xml

------------------------------------------------------------------------------

Executing the single test cases in the test suite ...

------------------------------------------------------------------------------
Executing next test case ...
------------------------------------------------------------------------------
Testcase name : Invariante-1-Basic

Testcase description :

       Ein Pruefer wird von keinem Referenten geprueft, der von diesem
       Pruefer bewertet wurde.


- Using a special translation control object of class : de.uka.ilkd.key.ocl.
ocldltranslator.AlwaysDefaultTranslationControl !

Constraint to be translated :
```

```
context Referent inv:
  self.pruefling->notEmpty implies
    not Referent.allInstances->exists(r|
        r.pruefling->includes(self) and
        self.pruefling->includes(r))


Translation result :


 ?
all self:Referent.(all Referent1:Referent.Set_Of_Referent::contains(Set_Of_
Referent::allInstancesOfReferent,Referent1) -> ex Referent0:Referent.Set_
Of_Referent::contains(Referent::pruefling(self),Referent0) -> !ex Referent2:
Referent.(Set_Of_Referent::contains(Set_Of_Referent::allInstancesOfReferent,
Referent2) & Set_Of_Referent::contains(Referent::pruefling(Referent2),self)
 & Set_Of_Referent::contains(Referent::pruefling(self),Referent2)))



--------------------------------------------------------------------------
Executing next test case ...
--------------------------------------------------------------------------
Testcase name : Invariante-2-Optimized

Testcase description :

       Die Pruefungsergebnisse eines Prueflings muessen sich mit der Zeit
       immer weiter verbessern, oder, falls es 30 Punkte oder mehr betraegt,
       nicht mehr unter 30 Punkte fallen.


- Using a special translation control object of class : de.uka.ilkd.key.ocl.
ocldltranslator.SimpleTranslationControl !

Constraint to be translated :

context Referent inv:
  self.pruefungsErgebnis_pruefer->forAll(pe1,pe2|
      pe1.datum.liegtVor(pe2.datum) implies
         (pe2.bewertung > pe1.bewertung or
          pe2.bewertung >= 30))


Translation result :


 ?
all self:Referent.all PruefungsErgebnis0:PruefungsErgebnis.(Set_Of_
PruefungsErgebnis::contains(Referent::pruefungsErgebnis_pruefer(self),
PruefungsErgebnis0) -> all PruefungsErgebnis01:PruefungsErgebnis.(Set_Of_
PruefungsErgebnis::contains(Referent::pruefungsErgebnis_pruefer(self),
PruefungsErgebnis01) -> Datum::liegtVor(PruefungsErgebnis::
datum(PruefungsErgebnis0),PruefungsErgebnis::datum(PruefungsErgebnis01)) ->
 Integer::gt(PruefungsErgebnis::bewertung(PruefungsErgebnis01),
PruefungsErgebnis::bewertung(PruefungsErgebnis0)) |
Integer::geq(PruefungsErgebnis::bewertung(PruefungsErgebnis01),
Integer::0(Integer::3(Integer::#)))))



--------------------------------------------------------------------------
```

```
Executing next test case ...
--------------------------------------------------------------------------------
Testcase name : Pre/Postconstraint-1-Optimized

Testcase description :

       Nach dem Aufruf der Methode gilt das Ereignis als ,,begonnen''
       und es wird die Folie mit der Foliennummer 1 gezeigt.

       Die Option 'diamond-form' mit dem Wert 'true' sorgt dafuer, dass
       eine DL-Formelm mittels einem Diamant-Operator gebildet wird.


- Translation option: Using diamond form for result formula !

Constraint to be translated :

       context FolienVortragsEreignis::beginnen()
       pre: self.status = self.status.ruhend
       post: self.status = self.status.begonnen
       post: self.aktuelleFolie->notEmpty and
             self.aktuelleFolie.folienInfo->exists(fi|
               fi.folienVortrag = self.vortrag and fi.position = 1)


Translation result :


 ?
all self:FolienVortragsEreignis.(FolienVortragsEreignis::status(self) =
Status::ruhend(FolienVortragsEreignis::status(self)) -> <{
  self.beginnen ();
}>(FolienVortragsEreignis::status(self) =
Status::begonnen(FolienVortragsEreignis::status(self)) &
ex Folie0:Folie.Folie0 = FolienVortragsEreignis::aktuelleFolie(self) &
ex FolienInfo0:FolienInfo.(Set_Of_FolienInfo::contains(
Folie::folienInfo(FolienVortragsEreignis::aktuelleFolie(self)),FolienInfo0)
& FolienInfo::folienVortrag(FolienInfo0) =
FolienVortragsEreignis::vortrag(self) & FolienInfo::position(FolienInfo0) =
Integer::1(Integer::#))))
```