Bauhaus-Universität Weimar
Fakultät Bauingenieurwesen
Professur Informations- und Wissensverarbeitung

Contraint-basierte Konsistenzprüfungen in dynamischen Bauwerksmodellierumgebungen

Diplomarbeit

vorgelegt von
cand.-Ing. André Borrmann
geb. am 17. Februar 1978
in Jena

Erstprüfer: Prof. Dr.-Ing. Reinhard Hübler
Zweitprüfer: Dr.-Ing. Thomas Hauschild

Ausgabedatum: 01.06.2003
Abgabedatum: 15.09.2003

Danksagung

Mein Dank gilt in erster Linie meinen Eltern, die mir durch ihre stetige finanzielle, aber vor allem auch ideelle Unterstützung das Studium ermöglicht haben, das mit dieser Diplomarbeit seinen Abschluss findet.

Ganz besonders möchte ich mich auch bei Thomas Hauschild bedanken, von dem ich im Studium und während meiner Tätigkeit als studentischer Mitarbeiter viel gelernt habe und der mich als Betreuer dieser Diplomarbeit mit zahlreichen Hinweisen und Ratschlägen unterstützt hat.

Ich danke weiterhin Julie Rousset, die mir während der Diplombearbeitung zur Seite stand und die Gestaltung dieser Arbeit übernommen hat, sowie Robert Schmidt und Martin Lötzsch für die Hilfe beim Beschaffen von Literatur.

Schließlich danke ich Katrin Wender für die fruchtbare Zusammenarbeit, das gründliche Durchsehen dieser Arbeit und die konstruktive Kritik.

Inhalt

1 Einleitung

1.1 Ausgangspunkt

Die Bauplanung ist durch eine große Zahl stark spezialisierter Beteiligter aus unterschiedlichen Fachdisziplinen gekennzeichnet. Immer kürzer werdende Planungszeiträume werden die Zahl der gleichzeitig im Projekt involvierten Fachplanern weiter erhöhen. Aus der hohen Komplexität des Planungsgegenstands *Bauwerk* ergibt sich eine Vielzahl von Abhängigkeiten zwischen den Planungsleistungen der einzelnen Beteiligten.

Diese zunehmende Verzahnung der Planungsleistungen führt zur Notwendigkeit eines hohen Maßes an Kooperation zwischen den Beteiligten. Eine wesentliche Grundlage hierfür bilden geeignete Formen von Kommunikation, die das zu planende Bauwerk bzw. die zu erbringenden Planungsleistungen zum Gegenstand hat. In den letzten Jahren wurden vielfältige Applikationen entwickelt, die fachspezifische Teilprozesse der Planung auf sehr hohem Niveau unterstützen. Jedoch mangelt es nach wie vor an den für die kooperative Zusammenarbeit von Fachplanern verschiedener Fachbereiche notwendigen Möglichkeiten des Datenaustauschs auf hohem semantischen Niveau. Hannus prägte für diesen Zustand die Beschreibung „Islands of Automation“. [Hannus98]

Die mangelnde Unterstützung des Datenaustauschs äußert sich u.a. darin, dass Informationen größtenteils immer noch in Papierform ausgetauscht und manuell in das jeweilige Zielsystem eingepflegt werden. Zur Erhöhung der Effizienz der Arbeit der an der Planung Beteiligten und zur Vermeidung von Planungsfehlern ist eine durchgängig computergestützte Erhebung, Be- bzw. Verarbeitung und Weitergabe aller im Planungsprozess anfallenden Daten unabdingbar.

Zwar beginnt sich in der Praxis sich der Einsatz von Dokumentenverwaltungssystemen durchzusetzen, die dem Austausch und der zentralen Verwaltung von Dateien dienen. Die Granularität des Zugriffs auf einzelne Planungsinformationen ist bei dateibasierten Lösungen jedoch sehr grob, woraus sich fundamentale Schwächen derartiger Systeme hinsichtlich des Änderungsmanagements und der Unterstützung synchroner Arbeitsphasen ergeben.

Ein Ansatz mit weitaus größerem Potential zur Realisierung einer adäquaten Computerunterstützung des kooperativen Planungsprozesses ist die Integration aller verwendeten Applikationen durch die Verwendung eines gemeinsamen Informationsraums. Eines der wesentlichen Probleme bei der Umsetzung eines solchen

Informationsraums liegt in der Schwierigkeit, ein gemeinsames, für alle Fachdisziplinen gültiges Datenmodell zu finden.

Die Ergebnisse der Forschungsarbeit der letzten Jahre haben gezeigt, dass die Nutzung verteilter, objektorientierter und partitionierter Bauwerksmodelle einen geeigneten Ansatz zur Integration der bei der Planung, Nutzung und Instandhaltung von Bauwerken anfallenden Daten darstellt. Durch den Einsatz eines dynamisch adaptierbaren Bauwerkmodells kann zudem die wegen des langen Lebenszyklus und des Unikatcharakters von Bauwerken notwendige Flexibilität des Datenmodells gewährleistet werden. [Hübler02]

1.2 Motivation, Gegenstand und Zielsetzung dieser Arbeit

Ziel dieser Arbeit ist es, ein Konzept für die Erweiterung der rechnerinternen Repräsentation von Bauwerksmodellen um Zwangs- bzw. Nebenbedingungen (sog. *Constraints*) zu entwickeln. Diese Bedingungen sollen durch die Verknüpfung von Begrifflichkeiten der domänenspezifischen Taxonomie mit mathematischen und logischen Operatoren sowie Zahlen und Zeichenketten ausgedrückt werden können. Sie müssen dem Digitalen Bauwerksmodell zur Laufzeit des verwaltenden Systems hinzugefügt bzw. von ihm entfernt werden können.

Diese Arbeit wird die verschiedenen Möglichkeiten der Anwendung von Constraints in Bauwerksmodellen untersuchen. Durch das Prüfen von Instanzen der Klassen eines Domänenmodells auf Einhaltung der für sie gültigen Zwangsbedingungen wird es möglich sein, Aussagen über die Zulässigkeit des Zustands zu treffen, in dem sich diese Instanzen befinden. Um das Einhalten der Bedingungen durch das Informationssystem prüfen lassen zu können, muss bei deren Definition eine formale Sprache verwendet werden. Ein wichtiger Teil der Aufgabenstellung war es, eine für den Einsatz in Bauwerksmodellierumgebungen geeignete Sprache mit der nötigen Ausdrucksfähigkeit zu finden. Hierzu wurde die Eignung der *Object Constraint Language* (OCL) untersucht, die Teil des Industriestandards *Unified Modeling Language* (UML) ist. Wegen der dabei festgestellten Unterschiede im zugrunde liegende Metamodell musste ein Dialekt dieser Sprache mit leicht verändertem Sprachumfang entwickelt werden.

Zum Nachweis der Tragfähigkeit des entwickelten Konzepts wurde im Zuge dieser Arbeit ein Constraint-Modul für das im SFB 524[1] im Einsatz befindliche Modellverwaltungssystem prototypisch implementiert. Die Funktionalität dieses Moduls besteht u.a. in der Möglichkeit zum Prüfen von Instanzen des Domänenmodells auf die Einhaltung der für sie geltenden Bedingungen. Vorraussetzung hierfür ist die Fähigkeit des Interpretierens von Ausdrücken, die in der Constraint-Sprache formuliert wurden.

---

[1] Diese Arbeit ist ein Beitrag zur Forschungsarbeit im Teilbereich D3 „Digitales Bauwerksmodell als Grundlage der Prozessintegration“ des Sonderforschungsbereiches 524 „Werkstoffe und Konstruktionen für die Revitalisierung von Bauwerken“ an der Bauhaus-Universität Weimar, gefördert durch die Deutsche Forschungsgemeinschaft (DFG).

1.3 Kapitelübersicht

In Kapitel 2 wird ein Überblick über den derzeit aktuellen Forschungsstand im Bereich der integrativen Datenhaltung unter Nutzung von verteilten, dynamischen Bauwerksmodellen gegeben. Das dabei vorgestellte Informationssystem bildet die technische Grundlage für die Umsetzung der Funktionalitäten zur Definition und Prüfung von Zwangsbedingungen in Bauwerksmodellen. In Kapitel 3 folgt ein kurzer Exkurs, der die unterschiedliche Bedeutung des Begriffs *Constraint* und den Entwicklungsstand der damit verbundenen Techniken in verschiedenen Teilbereichen der Informatik behandelt.

Kapitel 4 wird das im Zuge dieser Arbeit entwickelte Konzept zum Einsatz von Constraints in Digitalen Bauwerksmodellen vorstellen und die damit verbundenen Möglichkeiten aufzeigen. Für die Formulierung von Constraint-Ausdrücken ist die Verwendung einer formalen Sprache notwendig. Als Kandidat hierfür wird in Kapitel 5 die standardisierte Constraint-Sprache OCL vorgestellt und die Elemente ihres Sprachumfangs detailliert beschrieben. In Kapitel 6 wird daraufhin untersucht, inwieweit sich OCL als Constraint-Sprache für den Einsatz in Modellverwaltungssystemen eignet, die auf dem AKO-Metamodell beruhen.

Auf Grundlage der dabei gewonnen Erkenntnissen wurde die Constraint-Sprache CML entwickelt, deren Sprachumfang und Grammatik in Kapitel 7 vorgestellt wird. Die Umsetzung des Constraint-Moduls, darunter die Implementierung des CML-Interpreters und dessen Integration in das Modellverwaltungssystem, wird in Kapitel 8 diskutiert. Kapitel 9 wird die im Rahmen dieser Arbeit gewonnnen Erkenntnisse zusammenfassen und einen Ausblick auf die weitere Forschungsarbeit in diesem Kontext geben.

2 Ansatz für ein integrierendes Informationssystem

2.1 Überblick

Ziel der Nutzung einer integrierten Entwurfsumgebung ist das parallele Arbeiten der an der Bauwerksplanung Beteiligten. Dies ist mit dem herkömmlichen dateibasierten Austausch von Planungsdaten nur durch Inkaufnahme eines bedeutenden zusätzlichen Aufwands hinsichtlich des Änderungs- und Konsistenzmanagements möglich. Die Ursache hierfür liegt im wesentlichen in den äußert grobgranularen Zugriffsmöglichkeiten auf die in Dateien gespeicherten Planungsinformationen.

Ein Ansatz für eine bessere Unterstützung des parallelen Arbeitens und der Kooperation der Planungsbeteiligten liegt in der Verwaltung aller anfallenden Planungsdaten in einem gemeinsamen Informationsraum. Die wesentliche Schwierigkeit bei der Umsetzung eines solchen Informationsraums besteht im Finden geeigneter Datenstrukturen für die rechnerinterne Abbildung eines Bauwerksmodells, das den verschiedenen Sichten aller beteiligten Fachdisziplinen gerecht wird.

Dieses Kapitel wird die Anforderungen an ein geeignetes Informationssystem, die sich daraus ergebenden Probleme und den im Teilprojekt D3 des SFB 524 verfolgten Lösungsansatz vorstellen.

2.2 Allgemeine Anforderungen

An der Planung, der Realisierung und dem Betreiben von Bauwerken ist in der Regel eine große Zahl von Beteiligten aus unterschiedlichsten Fachdisziplinen und verschiedenen Unternehmen beteiligt. Das Informationssystem soll einerseits als Integrationsplattform fungieren und den reibungslosen Datenaustausch zwischen den Beteiligten gestatten, diese andererseits aber nicht in ihrer Arbeit einschränken oder behindern.

Ein geeignetes Informationssystem muss weiterhin alle Phasen des Lebenszyklus eines Bauwerks durchgängig unterstützen, angefangen bei dessen Entwurf, über die Ausführungsphase bis zur Gebäudebewirtschaftung (Facility Management) und schließlich dem umweltgerechten Rückbau. Die Anforderungen der einzelnen Phasen variieren stark. Während beispielsweise der frühe Entwurf durch vage Informationen, vielfältige Variantenbildung und keine bzw. wenig Detaillierung gekennzeichnet ist, sind bei der Ausführungsplanung eine große Menge von Beteiligten verschiedener Domänen involviert, deren gemeinsames Ziel ein präzises

Abbild des zu errichtenden Bauwerks ist. Das zieht eine intensive Kooperation mit hohem Kommunikationsbedarf nach sich. In der langen Phase der Nutzung eines Bauwerks ist die Zahl der Beteiligten zwar vergleichsweise gering, steht jedoch eine Umnutzung bzw. Revitalisierung an, werden für das Informationssystem bzw. den von ihm verwaltenden Datenbestand erneut die hohen Anforderungen der kooperativen Planung aktuell. Da in Planungsphasen im allgemeinen die höchsten Ansprüche an das Informationssystem gestellt werden, sollen diese bei den folgenden Ausführungen als maßgeblich betrachtet werden.

2.3 Digitale Bauwerksmodelle

Das Digitale Bauwerksmodell stellt als computerinterne Abbildung des Betrachtungsgegenstandes Bauwerk ein Ordnungsschema für die beim Planungsprozess anfallenden, das bauliche Objekt beschreibenden Daten bereit und dient so als Verständigungsplattform für die an der Planung Beteiligten.

2.3.1 Objektorientierte Bauwerksmodelle

Die derzeit in der Praxis übliche zeichnungs- bzw. geometriebasierte Modellierung von in der Planung befindlichen Bauwerken weist immense Schwächen hinsichtlich der verfügbaren Ausdrucksmächtigkeit auf: Die in der Bauplanung auftretenden Entitäten sind in der Regel nicht nur durch ihre räumlichen Ausdehnungen charakterisiert, sondern weisen weitere für den Planungsprozess relevante Eigenschaften auf, die sich u.a. aus ihrer physikalischen Beschaffenheit ableiten. Die Zuordnung bzw. Speicherung derartiger Daten ist meist nur unstrukturiert möglich, was eine Weiterverarbeitung durch andere Planungswerkzeuge auf dem notwendigen semantischen Niveau ausschließt.

Aus diesen Gründen wird von großen Teilen der Forschergemeinschaft seit längerem eine Bauwerksmodellierung nach dem objektorientierten Paradigma propagiert. [Hartmann00] Dieses bietet hilfreiche Mittel der Abstraktion wie Kapselung, Erbschaft und Aggregation, die zu einer hohen Ausdrucksstärke des Modells bei gleichzeitig handhabbarer Komplexität beitragen. Zudem gilt die objektorientierte Modellierung als nah verwandt mit der Art und Weise des menschlichen Denkens, was den Umgang mit objektorientierten Modellen erleichtert. [Booch99]

2.3.2 Partialmodelle

Wegen der zum Teil beträchtlichen Unterschiede zwischen den Sichten der involvierten Fachdisziplinen auf den Entwurfsgegenstand und damit auch zwischen den verwendeten Taxonomien gestaltet sich die Integration in einem einzigen gemeinsamen Bauwerksmodell als äußerst komplex. Zur Verminderung der Komplexität des resultierenden Modells hat es sich daher als sinnvoll erwiesen, zunächst für jede Domäne ein gesondertes, auf die jeweiligen Bedürfnisse zugeschnittenes Modell zu formulieren, das dementsprechend *Domänenmodell* genannt wird.

Da die Domänenmodelle letztlich nur einen Teil des gesamten Bauwerksmodells abbilden, werden sie auch *Partialmodelle* genannt. Die Domänenmodelle können, aber müssen nicht über einen Kern von gemeinsam genutzten Klassen verfügen. Beim verknüpfungsbasierten Ansatz wird die Kohärenz zwischen den Partialmodellen nicht durch einen gemeinsames Kernmodell (auch neutrales

Modell) abgebildet, sondern durch sogenannte Verknüpfungen realisiert, die die Integrität des Gesamtdatenbestandes sichern. [Willenbacher02]

2.3.3 Standardisierte Bauwerksmodelle

Internationale Initiativen, wie die ISO-Normierung 10303 *„Standard for the Exchange of Product Data“* (STEP) und die Industrieinitiative *International Alliance for Interoperability* (IAI), widmen sich der Standardisierung von Bauwerksmodellen und haben dabei bereits deutliche Fortschritte erzielt.

Im folgenden soll auf die von der IAI entwickelten *Industry Foundation Classes* (IFC) näher eingegangen werden, da sie den am weitesten entwickelten Standard darstellen und mittlerweile einen beachtlichen Verbreitungsgrad aufweisen. [IFC] Abbildung 2.1 zeigt einen Ausschnitt aus dem durch die IFC beschriebenen Produktmodell.

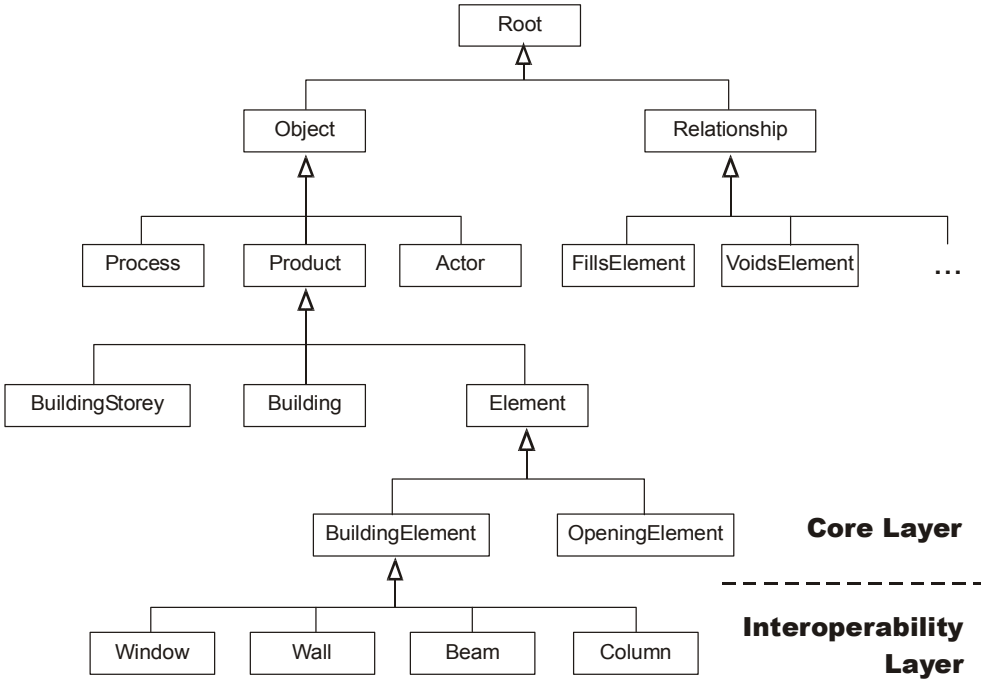

Abbildung 2.1: Ausschnitt aus dem IFC-Modell

Der IFC-Standard verfolgt zur Integration verschiedener Domänen das Konzept der Verwendung eines gemeinsamen Kerns, der solche Klassen beinhaltet, die in allen Domänen Verwendung finden. Die verschiedenen Schemata der Domänenschicht (engl. domain layer) verfügen hingegen über Klassen, die nur in der jeweiligen Domäne benötigt werden. Dazwischen wurde eine sogenannte Interoperabilitätsschicht (engl. interoperability layer) mit einer sehr großen Menge von Klassen angeordnet, die in den Taxonomien verschiedener Domänen auftauchen und daher zum domänenübergreifenden Austausch von Daten dienen können.

Trotz der erreichten Fortschritte weist der IFC-Standard einige fundamentale Schwächen auf. Zwar haben die berücksichtigten Domänen in der aktuellen Version

bereits eine beträchtliche Anzahl[1] erreicht, jedoch werden bei komplexeren Bauvorhaben in der Regel Spezialisten einbezogen, deren fachgebietsspezifische Sicht bei diesen allgemeinen Ansätzen unberücksichtigt bleiben. Zudem beschränkt sich die IAI-Initiative in ihren Standardisierungsbemühungen auf das vergleichsweise enge Spektrum von Bauwerkstypen aus dem nicht-industriellen Hochbau.

Ein anderes Problem standardisierter Bauwerksmodelle ist die fehlende Detaillierung in der Klassenhierarchie. So bleibt auch der Spezialisierungsgrad der IFC-Klassen relativ grob[2]. Die IFC sehen zwar grundsätzlich eine Erweiterbarkeit des Modells durch den Nutzer bzw. den Applikationsprogrammierer vor, dabei ist die Modellierungsmächtigkeit jedoch stark eingeschränkt, da Erweiterungen nicht nach objektorientierten Prinzipien vorgenommen werden können.

Trotz der bedeutenden Fortschritte bei der Standardisierung von Bauwerksmodellen muss konstatiert werden, dass ein allgemeines, für alle Bauwerkstypen und potentiell involvierten Fachdisziplinen gültiges, umfassendes und ausreichend differenziertes Modell nicht zu definieren ist. Ein Ansatz zur Lösung dieses Dilemmas besteht in der Verwendung (dynamisch) anpassbarer Bauwerksmodelle, wie sie im folgenden vorgestellt werden sollen.

2.3.4 Dynamische Bauwerksmodelle

Ein dynamisches Bauwerksmodell ist während seiner Nutzung nach objektorientierten Prinzipien erweiterbar. Es gibt eine Reihe gewichtiger Gründe, die dafür sprechen, ein dynamisch modifizierbares Bauwerks- bzw. Domänenmodell einzusetzen. Dies sind im einzelnen:

- Breites Spektrum von Bauwerkstypen. Es gibt ein breites Spektrum von zu planenden bzw. zu revitalisierenden Bauwerkstypen, die ein Bauwerksmodell abbilden können muss.
- Unikat-Charakter. Im allgemeinen sind die zu planenden Bauwerke Unikate bzw. sollen in vergleichsweise geringer Stückzahl errichtet werden. Damit verbunden ist die notwendige Anpassbarkeit des Bauwerksmodells an die spezifischen Bedürfnisse eines konkreten Projektes. Abhängig von der Bauweise sind mitunter keine oder nur sehr wenige standardisierte Bauteile verfügbar bzw. deren Verwendung gar nicht erwünscht.
- Langer Lebenszyklus. Verglichen mit den Produkten anderer Industriezweige durchläuft ein Bauwerk einen sehr langen Lebenszyklus. Dieser beginnt mir der Planung und der Errichtung, durchläuft eine lange Phase der Nutzung, unterbrochen von Revitalisierungs- und Umnutzungsphasen und endet mit der umweltgerechten Demontage und Entsorgung. Es gilt als nahezu unmöglich, ein für diesen langen und vielgestaltigen Lebenszyklus gültiges Bauwerksmodell im Vornherein zu definieren.

---

[1] In der Version 2x2 vom Mai 2003 werden die Domänen Architektur, Technische Gebäudeausrüstung, Strukturanalyse, Bauausführung, Brandschutz und Liegenschaftsverwaltung berücksichtigt.

[2] Beispielhaft seien hier die Klassen *Bauwerk* (engl. building) und *Träger* (engl. beam) genannt, für die im IFC-Standard keine Subklassen existieren.

- Individuelles Wissen. Durch Nutzung eines anpassbaren Bauwerksmodells kann die individuelle Erfahrung eines Planers und seine persönliche Sicht auf den Planungsgegenstand eingebracht werden. Da davon ausgegangen werden muss, dass während der Arbeit an einem Projekt sowohl die Erfahrung des Planers wächst, als auch dessen Sicht sich ändern kann, ist eine dynamische Modifizierbarkeit des Modells notwendig.

Bei der zu Beginn eines Projektes stattfindenden Konfiguration des Informationssystems kann eine Anpassung der verwendeten Domänenmodelle an die besonderen Spezifika eines konkreten Vorhabens stattfinden. Dabei können in den meisten Fällen standardisierte Produktmodelle (z.B. IFC) oder für bestimmte Bauwerkstypen vordefinierte Bauwerksmodelle als Basis genutzt und um spezielle Entitäten erweitert werden.

2.4 Architektur des Informationssystems

Die Architektur eines integrierenden Informationssystems muss den oben genannten Anforderungen an die Bauwerksmodellierung gerecht werden (Partitionierung, Dynamik) und zusätzliche Aspekte hinsichtlich des verteilten Zugriffs auf die Planungsdaten genügen.

2.4.1 Dynamische Modellverwaltungssysteme

Zur Verwaltung dynamischer Bauwerksmodelle und deren Ausprägungen kommen dynamische Modellverwaltungssysteme zum Einsatz. Diese sind technisch durch die Nutzung schema-dynamischer Datenbanksysteme oder Repositories realisierbar. Die Forschungsarbeit der letzten Jahre hat jedoch gezeigt, dass die Funktionalität derartiger Systeme für den Einsatz als Modellverwaltungssystem nicht hinreichend ist. Daher wurden umfangreiche Eigenimplementierungen vorgenommen. Diese basieren auf der sogenannten AKO-Schnittstelle, mit deren Hilfe zum einen objektorientierte Domänenmodelle erstellt und zur Laufzeit manipuliert werden können und zum anderen Zugriff auf Instanzen eines Domänenmodells möglich ist. [AKO97] [Kolbe98] Das implizit durch die AKO-Schnittstelle in der aktuellen Version von 2001 beschriebene Metamodell ist in Abbildung 2.2 dargestellt

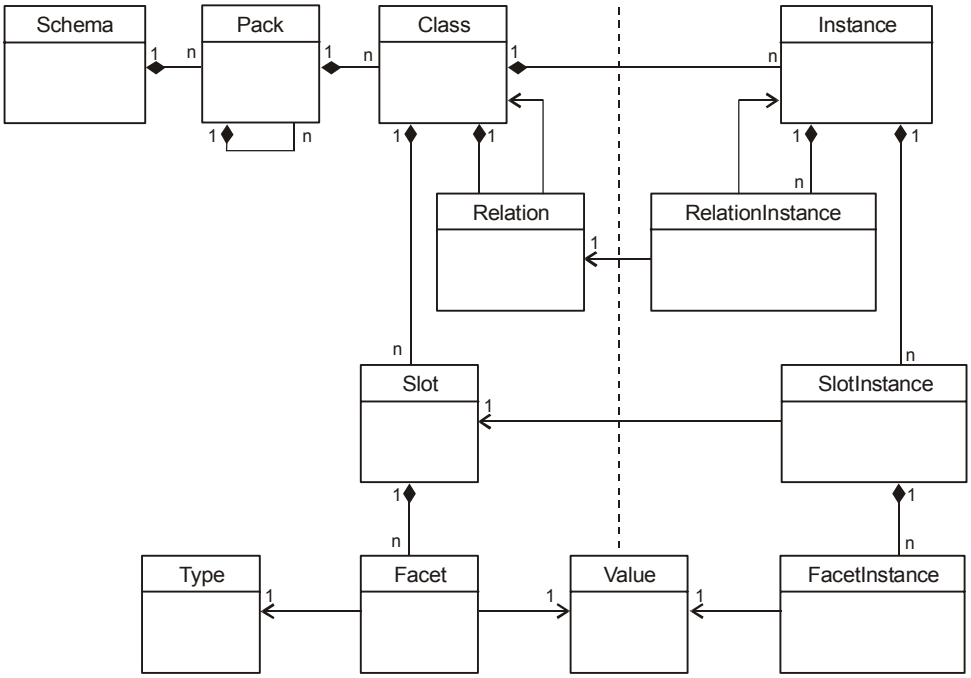

Abbildung 2.2: Das AKO-Metamodell in UML-Notation

Während die linke Seite der Abbildung diejenigen Metaklassen zeigt, die ein vom MVS verwaltetes Domänenmodell repräsentieren, sind auf der rechten Seite die Metaklassen dargestellt, die die konkreten Ausprägungen von einzelnen Klassen des Domänenmodells repräsentieren. Die Semantik der einzelnen Teile des Metamodells wird detailliert in Kapitel 6.3 behandelt.

Die AKO-API selbst beinhaltet Methoden zum Lesen und Setzen der Attribute und Assoziationsenden des dargestellten Metamodells, wobei die Semantik der Schnittstelle aus technologischen bzw. entwicklungsgeschichtlichen Gründen an verschiedenen Stellen von der des Metamodells abweicht. Im SFB 524 kommt eine Java-basierte Implementation der AKO-Schnittstelle zum Einsatz, die die Funktion eines domänenzentralen Modellverwaltungsservers übernimmt.

2.4.2 Hybride Topologie des Gesamtsystems

Ein Planungsteam setzt sich zumeist aus mehreren kleinen Einheiten (Planungsbüros) zusammen, deren Mitglieder ähnliche Aufgaben und Ziele innerhalb des Gesamtprojektes haben und die gleiche Sicht auf den Planungsgegenstand besitzen. Während die Beteiligten eines solchen Fachplanerbereiches im allgemeinen keiner räumlichen Trennung unterworfen sind, sind die geographischen Abstände zwischen den einzelnen Planungsbüros als potentiell sehr groß anzunehmen.

Eine zentrale Verwaltung aller Domänenmodelle und ihrer Instanzdaten scheidet u.a. wegen dem geringen Durchsatz und der hohen Latenz bei Datenübertragungen sowie der nicht garantierten Verfügbarkeit des Kommunikationsmediums in Weitverkehrsnetzen aus. Das im SFB 524 konzipierte System besitzt daher eine hybride Topologie, die zudem auf physischer Ebene die Partitionierung des Bauwerkmodells in Domänenmodelle wiederspiegelt. [Hauschild02] (Abbildung 2.3)

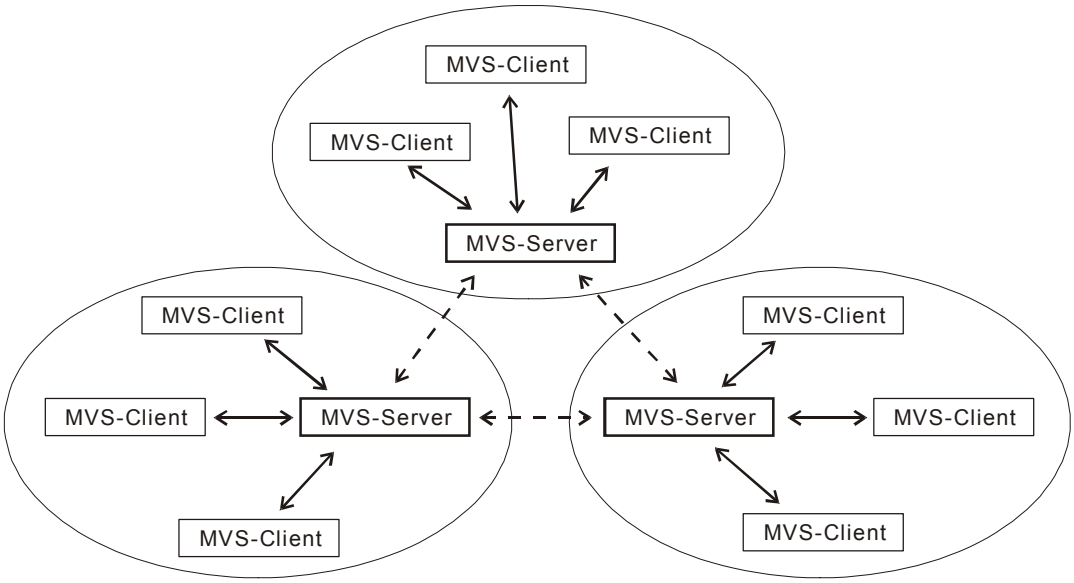

Abbildung 2.3: Hybride Architektur des konzipierten Informationssystems

Domänenintern werden die Planungsdaten zur Gewährleistung ihrer Konsistenz zentral gehalten: die Infrastruktur einer Domäne setzt sich aus einem Modellverwaltungsserver (auch Domänenmodellserver) und einer beliebigen Anzahl von Fachapplikationen zusammen, die als Clients fungieren und die Schnittstelle des Systems zu den einzelnen Fachplanern bilden.

Der MVS-Server verwaltet das Domänenmodell und dessen Instanzen und übernimmt dabei die klassischen Aufgaben der zentralisierten Datenhaltung: Gewährleistung des verteilten Zugriffs, Sicherung der Persistenz, Kontrolle der Nebenläufigkeit, Überwachung der Zugriffsrechte und ggf. Steuerung von Transaktionen. Technisch wird die Implementierung dieser Primärdienste beim MVS-Server des SFB 524 durch die Kombination eines CORBA[1]-ORBs mit diversen CORBA-Diensten und einer objektorientierten Datenbank umgesetzt.

Zur technischen Realisierung der Verknüpfungen zwischen den Domänenmodellen können auf Softwareagenten basierende Technologien eingesetzt werden. Sie erlauben eine vergleichsweise lose Kopplung, die für die Unabhängigkeit der einzelnen Teil des Informationssystems notwendig ist. [Willenbacher02]

2.4.3 Modi des verteilten Zugriffs

Für die Arbeit eines Fachplaners mit dem Informationssystem gibt es zwei verschiedene Modi, in den die Applikation auf den domänenzentralen Modellverwaltungsserver zugreifen kann: Der sogenannte *synchronisierte* oder auch *Online-Zugriff* sieht die ständige Kopplung des Datenbestandes der Clientapplikation mit dem des Modellverwaltungsservers vor. Im *Offline-Modus* werden die Daten von Fachapplikation und MVS-Server nur zu bestimmten, vom Nutzer festzulegenden Zeitpunkten abgeglichen.

Die Arbeit im sogenannten *Online*-Modus ist vor allem beim gleichzeitigen Zugriff mehrerer Fachplaner auf die selbe Teilmenge der Domänendaten (synchrone

[1] Common Object Request Broker Architecture

Kollaboration) sinnvoll und eine notwendige Vorraussetzung für die Bereitstellung von (relaxierten) WYSIWIS[1]-Funktionalitäten. [Hauschild00] [Borrmann02]

Für eine Reihe von im Entwurfsprozess auftretenden Szenarien, wie der Prüfung von Planungsvarianten und der Durchführung umfangreicher Berechnungen ist eine feste Kopplung der Datenbestände des Clients mit dem des MVS nicht wünschenswert. Anwendungsprogramme, die in solchen Szenarien zum Einsatz kommen, arbeiten daher im *Offline*-Modus. Dabei wird zunächst eine Kopie der betreffenden Daten beim Client erzeugt, die die Anwendungssoftware frei manipulieren kann. Nach Abschluss der Bearbeitung werden die veränderten bzw. neu erzeugten Daten in das MVS eingespeist.

Während diejenigen Inkonsistenzen, die infolge paralleler Bearbeitung entstehen können, beim Online-Zugriff von vornherein gänzlich vermieden werden, muss im Falle der Offline-Bearbeitung die Konsistenz durch ein in das MVS integriertes Versionsmanagement gesichert werden.

2.5 Zusammenfassung

In diesem Kapitel wurde das Konzept für die Umsetzung eines Informationssystems dargelegt, das der Integration der beim Planungsprozess anfallenden Daten dienen kann. Das Digitale Bauwerksmodell bildet die Basis der Integration. Die Modellierung nach dem objektorientierten Paradigma und die Partitionierung in Domänenmodelle verringert die Komplexität und erhöht die Handhabbarkeit des resultierenden Modells. Die exemplarische Betrachtung des IFC-Standards hat die Grenzen standardisierter Bauwerksmodelle aufgezeigt. Gründe für die Verwendung eines anpassbaren Bauwerksmodells sind u.a. das große Spektrum zu unterstützender Bauwerkstypen, der Unikatcharakter eines jeden Bauwerks und der Wunsch danach, die individuelle Erfahrung eines Planers einzubringen.

Das im SFB 524 konzipierte Informationssystem zur Verwaltung eines dynamischen, partitionierten Bauwerkmodells weist eine hybride Struktur auf und setzt sich domänenintern aus einem zentralen Modellverwaltungsserver und beliebig vielen Fachapplikationen als dessen Clients zusammen. Um die Dynamik des Modells umsetzen zu können, kommunizieren die Clients mit dem Server über eine Schnittstelle (AKO-API), die die Modifikation des Domänenmodells und den generischen Zugriff auf Instanzdaten erlaubt.

Das folgende Kapitel wird eine Einführung zum Begriff *Constraint* und zu seiner Verwendung in unterschiedlichen Teilgebieten der Informatik geben. Kapitel 4 wird darauf aufbauend die Möglichkeiten des Einsatzes von Constraints in digitalen Bauwerksmodellen beschreiben und die sich daraus ergebenden Vorteile erörtern. In Kapitel 8 wird schließlich eine Erweiterung des AKO-Metamodells vorgeschlagen, die die Aufnahme von Constraints in das Digitale Bauwerksmodell erlaubt.

[1] What You See Is What I See

3 Der Constraint-Begriff

3.1 Überblick

Der englischsprachige Begriff *Constraint* lässt sich ins Deutsche mit Nebenbedingung, Zwangsbedingung, Einschränkung oder Beschränkung übersetzen. In der Informatik besitzt er jedoch eine zusätzliche bzw. verfeinerte Bedeutung. Diese ist, wie häufig im wissenschaftlichen Sprachgebrauch, vom jeweiligen Kontext abhängig, in dem dieser Begriff Verwendung findet.

Zwei der hier wesentlichen Gebiete der Informatik, in denen der Begriff *Constraint* gebraucht wird, sind das der wissensbasierten Systeme und das der formalen Spezifikation von Softwaresystemen. Diese Kapitel wird einen Überblick über dort gängigen Definitionen für den Begriff *Constraint* und der damit verbundenen Konzepte geben und deren jeweilige Entwicklungsgeschichte skizzieren.

3.2 Constraints im Kontext wissensbasierter Systeme

3.2.1 Wissensbasierte Systeme

Wesentlichstes Merkmal wissensbasierter Systeme ist die klare Trennung zwischen der Repräsentation von Wissen in einer sogenannten Wissensbasis und der Anwendung dieses Wissens zur Lösung von Problemen (Wissensverarbeitung). Im Gegensatz zu anweisungsorientiert arbeitenden Softwaresystemen ist der Kontrollfluss in wissensbasierten Systemen nicht starr, sondern ergibt sich aus dem abgebildeten Wissen.

3.2.2 Begriffsdefinition

Die Begriffe *Constraint* und *Constraint-Problem* werden im Kontext wissensbasierter Systeme wie folgt formal charakterisiert: Ein Constraint ist ein Tripel von Name, Variable und Definition, wobei die Definition eine Relation zwischen Variablen beschreibt. Ein Constraint-Problem besteht aus einer Menge von Constraints, die durch gemeinsame Variablen verbunden sind und einer Anfangsbelegung von Variablen. Die Lösung des Constraint-Problems ist eine maximale Einschränkung des Wertebereichs aller Variablen. [Voss88]

[Puppe88]: „Constraints dienen zur Repräsentation von Relationen, d.h. von irgendwelchen Beziehungen zwischen Variablen. Constraints eigenen sich besonders zur Darstellung von lokalen Randbedingungen, die die Problemlösung in jedem Fall erfüllen muss, ohne dass damit eine konkrete Problemlösung festgelegt

wird. [...] Constraints eignen sich [...] vorzüglich zur quantitativen oder qualitativen Modellierung von Systemen und physikalischen Zusammenhängen."

Constraints gelten neben Regeln, Frames und Logik als eine der Grundtechniken der Wissensrepräsentation. Der wesentliche Unterschied zwischen Regeln und Constraints besteht darin, dass Regeln gerichtete Zusammenhänge ausdrücken (aus A folgt B), während Constraints ungerichtete Abhängigkeiten zwischen Variablen repräsentieren, die nach jeder Variable hin aufgelöst werden können (zum Beispiel A = B * C).

Durch die Verwendung gemeinsamer Variablen können einzelne lokale Constraints zu einem Constraint-Netz verbunden werden. Existieren äußere Randbedingungen, können diese durch das Constraint-Netz propagiert werden.

Wenn die Semantik der einzelnen Constraints als mathematische Gleichung abbildbar ist, lässt sich das Constraint-Netz als ein Gleichungssystem betrachten. Das Ausrechnen dieses Gleichungssystems entspricht der Lösung des Constraint-Problems. In der Regel sind Constraints jedoch nicht nur auf Gleichungen beschränkt, sondern beinhalten auch Ungleichungen und nichtnumerische Zusammenhänge (beispielsweise boolesche Verknüpfungen).

3.2.3 Propagierungsalgorithmen

Constraints können mit Hilfe von Datenbanken, als Regeln oder als beliebige Programme implementiert werden. Dem Algorithmus zur Constraint-Propagierung wird als Eingabe ein Constraint-Netz, sowie eine Teilbelegung von Variablen mit Werten übergeben. Ergebnis ist eine mit den Constraints konsistente Wertzuweisung an die nicht-vorbelegten Variablen.

Wenn alle Variablen genau einen Wert annehmen, liegt eine eindeutige Lösung vor. Werden Variablen Wertebereiche bzw. -mengen zugewiesen, existieren verschiedene Lösungen. Die Zuweisung der leeren Menge bedeutet, dass ein mit allen Constraints konsistenter Zustand nicht herstellbar ist, d.h. keine Lösung besteht.

Die Propagierung besteht im wesentlichen darin, dass die Beschränkung der Wertemenge einer Variablen über die mit ihr verbundenen Constraints an andere Variablen weitergegeben wird, bis keine weitere Einschränkung des Wertebereichs irgendeiner Variablen mehr möglich ist. Propagierungsalgorithmen können danach unterschieden werden, was entlang einer Variablen propagiert werden kann:

- nur feste Werte, z.B. $X = 5$
- Wertemengen, z.B. $X \in \{3, 4, 5, 6\}$
- symbolische Ausdrücke, z.B. $X = 2*Y$

3.2.4 Entwicklungsgeschichte

Ende der siebziger Jahre begann man, Constraints in Werkzeugen und Programmiersprachen verfügbar zu machen, um dadurch Constraint-Probleme flexibel abbilden und lösen zu können. Diese Systeme waren vor allem für interaktive graphische Anwendungen gedacht. Die ersten Constraint-basierten Systeme waren zunächst auf ein spezifisches Anwendungsgebiet festgelegt. Beispielhaft sei hier *EL*, ein System zur Simulation elektrischer Schaltkreise genannt. [Stallman77] Kurz darauf folgte die Entwicklung der ersten anwendungsneutralen Constraint-Sprache CONSTRAINT [Sussman80].

Mitte der achtziger Jahre entstand die Constraint-Logikprogrammierung (CLP) als natürliche Fusion der zwei deklarativen Paradigmen Constraint-Lösen und Logikprogrammierung. Die ersten CLP-Sprachen waren CLP(R), CHIP und Prolog III. Jaffar und Lassez stellten 1987 ein vereinheitlichtes Modell für eine formal-logische Beschreibung von CLP vor, das sogenannte CLP-Schema. [Jaffar87]

In den neunziger Jahren wurden vorwiegend von der Forschung im Bereich der Künstlichen Intelligenz Lösungsalgorithmen für eine Reihe spezieller Constraint-Systeme entwickelt, unter anderem für nicht-lineare Gleichungssysteme über reelle Zahlen, Systeme der Booleschen Algebra, für das Rechnen mit endlichen Wertebereichen und die Manipulation von Zeichenketten. [Frühwirth97]

Die aktuelle Forschung beschäftigt sich mit der Lösung anwendungsrelevanter Klassen von Constraint-Problemen, hier vor allem der Suche nach Algorithmen für das Lösen zeitlicher, geometrischer und räumlicher Constraints. In den letzten Jahren wurde die Relevanz der Verknüpfung der Constraint-Programmierung mit dem Paradigma der Objektorientierung erkannt. Wesentliche Fortschritte wurden im Bereich der Constraint-basierten Datenbanken erzielt. [Kuper00]

3.3 Constraints im Kontext der Spezifikation von Softwaresystemen

3.3.1 Formale Spezifikation von Softwaresystemen

Auf dem Gebiet der formalen Spezifikation von Softwaresystemen besitzt der Begriff *Constraint* eine etwas abweichende Bedeutung. Constraints dienen hier dazu, die für ein System gültigen Zustände und Zustandsübergänge und damit dessen Funktionalität formal zu beschreiben.

Besondere Signifikanz hat das Konzept der Constraints in den Prozessen der Objektorientierten Analyse (OOA) und dem Objektorientierten Design (OOD) erlangt. Constraints werden in der Regel der visuellen Notation der dabei verwendeten Modelle hinzugefügt, um zusätzliche Informationen über die Modellelemente und deren wechselseitigen Beziehungen auszudrücken. Letztlich erhöhen sie auf diese Weise die Präzision des Modells.

3.3.2 Begriffsdefinition

[Warmer99] definiert den Begriff *Constraint* wie folgt:

> „A constraint is a restriction on one or more values of (part of) an object-oriented model or system."

Im allgemeinen werden im Kontext der formalen Spezifikation drei verschiedene Typen von *Constraints* unterschieden:

- Vorbedingungen (engl. preconditions),
- Nachbedingungen (engl. postconditions) und
- Invariante Bedingungen (engl. invariants).

Während Vor- und Nachbedingungen den Zustand des Systems unmittelbar vor bzw. nach dem Ausführen einer Operation beschreiben, müssen invariante Bedingungen zu jedem Zeitpunkt eingehalten sein.

Die Verwendung von Vor- und Nachbedingungen ist eine effektive Art, die Semantik von Operationen bzw. Methoden zu spezifizieren. Das Prinzip hinter der Nutzung von Vor- und Nachbedingungen zur Festlegung von erwartetem Verhalten wird auch als „Design by contract“ bezeichnet. Dieses Prinzip kann im Kontext jedweder objektorientierten Entwicklungsmethode Anwendung finden und wird häufig als Grundlage für ein gutes *Software Engineering* gesehen.

Zusätzlich zu Vor- und Nachbedingungen definiert Meyer eine dritte Art von Constraints: die invarianten Bedingung. Invariante Bedingungen müssen zu jeder Zeit erfüllt sein. Sie sind immer an Klassen, Typen oder Interfaces gekoppelt und müssen von allen Instanzen dieser Klasse, des Typs oder des Interfaces erfüllt werden. [Meyer88]

3.3.3 Entwicklungsgeschichte

Meyer nutzte erstmalig die Semantik von Constraints in der objektorientierten Modellierung, benutzte dafür jedoch den Begriff *Behauptung* (engl. assertion*)*. Das Konzept der *Assertions* wurde erstmalig in der Programmiersprache *Eiffel* implementiert, wo es als Basis für die Umsetzung von „Design by contract“ dient.

Bei der von Walden und Nerson entwickelten objektorientierten Modelliermethode *Business Object Notation* (BON) spielt das Prinzip des *Design By Contract* ebenfalls eine zentrale Rolle. Wesentlicher Teil eines *Contracts* sind Vor- und Nachbedingungen von Methoden. [Walden95]

In der von Graham entwickelten objektorientierten Modelliermethode *Soma* findet sowohl das Konzept der *Behauptung* als auch das der *Regel* Anwendung. Letzteres stammt vom Gebiet der wissensbasierten Systeme und drückt hier aus, wie ein Objekt von anderen beeinflusst wird. [Graham95]

Der Terminus *Constraint* wird im Zusammenhang mit der objektorientierten Modellierung erstmals von Rumbaugh et al. geprägt und als eine „funktionale Beziehung zwischen Entitäten eines Objektmodells“ bezeichnet. In der von ihnen entwickelten Analyse- und Designmethode OMT schränken Constraints den Wertebereich einzelner Entitäten ein. [Rumbaugh91]

Booch definiert den Begriff *Constraint* als „den Ausdruck für eine semantischen Bedingung, die eingehalten werden muss“. Nach seiner Auffassung, kann eine solche Bedingung nur dann eingehalten werden, wenn sich das System in einem stabilen Zustand befindet. Es kann folglich temporär Zustände geben, in denen die Constraints nicht vom System eingehalten werden. [Booch93]

Die seit Mitte der 90er stattfindenden Standardisierungsbemühungen im Bereich der objektorientierten Analyse und dem Design von Softwaresystemen unter dem Dach der OMG resultierten in der Verabschiedung der UML-Spezifikation. Diese beinhaltet ab der 1997 erschienen Version 1.1 das Konzept der Constraints. [OMG97]

3.3.4 Deklarative und operationale Constraints

Allgemein steht der Begriff *Constraint* im Kontext der Spezifikation von Softwaresystemen für Restriktionen, denen das System oder Modell genügen muss, damit es als konsistent zu betrachten ist bzw. sich in einem zulässigen Zustand befindet. Abweichungen in den Interpretationen von Constraints liegen vor allem darin, wie darauf reagiert werden soll, wenn ein Constraint nicht erfüllt wird.

Deklarative Constraints legen nur fest, wie ein konsistenter Zustand aussieht, nicht wie auf Inkonsistenzen reagiert werden soll. Deklarative Constraints haben keine Seiteneffekte, d.h. der Zustand des Systems wird beim Prüfen von Constraints nicht verändert. Der Modellierer muss nicht entscheiden, wie die Verletzung eines Constraints behandelt werden muss. Dieser Ansatz führt zu einer klaren Trennung von Spezifiktion und Implementation.

Ein anderer Ansatz besteht darin, dass die Verletzung eines Constraints das Werfen irgendeiner Art von Exception zur Folge haben oder das ein solches Verhalten zumindest eine Option darstellen sollte. In den Programmiersprache *Eiffel* und *Java*[1] können Assertions als Werkzeug zur Fehlerfindung dienen: wenn eine Assertion nicht erfüllt ist, wird eine entsprechende Exception geworfen.

Eine dritte Möglichkeit besteht darin, dass bei der Verletzung eines Constraints eine zu spezifizierende Methode aufgerufen wird. Man nennt diese Art von Constraints „Operationale Constraints“. In *Soma* beispielsweise können Regeln als Auslöser für vom System durchzuführende Aktionen dienen. [Graham95]

Die für eine bestimmte Domäne definierten Constraints bleiben im allgemeinen über einen längeren Zeitraum stabil. Die Aktionen mit denen das System auf die Verletzung eines Constraints reagieren sollte, ändern sich hingegen weit häufiger bzw. können von Applikation zu Applikation verschieden sein.

3.4 Zusammenfassung

Dieses Kapitel hat die Bedeutung und die historische Entwicklung des Begriffs *Constraint* in den Teilgebieten „Wissensbasierte Systeme“ und „Spezifikation von Softwaresystemen“ aufgezeigt. Dabei wurde deutlich, dass zwar Übereinstimmungen in der grundsätzlichen Interpretation des Begriffs bestehen, im Detail jedoch wesentliche Unterschiede zu finden sind. Während Constraints in wissensbasierten Systemen zum Beschreiben und Lösen von Problemen verwendet werden, haben Constraints beim Einsatz für die Spezifikation von Softwaresystemen die Aufgabe, zulässige Zustände und Zustandsübergänge zu definieren.

Das folgende Kapitel wird das Konzept zur Anwendung von Constraints in digitalen Bauwerksmodellen vorstellen. Dabei wird auch hier die grundsätzliche Bedeutung des Begriffs *Constraint* übernommen, diese wird jedoch mit einer zusätzlichen, spezifischen Semantik versehen, die sich letztlich aus Teilaspekten der beiden oben genannten Ansätze ergibt. Konkret können Constraints in Bauwerksmodellen sowohl zur Formulierung von konsistenten Zuständen als auch zur Beschreibung eines durch Randbedingungen gekennzeichneten Problems verwendet werden. Für letztere werden später die Techniken des Constraint-basierten Lösens zum Einsatz kommen können.

[1] ab Version 1.4

4 Einsatz von Constraints in digitalen Bauwerksmodellen

4.1 Überblick

Constraints im Kontext der Bauwerksmodellierung sind Zwangs- bzw. Nebenbedingungen, die für Instanzen der Klassen eines Domänenmodells gelten.

Sie können auf der einen Seite dazu eingesetzt werden, einen Teil des Domänenwissens rechnerintern abzubilden. Solche Constraints repräsentieren Regeln, die sich aus rechtlichen Normen ableiten oder den Stand der Kunst einer Fachdisziplin wiederspiegeln und in allen Projekten Gültigkeit besitzen.

Daneben können Constraints auch zur Formulierung von Entwurfsbedingungen in einem konkreten Projekt verwendet werden. Derartige Constraints formalisieren beispielsweise die speziellen Wünsche des Bauherrn oder die spezifischen fachlichen Anforderungen eines Projekts. Durch die Prüfung dieser Bedingungen können frühzeitig potentielle Entwurfskonflikte erkannt werden. So gesehen kann mit Hilfe projektspezifischer Constraints der Soll-Zustand des Planungsgegenstandes beschrieben und durch ihre Auswertung mit dem aktuellen Ist-Zustand verglichen werden.

Constraints werden mit Hilfe von Begrifflichkeiten aus der Taxonomie der Domäne formuliert, d.h. mit den im Domänenmodell verwendeten Namen von Klassen, Attributen und Rollen beschrieben. Aus der angestrebten Dynamik des Domänenmodells (vgl. Kapitel 2.3.4) folgt, dass Constraints dem Domänenmodell dynamisch, d.h. zur Laufzeit des Informationssystems, hinzugefügt werden können müssen. Dadurch kann erreicht werden, dass das im Informationssystem formalisierte Wissen sukzessive erweitert bzw. vervollständigt werden kann.

Im einfachsten Fall grenzt ein Constraint den für ein Attribut gültigen Wertebereich ein. Constraints können auch zur Abbildung von Abhängigkeiten zwischen Attributen dienen, was das Auftreten von Redundanzen im Domänenmodell erlaubt. Sie können auf diese Weise dazu beitragen, die Aussagekraft des Domänenmodells zu erhöhen und seine Verwendbarkeit für die unterschiedlichen Applikationen einer Domäne sicherzustellen. Werden Constraints in diesem Sinne verwendet, können sie als Mittel zur Formulierung und Prüfung von Konsistenzbedingungen angesehen werden.

Um die notwendige Flexibilität bei der Planung zu gewährleisten, müssen temporäre Unstimmigkeiten zwischen den Planungsdaten und den formulierten Bedingungen vom Informationssystem toleriert werden. Wenn diese jedoch nicht erkannt, überwacht und zu gegebenen Zeitpunkt beseitigt werden, kann das zu schweren Planungsfehlern mit entsprechendem Aufwand zu deren Beseitigung führen. Das kann durch Prüfung der Instanzdaten auf Einhaltung aller für sie

gültigen Constraints vermieden werden. Dabei sollte sowohl der Zeitpunkt als auch der Umfang der Prüfungen vom Planer gewählt werden können.

Constraints bereichern das Digitale Bauwerksmodell um zusätzliche Semantik und können auf diese Weise dazu beitragen, dass das Informationssystem den hohen Anforderungen der kooperativen Bauplanung gerecht wird.

4.2 Abbildung von Domänenwissen

Der Einsatz von Constraints kann dazu dienen, das rechnerintern abgebildete Fachwissen einer Domäne zu erweitern bzw. zu vervollständigen. Constraints zur Abbildung von Domänenwissen werden immer mit einer Klasse des Domänenmodells assoziiert. Die darin formulierten Bedingungen gelten dann für alle Instanzen dieser Klasse und ihrer Kindklassen.

Constraints, die Teile des Domänenwissens repräsentieren, können sich u.a. aus gesetzlichen Normen ableiten, wie beispielsweise zur Mindestneigung von Dachflächen oder der Mindestdicke von Außenwänden. Sie können neben dem im jeweiligen Fachgebiet allgemein anerkannten Wissen (Stand der Technik) aber auch das individuelle Fachwissen eines Planers ausdrücken.

Ebenso wenig wie davon ausgegangen werden kann, dass das Domänenmodell für die gesamte Laufzeit eines Projektes konstant bleibt, kann man von einer Menge statischer Zwangsbedingungen ausgehen, die das Domänenwissen repräsentieren. Auch hinsichtlich der Formulierung von Constraints gilt, dass die Menge des im Informationssystem abgebildeten Wissens mit den Anforderungen des Nutzers an das System wachsen sollte.

Deshalb muss äquivalent zur dynamischen Modifizierbarkeit des Domänenmodells dem Fachplaner die Möglichkeit eingeräumt werden, die für ein Domänenmodell festgelegten Bedingungen sukzessive erweitern bzw. an die konkreten Bedingungen eines Projekts anpassen zu können.

Bei der Entwicklung von herkömmlichen Expertensystemen wird das Fachwissen durch Wissensingenieure erfasst und in die Wissensbasis eingespeist (vgl. Kapitel 3.2). Ein Aspekt des hier propagierten Informationssystems mit der Möglichkeit zur laufzeitdynamischen Formulierung von Constraints ist es, dass der Anwender seine Wissensbasis ohne das Mitwirken von Wissensingenieuren selbst erweitern kann.

4.3 Abbildung von Projektwissen

Projektspezifische Constraints haben die Aufgabe, Entwurfspielräume innerhalb eines Projekts zu beschreiben. Die Daten eines spezifischen Planungsprojekts werden durch eine Population von Instanzen der Klassen im Domänenmodell repräsentiert. Projektspezifische Constraints beschreiben Bedingungen für eine oder mehrer Instanzen aus dieser Instanzenpopulation. Sie werden nicht mit einer Klasse des Domänenmodells, sondern mit den konkreten Instanzen assoziiert, für die diese Bedingungen gelten soll.

Beispiele für projektspezifische Constraints sind die für ein Baufeld festgelegte maximale Giebelhöhe des zu errichtenden Gebäudes und die vom Bauherrn vorgegebene notwendige Anzahl an Sitzplätzen in einem Raum.

Mit projektspezifischen Constraints kann der Soll-Zustand des zu planenden bzw. zu revitalisierenden Gebäudes beschrieben werden. Die Prüfung der Instanzpopulation auf Einhaltung derartiger Constraints kann dann zum Vergleich zwischen dem aktuellen Zustand der Planung und ihrem Soll-Zustand eingesetzt werden. Die verletzten Constraints können dem Planer als eine Art Agenda von noch zu erfüllende Aufgaben dienen. Durch die sukzessive Annäherung an den Soll-Zustand verschwinden nach und nach die verletzten Constraints und damit die noch zu erfüllenden Entwurfsaufgaben von seiner Agenda.

Projektspezifische Constraints sind einer bedeuten höheren Dynamik unterworfen als die Constraints, die dem Domänenwissen zuzuordnen sind. Dies resultiert aus der ständigen Suche nach Kompromissen während der Bauplanung und den daraus folgenden Änderungen an den Anforderungen. Weiterhin können Constraints in den frühen Phasen der Bauplanung dazu verwendet werden, vages Wissen rechnerintern abzubilden, indem sie beispielsweise den Bereich angeben, in dem sich ein Wert potentiell bewegt. Sie können auf diese Weise als Grundlage für Entscheidungsfindungen dienen und werden im Laufe der fortschreitenden Planung durch konkrete Werte ersetzt.

Der Einsatz von *Entwurfsbedingungen* im Bauplanungsprozess wurde bereits ausführlich in [Sturm97] untersucht. Sturm stellt die Entwurfsbedingungen und die sogenannten *Bauteilentscheidungen* (Einfügen und Modifizieren eines physikalischen Objektes) auf eine gemeinsame konzeptionelle Ebene und nennt beide zusammen *Entwurfsentscheidungen*. Entsprechend haben auch Entwurfsbedingungen eine Position und eine Ausdehnung. Hinzu kommen weitere sogenannte *Anwendungsdimensionen* wie Zeitraum und Auflösungsgrad, die aus dem 4D-Modell entlehnt sind. [Hovestadt94] Durch das zentrale Konzept der *Bereichsorientiertheit* werden Konflikte zwischen den Bauteilen und den Bedingungen eines Bereichs erkannt. Zusätzlich ist das Festlegen einer Reaktion auf die Verletzung einer Bedingung möglich.

Zur integrativen Verwaltung der Entwurfsentscheidungen nutzt Sturm ein DBMS[1] mit einem Datenmodell, dessen Grundstruktur eine Zweiteilung in Integrationsdatenteil und spezifischen Datenteil vorsieht. Während der spezifische Datenteil durch das Schema der Domäne geprägt ist, wird der Integrationsdatenteil durch den Datentyp *Bereich* repräsentiert, dessen Attribute die Anwendungsdimensionen enthalten. Jedes Objekt der Datenbasis besitzt beide Datenteile.

4.4 Modellierungsmöglichkeiten bei Verwendung von Constraints

4.4.1 Festlegen des gültigen Wertebereichs eines Attributes

An häufigsten werden Constraints dazu verwendet, den Wertebereiche eines Attributs einzugrenzen. Dieser Typ von Constraint kann sowohl bei der Formulierung von Bedingungen Anwendung finden, die dem Domänenwissen zuzurechnen sind, als auch beim Festlegen von projektspezifischen Bedingungen.

[1] Database Management System

Meist schränkt der für ein Attribut verwendete Datentyp den Wertebereich nicht ausreichend, d.h. auf die in der Realität zulässigen Werte ein. Die Dimensionen eines räumlichen Objektes werden beispielsweise in der Regel durch Attribute repräsentiert, die sowohl mit positiven als auch negativen Werten belegt werden können. Die Formulierung eines entsprechenden Constraints kann dafür sorgen, dass die Belegung mit einem negativen Wert als Fehler erkannt wird. (Abbildung 4.1)

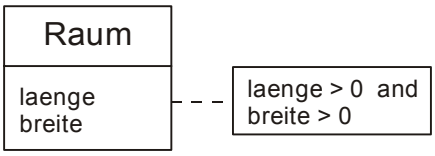

Abbildung 4.1: Beispiel für die Einschränkung der Wertebereiche von Attributen

Ebenso sind auf diese Weise Einschränkungen formulierbar, die sich aus fachspezifischen Bestimmungen ableiten, darunter die Zuweisung von maximalen und minimalen zulässigen Werten. Mit dem Einsatz eines projektgebundenen Constraints kann darüber hinaus die Belegung eines Attributs einer bestimmten Instanz auf einen spezifischen Wert festgeschrieben werden. Ein solches Constraint ist dann eher im Sinne eines Planungsziels als als strenge Konsistenzbedingung aufzufassen.

4.4.2 Modellieren von Abhängigkeiten zwischen Attributen

Damit eine verlustfreie Kommunikation zwischen den Beteiligten eines Fachplanerbereiches möglich ist, muss das Domänenmodell von allen Anwendungen verwendet werden können, die innerhalb des jeweiligen Fachbereiches eingesetzt werden. Durch das Einführen redundanter Attribute[1] kann das Domänenmodell für eine Reihe verschiedener Fachapplikationen mit voneinander differierenden internen Datenmodellen nutzbar gemacht werden, ohne das Auftreten unkontrollierbarer Inkonsistenzen in Kauf nehmen zu müssen.

Beim objektorientierten Design von Softwaresystemen werden *Methoden* u.a. dazu eingesetzt, um lesenden Zugriff auf die Eigenschaften eines Objektes zu erlangen, deren Wert sich aus den Attributen des selben oder anderer Objekte ergibt. In rein deskriptiven Bauwerksmodellen können Methoden als Modellierungsmittel nicht verwendet werden. Das führt dazu, dass deskriptive Modelle mitunter nicht über genügend Ausdruckkraft verfügen, um alle Eigenschaften eines Objektes abzubilden.

Abhilfe hierfür kann durch die Einführung von *abgeleiteten Attributen* geschaffen werden. Die Berechnung des Wertes eines abgeleiteten Attributes wird von einer Fachapplikation übernommen, die über die hierzu notwendige Logik verfügt. Alle anderen Fachapplikationen der Domäne können auf diesen Wert direkt zugreifen,

[1] Das AKO-Metamodell definiert Slots als Modellierkonstrukt für Eigenschaften. Die Semantik ist zu der des Attributs weitestgehend äquivalent, erlaubt jedoch zusätzlich die Modellierung von Facetten. In diesem Kapitel soll zur Vereinfachung der Begriff *Attribut* verwendet werden, gleichwohl sind die beschriebenen Möglichkeiten ohne Beschränkung auf die einzelnen Facetten eines Slots übertragbar.

ohne die notwendigen Berechnungen selbst durchführen zu müssen. Mit Hilfe eines Constraints kann die bestehende Abhängigkeit zwischen einem abgeleiteten Attribut und seinen Quellen explizit modelliert werden. Damit kann einerseits die Konsistenz der betreffenden (im Prinzip redundant vorliegenden) Daten geprüft werden. Andererseits kann eine derartige Abbildung von indirekten Abhängigkeiten die Grundlage für die Implementierung von Funktionalitäten zur Versionierung von Planungsständen schaffen. [Firmenich02] beschreibt die mathematischen Grundlagen zur Versionierung von Objektmengen und nennt als eine wesentliche Voraussetzung die Kenntnis des Systems über die Abhängigkeiten, die zwischen den einzelnen Objekten bestehen. Er verwendet den Begriff der *Bindung* für die Beschreibung von Abhängigkeiten zwischen Objekten.

Abbildung 4.2 zeigt als Beispiel für ein abgeleitetes Attribut innerhalb einer Klasse die Klasse `Raum` mit den Attributen `laenge`, `breite` und `flaeche`. Möchte man redundant vorliegende Daten vollkommen vermeiden, darf das Attribut `flaeche` nicht modelliert werden, da sich dessen Wert aus den anderen beiden Attributen ergibt.

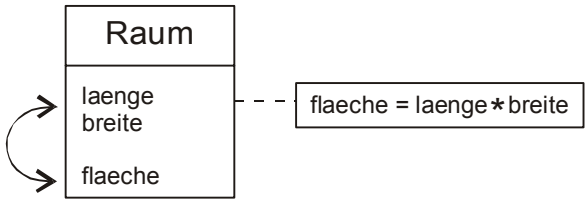

Abbildung 4.2: Beispiel für die Abhängigkeit zwischen Attributen einer Klasse

Die Notwendigkeit zur expliziten Modellierung von Abhängigkeiten zwischen Attributen im Domänenmodell leitet sich auch aus den unzureichenden Möglichkeiten derzeit existierender Fachapplikation zum Festlegen von Abhängigkeiten ab. Häufig werden entsprechende Attribute eines Objekts aus Gründen der Anwendungslogik gesondert vorgehalten und eine unabhängige Manipulation durch den Fachplaner ermöglicht. Um die Kompatibilität mit der Applikation zu gewährleisten, muss das Domänenmodell alle anwendungsintern verwalteten Attribute aufweisen.

Als Beispiel seien hier solche Anwendungen genannt, die für die dreidimensionale Modellierung von Bauwerken zum Einsatz kommen. In der Regel wird anwendungsintern jedes Bauteil als unabhängiger Körper modelliert. Alle Wände eines Geschosses besitzen also eine eigene Höhe, Breite und Länge. Diese Daten werden für jede Wand gesondert vorgehalten und können einzeln und unabhängig voneinander manipuliert werden. Damit die Anwendung mit dem Modellverwaltungssystem kommunizieren kann, muss das dort vorliegende Domänenmodell die Klasse `Wand` mit den Attributen `hoehe`, `breite` und `laenge` beinhalten, obwohl sich in der Sichtweise des Planers die Höhe einer Wand immer aus der Höhe des jeweiligen Geschosses ergibt. (Abbildung 4.3)

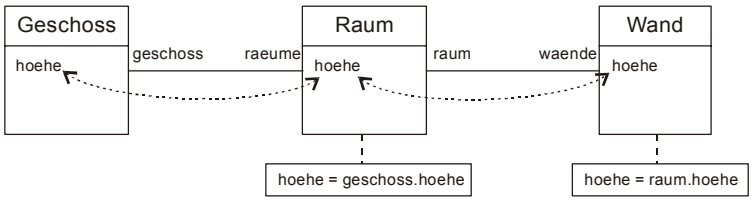

Abbildung 4.3: Beispiel für Abhängigkeiten zwischen Attributen verschiedener Klassen

4.4.3 Einschränkung assoziierbarer Instanzen

Constraints können dazu eingesetzt werden, um in einer Subklasse die Zielklasse einer geerbten Assoziation genauer festzulegen. Dadurch können aufwendige Modellierungen, die ein Domänenmodell unnötig aufblähen, vermieden werden.

Als Beispiel soll hier die Assoziation zwischen den Klassen `Bauteil` und `Aussparung` dienen. Um die Wirklichkeit möglichst präzise abzubilden, muss bei der Modellierung ohne Constraints für die Assoziationen zwischen den Klassen `Außenwand` und `Haustür` respektive `Innenwand` und `Zimmertür` jeweils eine gesonderte Subklasse von `Aussparung` gebildet werden (Abbildung 4.4).

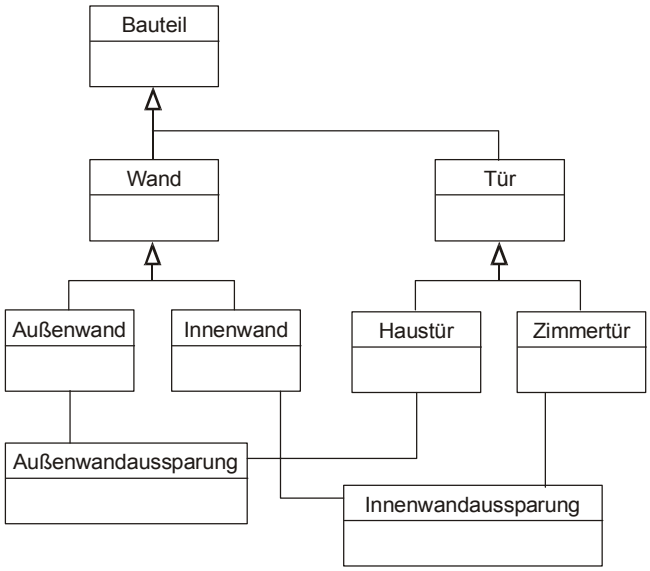

Abbildung 4.4: Präzise Modellierung von Assoziationen ohne Constraints

Dies kann vermieden werden, indem für die Klassen `Außenwand` und `Haustür` ein Constraint auf die geerbte Assoziation mit dem Rollennamen `aussparung` definiert wird, dass auf der anderen Seite nur ein Objekt der jeweils gültigen Subklasse von `Aussparung` von erlaubt. Geht man analog für die Klassen Innenwand und Haustür vor, ist das Resultat ein deutlich leichter zu handhabendes Domänenmodell. (Abbildung 4.5)

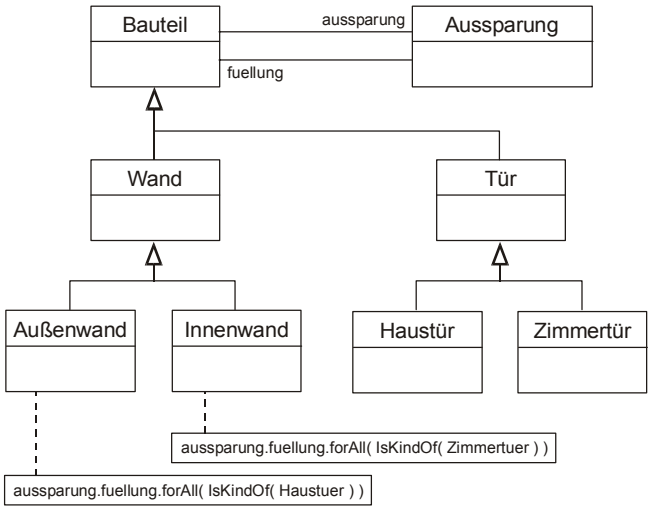

Abbildung 4.5: Vereinfachte Modellierung durch Einsatz von Constraints

4.4.4 Beschreibung der Semantik von Assoziationen

Assoziationen repräsentieren Beziehungen zwischen Objekten. Diese Beziehungen sind häufig an Nebenbedingungen für die beteiligten Objekte geknüpft. Werden Beziehungen durch Assoziationsklassen[1] modelliert, können die Nebenbedingungen mit Hilfe eines für die Assoziationsklasse gültigen Constraints formalisiert werden.

Als Beispiel soll hier das durch IFC-Produktmodell dienen, das eine ganze Reihe von sogenannten Relationsklassen[2] beinhaltet. Ganz besonders interessant im Zusammenhang mit der Möglichkeit der Beschreibungen von Nebenbedingungen durch Constraints sind die Assoziationsklassen, die eine topologische Beziehung abbilden. Hier sollen beispielhaft die Klassen `RelFillsElement` und `RelVoidsElement` betrachtet werden, die die Beziehung zwischen einem Bauteil (z.B. einer Wand), einer darin befindliche Öffnung und dem in dieser Öffnung befindlichen Bauteil (z.B. ein Fenster) wiedergeben (Abbildung 4.6).

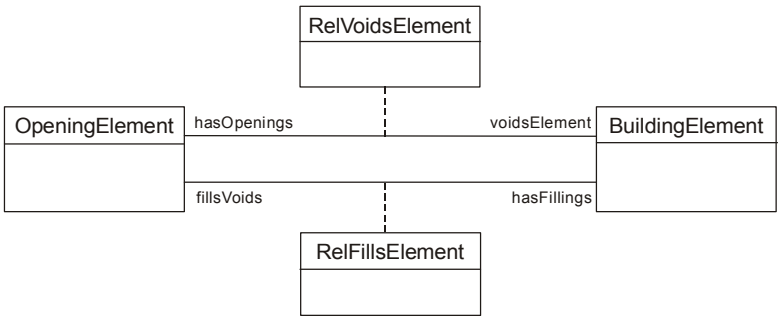

Abbildung 4.6: Assoziationsklassen im IFC-Produktmodell

[1] im AKO-Metamodell existieren das Konstrukt der Assoziationsklasse nicht, sie werden in AKO-Modellen daher wie reguläre Klassen modelliert

[2] Relationsklassen werden im IFC-Produktmodell wie reguläre Klassen modelliert, besitzen aber den zusätzlichen Präfix „Rel“ im Namen.

Für Assoziationsklassen mit einer topologischen Semantik ist die Formulierung von Konsistenzbedingungen besonders signifikant. Beispielsweise sollte für die Klasse `RelFillsElement` ein Constraint gelten, das anhand der Koordinaten und der Ausdehnung sicherstellt, dass das referenzierte Bauteil tatsächlich räumlich in der Öffnung enthalten ist.

4.4.5 Topologische Zwangsbedingungen

Da der Planungsgegenstand Bauwerk letztlich ein physikalisches Objekt mit geometrischer Ausformung ist, besitzen topologische Zwangsbedingungen eine besondere Signifikanz für die Konsistenz der Planungsdaten.

Als Beispiel sei hier die Klasse `Bauteil` genannt, die mit Bedingungen verknüpft werden muss, die die topologische Verträglichkeit der einzelnen `Bauteil`-Instanzen untereinander beschreiben. Bei auftretender Kollision von Bauteilen handelt es sich um eine schwerwiegende Inkonsistenz in den Planungsdaten, die auf diese Weise festgestellt werden kann.

Topologische Zwangsbedingungen gelten für die meisten der im Bauplanungsprozess involvierten Domänen. Sie sind in der Regel mit Klassen assoziiert, die im Wurzelbereich der Domänenmodelle zu finden sind. Topologische Zwangsbedingungen sollten zum Standardumfang der initialen Konfiguration von Modellen der Domänen gehören, in denen physikalische Objekte geplant werden, weil ihre Formulierung einerseits sehr komplex ist und sie andererseits eine gute Basis für darauf aufbauende nutzerdefinierte Constraints bilden.

4.5 Constraint-Prüfungen

Mit der Prüfungen von Instanzen auf die Einhaltung der für sie gültigen Constraints lässt sich feststellen, wo sich Fehler oder Unlogik in den Planungsdaten eingeschlichen haben bzw. welche Planungsaufgaben noch zu erfüllen sind. Die Constraint-Prüfung findet einheitlich für domänen- und projektbezogene Constraints statt. Hinsichtlich des Zeitpunkts der Durchführung und des Umfangs der zu prüfenden Instanzen gibt es unterschiedliche Möglichkeiten.

4.5.1 Explizite oder implizite Constraint-Prüfungen

Bezüglich des Auslösens von Constraint-Prüfungen sind zwei Ansätze zu unterscheiden. Entweder wird die Constraint-Prüfung durch das System initiiert (implizite Prüfung) oder durch den Nutzer (explizite Prüfung). Während implizites Prüfen bedeutet, dass das Prüfen der Instanzen auf Einhaltung der für sie gültigen Constraints vom Informationssystem initiiert wird, kann beim expliziten Prüfen der Nutzer bzw. seine Anwendung den Zeitpunkt der Prüfung bestimmen.

Eine Variante der impliziten Constraint-Prüfung besteht darin, dass nach jedem modifizierenden Zugriff auf die Daten einer Instanz geprüft wird, ob die für sie gültigen Konsistenzbedingungen eingehalten sind. Eine andere Möglichkeit ist, dass die Prüfung aller Instanzdaten in bestimmten Zeitintervallen automatisch vom System vorgenommen wird. Das Prüfen einer sehr großen Menge von Instanzdaten, wie sie in realen Projekten auftreten, ist jedoch als äußerst ressourcenintensiv anzusehen und kann zu Performance-Einbrüchen des Systems führen.

Um einerseits eine unnötige Belastung des Systems zu vermeiden und andererseits dem Anwender die nötige Flexibilität hinsichtlich der Toleranz von Unstim-

migkeiten einzuräumen, bieten sich explizit auszulösende Constraint-Prüfungen an. Dabei besitzt der Nutzer die Kontrolle darüber, zu welchem Zeitpunkt und in welchem Umfang Constraint-Prüfungen durch das System vorgenommen werden. Er kann damit steuern, wann und in welchem Maß das System durch Constraint-Prüfungen belastet wird. Arbeitet eine Anwendung im Online-Modus, kann sie ggf. dafür Sorge tragen, dass in bestimmten Intervallen oder nach größeren Modifikationen eine automatische Konsistenzprüfung initiiert wird.

4.5.2 Zeitpunkt der Constraint-Prüfung

Man kann hinsichtlich des Zeitpunkts der Durchführung von Constraint-Prüfungen verschiedene Klassen mit unterschiedliche Signifikanz für den Bauplanungsprozess unterscheiden.

Zur ersten Klasse gehören Konsistenzprüfungen, die zu einem, vom Fachplaner frei wählbaren Zeitpunkt während der Planungstätigkeit an einem Teil des Datenbestandes stattfinden. Derartige Konsistenzprüfungen werden vergleichsweise häufig durchgeführt. Sie dienen dem Fachplaner als Grundlage dafür, Entscheidungen über als nächstes durchzuführende Planungsaktivitäten zu fällen. Dieser Vorgang kann als äquivalent zum Kompiliervorgang bei der Softwareentwicklung betrachtet werden, bei dem der Quellcode syntaktisch und semantisch geprüft wird. Im Bauplanungsprozess kann auf diese Art und Weise festgestellt werden, ob der aktuell betrachtete Teil des Datenbestands in sich konsistent ist, d.h. ob alle Constraints eingehalten werden.

Die zweite Klasse von Constraint-Prüfungen wird beim Zusammenführen von Teildatenbeständen nach deren getrennter Offline-Bearbeitung durchgeführt. Dabei wird festgestellt, ob sich während der getrennten Bearbeitung unkontrollierte Entwurfskonflikte ergeben haben. In diesem Fall dient die Constraint-Prüfung der Konsistenzsicherung im Rahmen kooperativer Arbeit.

Eine dritte Art von Constraint-Prüfungen findet beim Erzeugen eines Freigabestands statt. Die Anforderungen an die Konsistenz der Planungsdaten von Freigabebeständen sind sehr hoch, da sie rechtlich verbindlich sind und als Grundlage der Bauausführung dienen.

4.5.3 Umfang der zu prüfenden Instanzdaten

Zur Gewährleistung einer größtmöglichen Flexibilität sollte dem Nutzer die Möglichkeit eingeräumt werden

- einzelne Instanzen auf bestimmte Constraints
- einzelne Instanzen auf Einhaltung aller für sie gültigen Constraints
- eine durch Aggregationsbeziehungen gekennzeichnete Instanzenmenge
- die gesamte Instanzpopulation eines Projekts

zu prüfen.

4.5.4 Umgang mit Constraint-Verletzungen

Zwar ist grundsätzlich eine automatisierte Reaktion auf die Verletzung eines Constraints in Form einer Reparatur der betroffenen Daten denkbar. Bei dem hier verfolgten Ansatz zur Verwendung von Constraints wird jedoch vor allem auf eine Toleranz des Systems gegenüber temporären Unstimmigkeiten Wert gelegt.

Die Reaktion auf eine festgestellte Verletzung von Zwangs- bzw. Nebenbedingungen obliegt daher dem Fachplaner. Die verletzten Constraints und die betroffenen Instanzen sollten ihm nach erfolgter Prüfung in der Entwurfsumgebung auf geeignete Weise angezeigt werden. Je nach Grad bzw. Signifikanz der Verletzung kann die Reaktion des Planers im sofortigen Beheben der Ursachen oder auch im Vertagen auf einen späteren Zeitpunkt bestehen. Im letzten Fall kann eine Liste von noch nicht eingehaltenen Bedingungen dem Planer als Agenda dienen.

4.6 Vorteile beim Einsatz von Constraints

Der Einsatz von Constraints in digitalen Bauwerksmodellen bringt eine Reihe von Vorteilen mit sich, die hier zusammenfassend aufgeführt werden sollen.

Constraints vervollständigen die rechnerinterne Abbildung des Domänen- bzw. Projektwissens. Sie sind in der Lage, den für ein Attribut gültigen Wertebereich zu beschreiben. Auf der einen Seite können so klare Konsistenzbedingungen für die Wertbelegungen dieses Attributs definiert werden, auf der anderen Seite ist damit vage bzw. unscharfe Information abbildbar.

Constraints erlauben die Modellierung von Abhängigkeiten zwischen den Wertbelegungen von Attributen. Damit kann die Integrität redundant vorliegender Daten gesichert werden. Die Modellierung von Abhängigkeiten zwischen Attributen ist eine wesentliche Grundlage für die Umsetzung von Funktionalitäten zur Versionierung von Planungsdaten. Darüber hinaus können Constraints zum Festlegen des Typs der über eine Relation assoziierbaren Instanzen oder zur näheren Beschreibung der Semantik von Assoziationsklassen dienen.

Der Vorteil beim Einsatz von Constraints ist, dass Unstimmigkeiten zwischen den Nebenbedingungen und dem Datenbestandes zeitweise toleriert werden können. Nur zu vom Fachplaner festzulegenden Zeitpunkten wird die Konsistenz des Datenbestandes geprüft und ggf. von ihm wiederhergestellt. Bei der manuellen Wiederherstellung werden je nach Typ des Constraints Inkonsistenzen behoben oder Entwurfskonflikte gelöst.

Der Einsatz von Constraints im digitalen Bauwerksmodell kann den Fachplaner bei der Planungsarbeit dahingehend unterstützen, dass Widersprüche in der eigenen Planungsleistung oder im Zusammenhang mit den Planungsleistungen anderer an der Planung Beteiligter vom Informationssystem erkannt werden. Daraus können noch zu erledigende Aufgaben abgeleitet werden, die dem Planenden in Form einer Agenda zur Verfügung gestellt werden können.

Dank der formalen Definition der Konsistenzbedingungen kann das Informationssystem beim Lösen von Entwurfskonflikten assistieren. Ausgehend von der Existenz eines Netzes verknüpfter Constraints, ist es denkbar, dass mit Hilfe von Methoden des Constraint-basierten Lösens (vgl. Kapitel 3.2.3) eine oder mehrere Möglichkeiten zur Beseitigung einer Konsistenzverletzung durch das Informationssystem berechnet werden können.

4.7 Anforderungen an die Modellverwaltung

Aus den in diesem Kapitel aufgezeigten Möglichkeiten zum Einsatz von Constraints in Digitalen Bauwerksmodellen lassen sich folgende Anforderungen an die Funktionalität eines Modellverwaltungssystems ableiten:

1. Die Bedingungen müssen mit Hilfe der im Domänenmodell verwendeten Bezeichner beschrieben werden können, also den Namen von Relationen, Slots und Facetten.
2. Zur Abbildung von Domänenwissen müssen Constraints mit Klassen assoziiert werden können.
3. Zur Abbildung von Projektwissen müssen Constraints mit Instanzen assoziiert werden können.
4. Die Bedingungen müssen zu einem vom Nutzer festzulegenden Zeitpunkt geprüft werden können.

4.8 Zusammenfassung

Dieses Kapitel hat die verschiedenen Möglichkeiten des Einsatzes von Constraints in Digitalen Bauwerksmodellen und die daraus erwachsenden Vorteile detailliert dargelegt. Im besonderen konnte aus den verschiedenen Arten der Verwendung von Constraints Anforderungen an das Modellverwaltungssystem abgeleitet werden.

Die Details der Definition einer Bedingung wurden bislang außen vorgelassen. Im nächsten Kapitel soll daher die *Object Constraint Language* (OCL) vorgestellt werden, eine standardisierte formale Sprache zur Formulierung von Constraints. Im Kapitel 7 wird dann die Umsetzung eines Constraint-Moduls für das Modellverwaltungssystem des SFB 524 besprochen, das den hier erarbeiteten Anforderungen genügt.

5 Die Object Constraint Language (OCL)

5.1 Einführung

5.1.1 Überblick

Die *Object Constraint Language* (OCL) ist seit der Version 1.1 Bestandteil der *Unified Modeling Language* (UML), einer Spezifikation des Industrieverbands *Object Management Group* (OMG). Hauptbestandteil der Spezifikation ist eine graphischen Notation für die objektorientierten Analyse und dem objektorientierten Design von Softwaresystemen. Die OCL ist eine formale Sprache in textueller Notation, mit deren Hilfe Constraints für ein in UML modelliertes System formuliert werden können. Die Constraints spezifizieren dabei Bedingungen, denen das modellierte System genügen muss.

Daneben wurde OCL in der UML-Spezifikation zur formalen Spezifikation des Metamodells eingesetzt: Da das UML-Metamodell zu seiner eigenen Beschreibung verwendet wird, können OCL-Constraints zum Festlegen von Zwangsbedingungen zwischen Entitäten der Metaebene dienen.

Wesentlicher Vorteil von OCL gegenüber älteren Constraint-Sprachen ist die Einfachheit der Sprache, die dazu führt, dass sie intuitiv erlernbar und der Einarbeitungsaufwand entsprechend gering ist.

5.1.2 Historische Entwicklung

Die OCL hat ihre geschichtlichen Wurzeln in den theoretischen Grundlagen der formalen Spezifikation von Softwaresystemen und denen daraus entwickelten Sprachen. Die Notationssprache *Z*, die als Vorfahre von OCL angesehen werden kann, wurde Ende der siebziger Jahre an der Oxford University entwickelt und ist mittlerweile von der ISO[1] als Standard[2] verabschiedet worden. Sie ist stark auf die theoretischen Grundlagen der formalen Spezifikation, die Mengenalgebra und die first-order-Prädikatenlogik, orientiert. *Z* gilt daher als nur von Experten mit vertieftem Wissen auf diesen Gebieten beherrschbar. [Spivey88]

Die Anfang der neunziger Jahre von John Daniels und Steve Cook entwickelte objektorientierte Modelliermethode *Syntropy* führte die graphischen Modellier-sprache *Object Modeling Technique* (OMT) mit einer Untermenge von *Z* zusammen,

[1] International Organization for Standardization

[2] Z Formal Specification Notation - Syntax, Type System and Semantics, ISO/IEC 13568:2002

um erstmals die Vorteile einer graphischen Notation mit denen der formalen Spezifikation zu vereinen. [Cook94]

Die OCL wurde 1995 von IBM-Mitarbeitern für ein Projekt entwickelt, in dem Geschäftsprozesse modelliert werden sollten. Die Entwicklung von OCL wurde zwar stark von *Syntropy* beeinflusst, im Unterschied zu *Syntropy* werden in OCL jedoch keine Symbole der Mengenalgebra ($\forall$, $\exists$, ...) verwendet, sondern deren Semantik durch entsprechende Operationen ersetzt (forAll, exists, ...). Dadurch sind OCL-Ausdrücke zwar zum einen formal, können aber zum anderen auch von Nutzern ohne mathematisches Hintergrundwissen verwendet werden.

Bei der Entwicklung der UML-Spezifikation unter dem Dach der OMG fokussierte der Beitrag der Rational Software Corporation vor allem auf der Beschreibung der Semantik und der graphischen Notation von UML-Konstrukten. OCL als nicht-graphische Erweiterung war der Beitrag von IBM, der der UML-Spezifikation erstmals in der Version 1.1 hinzugefügt wurde.

Die 1992 von D'Souza und Wills entwickelte, auf UML beruhende Modelliermethode *Catalysis* macht ausführlichen Gebrauch von der OCL-Syntax zur präzisen Spezifikation von Softwarekomponenten. Einige der OCL-Konstrukte werden mit einer leicht veränderten Syntax und Semantik verwendet und eine ganze Reihe zusätzlicher Sprachkonstrukte eingeführt. [D'Souza99]

Für die Version 2.0 der OCL-Spezifikation ist die Einführung einer abstrakten Syntax und deren Trennung von der konkreten (textuellen) Syntax geplant. Damit wird es möglich sein, eine alternative konkrete Syntax, wie beispielsweise eine graphische Notation, zu verwenden und sie auf die abstrakte Syntax abzubilden. Momentan wird daran gearbeitet, eine geeignete Form der graphischen Notation für OCL-Constraints zu entwickeln. [Kiesner02] Zudem sieht der aktuelle Vorschlag vor, die OCL als allgemeine Abfragesprache für UML-Modelle zu definieren. [OMG03b]

5.1.3 Verwendung von OCL

Ein UML-Diagramm, wie beispielsweise ein Klassendiagramm, kann in der Regel nicht alle relevanten Aspekte der Spezifikation eines Softwaresystems abbilden. Unter anderem ist es für eine genaue und vollständige Beschreibung notwendig, Nebenbedingungen festzulegen, denen die Objekte im System genügen müssen. Solche Nebenbedingungen werden in der Praxis häufig in natürlicher Sprache beschrieben.

Es hat sich jedoch gezeigt, dass die Formulierung von Zwangsbedingungen mit Hilfe der natürlichen Sprache immer zu Unklarheiten bzw. Zweideutigkeiten führt. Um eindeutige Nebenbedingungen formulieren zu können, wurden daher sogenannte formale Sprachen entwickelt. Der Nachteil traditioneller formaler Sprachen ist jedoch, dass sie zwar von Personen mit mathematischem Hintergrund gut zu verwenden sind, aber nur eingeschränkt von Modellierern von Softwaresystemen. Die OCL wurde entwickelt, um diese Lücke zu schließen. Sie ist eine formale Sprache, die leicht zu lesen und zu schreiben ist.

OCL ist eine rein deskriptive Sprache, d.h. ein OCL-Ausdruck ist garantiert ohne Seiteneffekte auf den Zustand des Systems: wenn ein OCL-Ausdruck ausgewertet wird, wird ein Wert zurückgegeben und nichts am Modell geändert. Trotzdem

können OCL-Ausdrücke in Form von Vor- und Nachbedingungen dazu benutzt werden, eine Zustandsänderung zu spezifizieren.

OCL ist keine Programmiersprache, deswegen besitzt OCL keine Elemente zur Beschreibung von Programmlogik oder Programmflusssteuerung. Außerdem können mittels OCL keine Prozesse oder Nicht-Abfrage-Operationen aufgerufen werden. Die Auswertung eines OCL-Ausdrucks geschieht unverzüglich, d.h. dass sich der Zustand der Objekte im Modell während der Auswertung nicht ändert.

In OCL ist jeder Ausdruck ist von einem bestimmten Typ. Um als „wohlgeformt“ zu gelten, muss ein OCL-Ausdruck den Typregeln der Sprache genügen. Beispielsweise ist es nicht erlaubt, einen Integer-Wert mit einem String zu vergleichen. Jeder Klassifizierer aus dem zugehörigen UML-Modell ist automatisch ein OCL-Typ. Darüber hinaus beinhaltet die OCL eine Menge von vordefinierten Typen wie *Integer*, *Boolean* usw.

Da OCL eine Spezifikationssprache ist, liegen alle Implementierungsaspekte außerhalb ihres Anwendungsbereiches, d.h. sie können nicht mit OCL ausgedrückt werden.

OCL kann für folgende Zwecke verwendet werden:

- Festlegen von invarianten Bedingungen für Klassen und Typen im Klassenmodell
- Festlegen von Typ-Invarianten bei Stereotypen
- Festlegen von Vor- und Nachbedingungen für Operationen und Methoden
- Festlegen von Einschränkungen für Operationen

5.2 Beschreibung der Sprache

Die folgende Beschreibung von OCL ist zu großen Teilen dem Kapitel 6 der UML-Spezifikation Version 1.4 vom September 2001 entnommen. [OMG01]

5.2.1 Beispiel-Klassendiagramm

Abbildung 5.1 zeigt das Klassendiagramm des UML-Modells, auf dem die in diesem Kapitel aufgeführten OCL-Constraints beruhen.

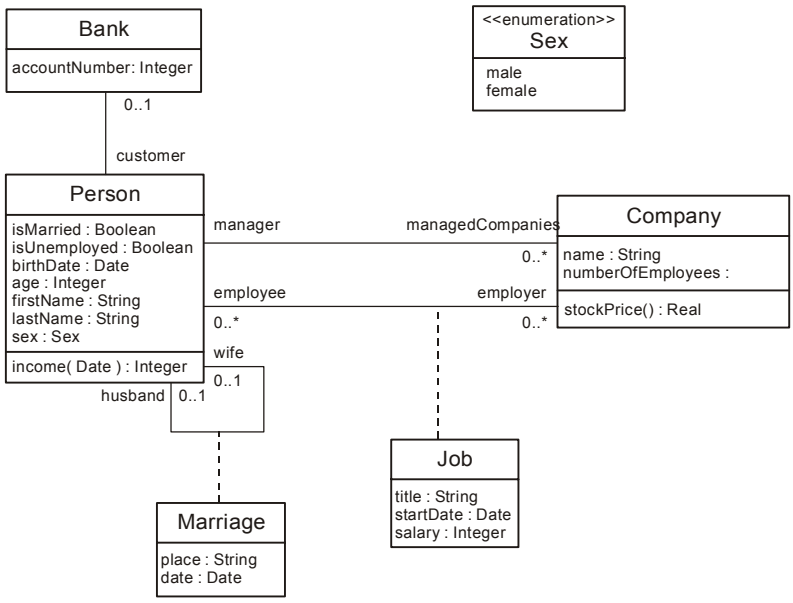

Abbildung 5.1: Klassendiagramm für die verwendeten Beispiele

5.2.2 OCL-Ausdrücke und OCL-Constraints

Die OCL-Spezifikation unterscheidet zwischen OCL-Ausdrücken und OCL-Constraints.

Definition: Ein OCL-Ausdruck ist genau dann gültig, wenn er entsprechend der Regeln von OCL formuliert wurde.

Jeder OCL-Ausdruck hat einen Typ: entweder einen der im UML-Modell definierten Typen oder einen der vordefinierten Typen von OCL. Jeder OCL-Ausdruck besitzt weiterhin ein Resultat: Das ist der Wert, der sich durch die Auswertung des OCL-Ausdrucks ergibt. Der Typ des Resultats ist gleich dem Typ des OCL-Ausdrucks. Beispielsweise ist „`3+4`“ ein gültiger OCL-Ausdruck vom Typ Integer, dessen Resultat `7` ist.

Ein Constraint ist als eine Regel für einen oder mehrere Werte definiert, die bei ihrer Auswertung *wahr* ergibt, wenn die Bedingungen eingehalten werden. Daraus folgt, dass ein Constraint ein OCL-Ausdruck vom Typ *Boolean* sein muss.

Definition: Ein OCL-Constraint ist ein gültiger OCL-Ausdruck vom Typ Boolean.

5.2.3 Verbindung zum UML-Metamodell

5.2.3.1 Self

Jeder OCL-Ausdruck wird im Kontext der Instanz eines spezifischen Typs geschrieben. Im OCL-Ausdruck kann das reservierte Wort `self` benutzt werden, um sich auf die kontextuelle Instanz zu beziehen. Wenn beispielsweise der Kontext eines Constraints die Klasse `Bauteil` ist, dann bezieht sich `self` auf eine Instanz von `Bauteil`.

5.2.3.2 UML-Kontext

Der Kontext eines OCL-Ausdrucks innerhalb eines UML-Modells kann durch eine entsprechende Deklarierung am Anfang eines OCL-Ausdrucks spezifiziert werden. Wenn das Constraint in einem Diagramm gezeigt wird und eine gestrichelte Linie zu seinem kontextuellem Element besitzt, ist keine explizite Deklaration des Kontextes notwendig. Die Kontext-Deklaration ist optional.

5.2.3.3 Invariante Bedingungen

Invariante Nebenbedingung werden durch Constraints des Stereotyps „invariant“ beschrieben. Wenn die invariante Bedingung mit einem Klassifizierer assoziiert ist, wird dieser als „Typ“ bezeichnet. Ein OCL-Ausdruck ist eine invariante Bedingung dieses Typs und muss für alle seine Instanzen zu jeder Zeit wahr sein. Alle OCL-Ausdrücke, die invariante Bedingungen beschreiben, sind vom Typ `Boolean`.

Beispielsweise würde im Kontext des `Company`-Typs der folgende Ausdruck eine invariante Bedingung spezifizieren:

```
self.numberOfEmployees > 50
```

Dabei ist `self` die Referenz auf eine Instanz vom Typ `Company`. Man kann `self` als das Objekt auffassen, von dem aus die Navigation startet.

Der Typ, für den das Constraints eine invariante Bedingung beschreibt, wird durch das Schlüsselwort `context`, gefolgt vom Namen des Typs festgelegt. Damit ist auch der Typ der kontextuellen Instanz festgelegt. Die Kennzeichnung mit `inv:` deklariert das Constraint als invariante Bedingung.

```
context Company inv:
  self.numberOfEmployees > 50
```

In den meisten Fällen kann das Schlüsselwort `self` entfallen, weil sich der Kontext wie in den vorangegangenen Beispielen implizit ergibt. Als Alternative zu `self` kann ein anderer Name definiert werden, der die Rolle von `self` übernimmt. Zum Beispiel ist das folgende Constraint äquivalent zum vorhergehenden:

```
context c : Company inv:
  c.numberOfEmployees > 50
```

Optional kann für das Constraint ein Name[1] festgelegt werden. Die Verwendung eines Namens erlaubt, dass das Constraint von anderen referenziert werden kann. Der Name des Constraints wird nach dem Schlüsselwort `inv` angegeben. Im folgenden Beispiel ist der Name des Constraints `enoughEmpoyees`.

---

[1] Im UML-Metamodell besitzt die Metaklasse *Constraint* ein Attribut *name*, das diese von ihrer Superklasse *ModelElement* erbt.

```
context c : Company inv enoughEmployees:
  c.numberOfEmployees > 50
```

5.2.3.4 Vor- und Nachbedingungen

Constraints vom Stereotyp „precondition“ bzw. „postcondition“, beschreiben Vor- oder Nachbedingung von Operationen und werden entsprechend mit einer Operation assoziiert. Das Schlüsselwort `self` im dazugehörigen OCL-Ausdruck bezieht sich auf eine Instanz von dem Typ, für den die Operation deklariert wurde. Zum Festlegen des Kontextes folgt nach dem Schlüsselwort `context` der Name des Typs und der Operation. Der Stereotyp des Constraints wird durch die Kennzeichnung mit `pre:` oder `post:` vor der eigentlich Vor- bzw. Nachbedingung angezeigt.

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

Das Schlüsselwort `self` kann im Ausdruck benutzt werden, um sich auf das Objekt zu beziehen, von dem die Operation gerufen worden ist. Das reservierte Wort `result` steht für den Rückgabewert einer Operation, sofern ein solcher existiert. Die Namen der Parameter der Operation (`param1`, `param2`...) können auch im OCL-Ausdruck benutzt werden.

```
context Person::income(d : Date) : Integer
  post: result = 5000
```

Optional kann ein Name für die Vor- bzw. Nachbedingung nach dem Schlüsselwort `pre` oder `post` angegeben werden. Dadurch kann das Constraint über diesen Namens referenziert werden. Im folgenden Beispiel ist der Name der Vorbedingung `parameterOk` und der Name der Nachbedingung `resultOk`.

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

5.2.3.5 Package-Kontext

Die obige Kontextdeklaration ist hinreichend genau, wenn das Package, zu dem der Klassifizierer gehört, eindeutig ist. Um explizit zu spezifizieren, zu welchem Package ein Constraint gehört, muss es zwischen `package` und `endpackage` Statements eingeschlossen werden. Die `package` Statements haben folgenden Syntax:

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

Ein OCL-File kann eine Vielzahl von `package`-Statements beinhalten. Das erlaubt, dass alle zu einem UML-Modell gehörigen Constraints in einem einzigen File geschrieben und gespeichert werden können.

5.2.4 Typen

In OCL hat jedes Objekt einen Typ. Er legt die Operationen fest, die auf dem Objekt ausgeführt werden können. Typen in OCL werden in zwei Gruppen unterschieden:

- Vordefinierte Typen
  - Basistypen
  - Kollektionstypen
- Nutzerdefinierte Typen des UML-Modells

Jede Klasse, jedes Interface und jeder Typ im zugehörigen UML-Modell ist automatisch ein Typ in OCL.

5.2.4.1 Vordefinierte Basistypen

Basistypen sind unabhängig vom UML-Modell und können in jedem OCL-Ausdruck verwendet werden. Tabelle 5.1 zeigt die verfügbaren Basisdatentypen.

Typ	**Wertebereich**
Boolean	true, false
Integer	ganze Zahlen
Real	reelle Zahlen
String	Zeichenketten

Tabelle 5.1: Die Basisdatentypen von OCL

Boolean-Typ

Ein Boolean-Typ kann einen von zwei Werten annehmen: *wahr* oder *falsch*. Die für Boolean definierten Operationen sind in Tabelle 5.2 aufgeführt:

Operation	**Notation**
ODER-Verknüpfung	a **or** b
UND-Verknüpfung	a **and** b
Exklusives ODER	a **xor** b
Negation	**not** a
Gleichheit	a = b
Ungleichheit	a <> b
Implikation	a **implies** b
Bedingung	**if** a **then** b **else** B' **endif**

Tabelle 5.2: Operationen des Boolean-Typs

Die Semantik der meisten Operationen ist identisch mit der in verbreiteten Programmiersprachen, daher wird hier nicht näher darauf eingegangen. Die Implikation wird hingegen vergleichsweise selten in Programmiersprachen und häufiger in Spezifikationssprachen verwendet. Die Operation gibt genau dann *wahr* zurück, wenn der erste Operand *wahr* ist und dann auch der zweite Operand *wahr* ist. Ist der Wert des ersten Operanden *falsch*, gibt der Implikationsausdruck *wahr* zurück.

Integer- und Real- Typen

Der Integer-Typ repräsentiert die natürlichen Zahlen. Da OCL eine Modelliersprache ist, gibt es keine Restriktionen bezüglich des Wertebereichs von Integer. Der Real-Typ repräsentiert reelle Zahlen. In der Mathematik sind die natürlichen Zahlen eine Untermenge der reellen, daher ist Real ein Subtyp von Integer.

Die für Integer und Real-Typen definierten Operationen zeigt Tabelle 5.3.

Operation	**Notation**
Gleichheit	a = b
Ungleichheit	a <> b
„kleiner als“	a < b
„größer als“	a > b
„kleiner-gleich“	a <= b
„größer-gleich“	a >= b
Addition	a + b
Substraktion	a – b
Multiplikation	a * b
Division	a / b
Modulus	a.mod(b)
Integer-Division	a.div(b)
Absolutwert	a.abs
Maximum	a.max(b)
Minimum	a.min(b)
Runden	a.round
Abrunden	a.floor

Tabelle 5.3: Operationen für Integer- und Real-Typen

String-Typ

Strings sind Sequenzen von Zeichen, also Zeichenketten. String-Literale werden innerhalb von einfachen Anführungszeichen geschrieben, wie beispielsweise `'rot'`. Die für Strings definierten Operationen zeigt Tabelle 5.4.

Operation	**Notation**
Verkettung	string1.concat(string2)
Länge	string.size
Kleinbuchstaben	sring.toLower
Großbuchstaben	string.toUpper
Teilstring	string.substring(int ,int)
Gleichheit	string1 = string2
Ungleichheit	string1 <> string2

Tabelle 5.4: Operationen für den String-Typ

5.2.4.2 Typen vom UML-Modell

Jeder OCL-Ausdruck ist einem UML-Modell zugeordnet, und befindet sich im Kontext der dort definierten Klassifizierer (Typen, Klassen, Interfaces), ihren Eigenschaften, Assoziationen und ihren Generalisierungsbeziehungen. Alle Klassifizierer im UML-Modell sind Typen in den OCL-Ausdrücken, die mit diesem Modell verbunden sind.

5.2.4.3 Aufzählungstypen

Aufzählungstypen sind Datentypen in UML und besitzen wie jeder andere Klassifizierer einen Namen. Eine Aufzählung (engl. enumeration) definiert eine Anzahl von Literalen, die eine mögliche Wertebelegung darstellen. Innerhalb von OCL kann man auf den Wert einer Aufzählung bezug nehmen. Wenn es beispielsweise im UML-Modell den Datentyp `Geschlecht` mit den möglichen Werten `männlich` und `weiblich` gibt, kann er wie folgt verwendet werden:

```
context Person inv:
  sex = Sex::male
```

5.2.4.4 Typverträglichkeit

Basistypen von OCL sind in einer Typhierarchie organisiert. Diese Hierarchie determiniert die Verträglichkeit der verschiedenen Typen zueinander. Beispielsweise ist es nicht möglich, einen *Integer*- mit einem *Boolean*-Wert zu vergleichen.

Ein OCL-Ausdruck, in dem die Typverträglichkeit nicht gegeben ist, ist ungültig. Ein *Typ1* ist verträglich mit einem *Typ2*, wenn eine Instanz von *Typ1* an jeder Stelle ersetzt werden kann, wo eine Instanz von *Typ2* erwartet wird

Regeln für die Typverträglichkeit:

- Jeder Typ ist mit allen seinen Supertypen verträglich.
- Die Typverträglichkeit ist transitiv: wenn *typ1* mit *typ2* verträglich ist und *typ2* mit *typ3*, dann ist auch *typ1* mit *typ3* verträglich.

Typ	**Ist verträglich mit / Ist Subtyp von**
Set(T)	Collection(T)
Sequence(T)	Collection(T)
Bag(T)	Collection(T)
Integer	Real

Tabelle 5.5: Verträglichkeit der OCL-Typen

Die Verträglichkeit zwischen Kollektionen ist nur gegeben, wenn es sich um Kollektionen von Elementen handelt, deren Typen miteinander verträglich sind.

5.2.4.5 Retyping oder Casting

Unter bestimmten Umständen ist es wünschenswert, Zugriff auf die Eigenschaft eines Objektes zu erhalten, die für einen Subtypen des aktuell von diesem Objekt bekannten Typs des definiert ist. Wenn sicher ist, dass der eigentliche Typ des Objekts der Subtyp ist, kann das Objekt mit Hilfe der Operation `oclAsType()` umgetypt werden. Der Rückgabewert dieser Operation ist das selbe Objekt, aber der

nunmehr vom Objekt bekannte Typ ist gleich dem als Argument `aType` übergebenen.

```
object.oclAsType(OclType aType)
```

5.2.5 Let-Ausdrücke und Definition-Constraints

Mitunter soll ein Teilausdruck mehrmals innerhalb eines Constraints benutzt werden. Der `let`-Ausdruck erlaubt es, einen Stellvertreter für diesen Teilausdruck als ein Attribut oder eine Operation zu definieren, die im Constraint an mehreren Stellen verwendet werden kann.

Zum Beispiel:

```
context Person inv:
  let income : Integer = self.job.salary->sum()
  let hasTitle(t : String) :
    Boolean = self.job->exists(title = t) in
  if isUnemployed then self.income < 100
  else self.income >= 100 and self.hasTitle('manager')
  endif
```

Ein `let`-Ausdruck kann in einer invarianten Bedingung, in einer Vor- oder einer Nachbedingung enthalten sein. Er ist dann nur in diesem spezifischen Constraint bekannt. Um die Wiederverwendbarkeit der Stellvertreter zu gewährleisten, kann man ein Constraint vom Stereotyp „definition“ verwenden, in dem ausschließlich `let`-Ausdrücke definiert sind.

Dieses Definition-Constraint muss mit einem Klassifizierer verbunden sein und darf ausschließlich `let`-Definitionen beinhalten. Alle so definierten Variablen und Operationen sind dann im selben Kontext bekannt, in dem der Zugriff auf reguläre Attribute und Operationen gestattet ist. D.h. solche Variablen und Operationen sind pseudo-Attribute und pseudo-Operationen des Klassifizierers. Sie werden in OCL-Ausdrücken in der gleichen Weise benutzt wie reguläre Attribute und Operationen. Die textuelle Notation eines Definition-Constraints benutzt das Schlüsselwort `def`:

```
context Person def:
  let income : Integer = self.job.salary->sum()
  let hasTitle(t : String) : Boolean =
  self.job->exists(title = t)
```

Die Namen von Attributen bzw. Operationen in einem `let`-Ausdruck dürfen nicht mit den Namen der regulären Attribute, Assoziationsenden und Operationen des Klassifizierers in Konflikt kommen. Auch müssen die Namen aller `let`- Variablen und -Operationen, die mit einem Klassifizierer verbunden sind, eindeutig sein.

5.2.6 Vorrangregeln

Operationen werden in folgender Reihenfolge ausgeführt (höchste Priorität zuerst):

- @pre
- Punkt- und Pfeiloperationen: '.' und '->'
- unäres 'not' and unäres '-'
- '*' und '/'
- '+' und binäres '-'
- 'if-then-else-endif'

- ‘<’, ‘>’, ‘<=’, ‘>=’
- ‘=’, ‘<>’
- ‘and’, ‘or’ und ‘xor’
- ‘implies’

Klammern können genutzt werden, um den Vorrang zu ändern. Dabei wird der am weitesten innen liegende Ausdruck zuerst ausgeführt.

5.2.7 Nutzung der Infix-Notation

Die Operatoren `+`, `-`, `*`, `/`, `<`, `>`, `<>`, `<=`, `>=` können sowohl in Infix- als auch in Postfix-Notation verwendet werden. Wenn ein Typ einen dieser Operatoren mit der korrekten Signatur definiert, kann er als Infix-Operator benutzt werden.

Der Ausdruck

```
a + b
```

ist identisch mit dem Ausdruck

```
a.+(b)
```

und bedeutet, dass die ‚+’-Operation an `a` mit `b` als Parameter ausgeführt wird.

5.2.8 Schlüsselwörter

Schlüsselwörter in OCL sind reservierte Wörter. Das bedeutet, dass diese Schlüsselwörter nicht an beliebiger Stelle im OCL-Ausdruck auftreten dürfen, wie als Name eines Packages, eines Typs, oder eines Attributs.

Die Liste von Schlüsselwörtern:

```
context  def  inv  pre  post  in
package  endpackage
if  then  else  endif
let
not  or  and  xor  implies
```

5.2.9 Kommentare

Kommentare werden in OCL nach zwei aufeinanderfolgenden Bindestrichen geschrieben. Alles was direkt diesen zwei Strichen folgt bis zum Ende der Zeile ist Teil des Kommentars:

```
-- this is a comment
```

5.2.10 Undefinierte Werte

Wann immer ein OCL-Ausdruck ausgewertet wird, gibt es die Möglichkeit das eine der Abfragen im Ausdruck nicht definiert ist. Wenn das der Fall ist, dann ist der gesamte Ausdruck undefiniert.

Es gibt zwei Ausnahmen von dieser Regel für Boolean-Operatoren:

- *true* **`or`** *irgendetwas* ergibt *true*
- *false* **`and`** *irgendetwas* ergibt *false*

5.2.11 Objekte und deren Eigenschaften

OCL-Ausdrucke können sich auf Klassifizierer aus dem UML-Modell beziehen, also auf Typen, Klassen, Interfaces, Assoziationen und Datentypen. Auch alle Attribute,

Assoziationsenden, Methoden und Operationen ohne Seiteneffekte, die für diese Typen definiert sind, können verwendet werden. In einem Klassenmodell ist eine Operation oder Methode als frei von Seiteneffekten definiert, wenn das `isQuery`-Attribut dieser Operation auf `true` gesetzt ist. In der OCL-Spezifikation werden Attribute, Assoziationsenden, Methoden und Operationen als *Eigenschaften* (engl. properties) bezeichnet.

5.2.11.1 Eigenschaften

Auf die Wertbelegung der Eigenschaft eines Objektes wird mit Hilfe des Punktoperators, gefolgt vom Namen der Eigenschaft zugegriffen.

```
context AType inv:
  self.property
```

Wenn `self` die Referenz auf ein Objekt ist, dann ist `self.property` der Wert dieser Eigenschaft von `self`.

5.2.11.2 Zugriff auf Attribute

Beispielsweise wird auf das Alter einer Person mittels `self.age` zugegriffen:

```
context Person inv:
  self.age > 0
```

Der Wert des Ausdrucks `self.age` ist gleich der Wertbelegung des Attributs `age` einer bestimmten Instanz von `Person`. Der Typ dieses Unterausdrucks ist gleich dem Typ des Attributs `age`, also `Integer`.

5.2.11.3 Zugriff auf Operationen

Operationen können Parameter besitzen. Zum Beispiel definiert die Klasse `Person` die Operation `income()`, die das Einkommen als Funktion des Datums berechnet. Der Zugriff auf die Operation geschieht wie folgt:

```
aPerson.income(aDate)
```

Die Semantik der Operation kann durch eine Nachbedingung spezifiziert werden. Auf den Rückgabewert kann dabei über das reservierte Wort `result` zugegriffen werden:

```
context Person::income (d: Date) : Integer
  post: result = age * 1000
```

Die rechte Seite der Nachbedingung kann sich auf die Operation beziehen, für die sie definiert wird. Das heißt, die Definition darf rekursiv sein, solange die Rekursion endlich ist. Der Typ von `result` ist gleich dem Rückgabetyp der Operation, im Beispiel `Integer`.

Um sich auf eine Operation zu beziehen, die keine Übergabeparameter besitzt, müssen Klammern mit einer leeren Argumentliste angegeben werden:

```
context Company inv:
  self.stockPrice() > 0
```

5.2.11.4 Assoziationsenden

Von einem spezifischen Objekt ausgehend, kann man eine Assoziation im Klassendiagramm entlang navigieren, um sich auf andere Objekte und ihre Eigenschaften zu beziehen. Um entlang einer Assoziation zu navigieren, wird der

Punktoperator gefolgt vom Rollenbezeichner des gegenüberliegenden Assoziationsendes verwendet:

```
object.rolename
```

Der Wert dieses Ausdrucks ist die Menge der Objekte auf der anderen Seite der Assoziation. Wenn die Kardinalität des Assoziationsendes maximal 1 ist („0..1“ oder „genau 1“), dann ist der Wert dieses Ausdrucks ein Objekt.

Wenn beispielsweise `manager` und `employee` die Rollen an gegenüberliegenden Assoziationsenden von `Company` sind:

```
context Company
  inv: self.manager.isUnemployed = false
  inv: self.employee->notEmpty()
```

In der ersten Bedingung ist `self.manager` eine Person, weil die Kardinalität der Assoziation 1 ist. In der zweiten Bedingung wird `self.employee` als eine ungeordnete Menge (`Set`) von Personen ausgewertet. Wenn die Assoziation in einem Klassendiagramm mit `{ordered}` versehen ist, wird eine Sequenz (`Sequence`) zurückgegeben.

Kollektionen wie `Set`, `Bag` und `Sequence` sind vordefinierte Typen in OCL. Es gibt eine große Zahl von vordefinierten Operationen für sie. Auf die Eigenschaft einer Kollektion selbst wird über das Pfeilsymbol „`->`“ gefolgt vom Bezeichner der Eigenschaft zugegriffen. Im folgenden Beispiel wird die Größe des Sets, also die Menge der über `employer` assoziierten `Objekte` mit Hilfe der Operation `size()` abgefragt.

```
context Person inv:
  self.employer->size() < 3
```

Navigation über Assoziationen mit Kardinalität „0 oder 1“

Weil im Beispielmodell die Kardinalität der Assoziation mit der Rolle `manager` genau 1 ist, ergibt der Ausdruck `self.manager` ein einzelnes Objekt vom Typ `Person`. Ein solches einzelnes Objekt kann jedoch ebenso als Set verwendet werden, das genau ein Objekt beinhaltet. Auch hierbei erfolgt der Zugriff auf Eigenschaften des Sets mit Hilfe des Pfeilsymbols gefolgt vom Bezeichner der Operation. Zum Beispiel:

```
context Company inv:
self.manager->size() = 1
```

Wird beim Zugriff auf eine Eigenschaft hingegen das Punktsymbol verwendet, so wird der vorangegangene Ausdruck nicht als Kollektion, sondern als Objekt behandelt. In folgendem Beispiel wird auf die Eigenschaft `age` des über `manager` assoziierten Objekts zugegriffen:

```
context Company inv:
self.manager.age > 40
```

Verketten von Eigenschaftszugriffen

Der Zugriff auf eine Eigenschaft resultiert immer in einem Objekt von einem bestimmten Typ. Auf dessen Eigenschaften kann über eine einfache Verkettung von Zugriffsoperationen erneut zugegriffen werden. Zum Beispiel:

```
context Person inv:
self.wife->notEmpty() implies self.wife.age >= 18
```

Navigation zu Assoziationsklassen

Im Fall der Verwendung einer Assoziationsklasse gibt es keinen expliziten Rollennamen im UML-Klassendiagramm. Um zu Navigationsklassen zu navigieren, wird in OCL daher der Punktoperator gefolgt vom Namen der Assoziationsklasse verwendet. Der Name der Assoziationsklasse muss mit kleinem Anfangsbuchstaben beginnen.

Die Auswertung des folgenden Ausdrucks resultiert beispielsweise in einem Set von Job-Objekten:

```
context Person inv:
self.job
```

Im Fall einer rekursiven Assoziation reicht die Angabe der Assoziationsklasse allein nicht aus, da auf diese Weise nicht unterschieden werden kann, in welche Richtung die Assoziation navigiert werden soll. Deswegen wird der betreffende Rollenname, der die Richtung der Navigation festlegt, in Klammern angegeben.

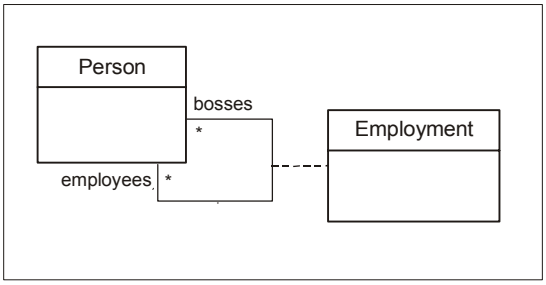

Abbildung 5.2: Beispieldiagramm zur Verwendung einer Assoziationsklasse

Der folgende Ausdruck gehört zum in Abbildung 5.2 dargestellten Modell. Seine Auswertung resultiert in einem Set von `Person`-Objekten, die Beschäftige der Person sind, die den Kontext bildet.

```
context Person inv:
self.employement [employees]
```

Navigation entlang von qualifizierten Assoziationen

Bei der Verwendung von qualifizierten Assoziationen, werden ein oder mehr kennzeichnende Attribute genutzt, um das Objekt am anderen Ende der Assoziation auszuwählen. Um entlang einer qualifizierten Assoziation zu navigieren, kann ein Wert für das kennzeichnende Attribut in eckigen Klammern angegeben werden. Das Resultat sind die Objekte, bei denen die Wertbelegung des Kennzeichnungsattributs mit dem angegebenen Wert übereinstimmt.

Beispielsweise kann die Kontonummer eines Bankkunden ein kennzeichnendes Attribut sein. Dann ist das Resultat des folgenden Ausdrucks der Bankkunde mit der Kontonummer 8764423:

```
context Bank inv:
self.customer[8764423]
```

5.2.11.5 Zugriff auf überschriebene Eigenschaften von Supertypen

Um auf die überschriebenen Eigenschaften von Supertypen zuzugreifen, muss das betreffende Objekt mit Hilfe der für `OclAny` definierten Operation `oclAsType()` gecastet werden.

5.2.11.6 Zugriff auf die vorhergehende Wertbelegung in Nachbedingungen

Der OCL-Ausdruck in einer Nachbedingung kann sich auf die Wertbelegungen der Eigenschaften des Objekts vor Ausführung der Methode beziehen. Dazu muss die Nachsilbe `@pre` an den Namen der Eigenschaft angehängt werden.

Zum Beispiel:

```
context Person::birthdayHappens()
post: age = age@pre + 1
```

Die Nachsilbe `@pre` darf nur in OCL-Ausdrücken verwendet werden, die Teil einer Nachbedingung sind.

5.2.12 Pfadnamen für Packages

Da die Elemente eines UML-Modells in Packages organisiert werden können, bietet OCL einen Weg, um sich explizit auf einen Typ in einem anderen Package zu beziehen. Die Syntax dafür ist eine Folge von Package-Namen, separiert durch jeweils zwei Doppelpunkte und gefolgt vom Typnamen:

```
Packagename1::Packagename2::Typename
```

5.2.13 Kollektionen

Kollektionen treten immer dann auf, wenn mit einem Objekt mehr als ein Objekt über den gleichen Rollennamen assoziiert ist, also genau dann, wenn eine Klasse im UML-Modell eine Assoziation besitzt, deren anderem Ende eine Kardinalität größer als 1 zugewiesen ist.

Die OCL definiert drei verschieden Typen von Kollektionen: `Set`, `Bag` und `Sequence`. Dabei kann in einem `Set` jedes Element nur genau einmal auftreten, in einem `Bag` hingegen mehrmals. Während die Elemente von `Bag` und `Set` keiner Ordnung unterliegen, sind Elemente einer `Sequence` geordnet.

Set ist die Standard-Kollektion für den Zugriff auf Objekte einer Assoziation, deren Kardinalität größer als 1 ist. Ist die Assoziation im UML-Modell als `{ordered}` gekennzeichnet, resultiert die Traversierung in einer `Sequence`-Kollektion. Eine `Bag`-Kollektion tritt in der Regel nur dann auf, wenn zwei Kollektion kombiniert werden.

Für Kollektionen definiert die OCL eine große Zahl von Operationen. Auf diese kann mit Hilfe des Pfeilsymbols (`->`) zugegriffen werden. Zum Beispiel:

```
context Company inv:
  self.manager->size() = 1
```

5.2.13.1 Operationen für alle Kollektionstypen

Die Tabelle 5.6 zeigt die für alle Kollektionstypen definierten Operationen. Die konkreten Operationen für die einzelnen Kollektionstypen unterscheiden sich lediglich im Rückgabetyp und im Typ des übergebenen Parameters.

Operation	**Semantik**
size()	gibt die Zahl der Elemente zurück
count(object)	Häufigkeit des Auftretens eines Objekts in der Kollektion
includes(object)	wahr, wenn das Objekt ein Element der Kollektion ist
includesAll(collection)	wahr, wenn alle Elemente der übergebenen Kollektion in der aktuellen Kollektion vorhanden sind
excludes(object)	wahr, wenn das Objekt kein Element der Kollektion ist
excludesAll(collection)	wahr, wenn keins der Objekte in der übergebenen Kollektion Element der aktuellen Kollektion ist
isEmpty	wahr, wenn die Kollektion kein Element beinhaltet.
notEmpty	wahr, wenn die Kollektion mindestens ein Element beinhaltet
iterate(expression)	der übergebene Ausdruck wird für jedes Element der Kollektion ausgewertet. Der Rückgabetyp ist vom Ausdruck abhängig.
sum()	die Addition aller Elemente in der Kollektion
exists(expression)	wahr, wenn der Ausdruck für mindestens ein Element der Kollektion wahr ist.
forAll(expression)	wahr, wenn der Ausdruck für alle Elemente wahr ist.
union(collection)	vereinigt zwei Kollektionen.
intersection(collection)	bildet die Schnittmenge zweier Kollektionen.
including(object)	fügt der Kollektion ein einzelnes Element hinzu
excluding(object)	entfernt ein einzelnes Element aus der Kollektion
collect(expression)	berechnet einen Wert für jedes Element der Kollektion und sammelt diese Werte in einer neuen Kollektion.
select(expression)	bildet die Teilmenge der Kollektion, deren Elemente die übergebene Bedingung erfüllen.
reject(expression)	bildet die Teilmenge der Kollektion, deren Elemente die übergebene Bedingung nicht erfüllen

Tabelle 5.6: Operationen für alle Typen von Kollektionen

5.2.13.2 Zusätzliche Operationen für Sequenzen

Da Sequenzen geordnete Kollektionen sind, sind für sie zusätzlich Operationen für den expliziten Zugriff auf eine bestimmte Position definiert. (Tabelle 5.7)

Operation	**Semantik**
first	Zugriff auf das erste Element der Sequenz
last	Zugriff auf das letzte Element der Sequenz
at (pos)	Zugriff auf das Element an der Position *pos*
append(object)	Fügt ein Element an erster Stelle ein.
prepend(object)	Fügt ein Element an letzter Stelle an.

Tabelle 5.7: Operationen für Sequenzen

5.2.13.3 Iterierende Operationen

Die Operationen `select`, `reject`, `collect`, `forAll`, `exists` und `iterate` iterieren über die einzelnen Element der Kollektion.

Es gibt drei syntaktische Varianten für iterierende Operationen:

```
collection->operation( <expression> )
collection->operation( element | <expression> )
collection->operation( element : Type | <expression> )
```

Die erste Variante ist die einfachste, da hier die Iterator-Variable implizit deklariert ist. Der Ausdruck in Klammern befindet sich im Scope der Klasse der Elemente. Das heißt der Zugriff auf deren Eigenschaften kann direkt erfolgen, z.B.

```
context Geschoss inv:
  raeume -> forAll( hoehe = self.hoehe )
```

Die zweite Variante findet in solchen Ausdrücken Verwendung, bei denen eine explizite Referenz auf die aktuelle Instanz benötigt wird, z.B.

```
context Raum inv:
  self.geschoss.raeume -> select( w:Wand | w<>self )
```

Dieser Ausdruck gibt alle übrigen Räume im selben Geschoss zurück, indem von den mit dem zugehörigen Geschoss assoziierten Räumen diejenigen ausgewählt werden, die nicht gleich der Kontextinstanz sind.

Bei der dritten Variante wird der Typ des Iterators explizit angegeben. Dadurch soll sichergestellt werden, dass die Element der Kollektion vom erwarteten Typ sind.

5.2.13.4 Kollektionen von Kollektionen

Innerhalb von OCL werden Kollektionen von Kollektionen automatisch verflacht, das heißt, es existieren keine verschachtelten Kollektionen. Die beiden folgenden Ausdrücke sind identisch:

```
Set { Set(1, 2), Set{3, 4}, Set{5, 6}}
Set { 1, 2, 3, 4, 5, 6 }
```

5.2.14 OclAny

Im Kontext von OCL ist `OclAny` der Supertyp von allen im zugehörigen Modell definierten Klassifizierern und den vordefinierten Basistypen von OCL. Die für `OclAny` definierten Eigenschaften zeigt Tabelle 5.8.

Operation	Semantik
=	wahr, wenn zwei Objekte identisch sind
<>	wahr, wenn zwei Objekte nicht identisch sind
oclIsKindOf()	wahr, wenn das Objekt vom angegebenen Typ oder einem seiner Subtypen ist
oclIsTypeOf()	wahr, wenn das Objekt vom angegebenen Typ ist.
oclAsType()	castet das Objekt zum angegebenen Typ
oclInState()	wahr, wenn das Objekt im angegebenen Zustand ist
oclIsNew()	für Nachbedingungen: wahr, wenn das Objekt während des Ausführens erzeugt wurde

Tabelle 5.8: Eigenschaften von OclAny

Die OCL-Kollektionen sind keine Subtypen von `OclAny`.

5.2.15 OclType

Alle im zugehörigen UML-Modell definierten Klassifizierer und die in OCL definierten Typen sind Instanzen von `OclType`. Mit Hilfe der für `OclType` definierten Operationen besitzt der Nutzer begrenzten Zugriff auf die Metaebene des Modells. (Tabelle 5.9)

Operation	Semantik
name()	gibt den Namen des Typs zurück
attribute()	gibt die Namen der Attribute zurück
associationEnds()	gibt die Namen der Assoziationsenden zurück
operations()	gibt die Namen der Operationen zurück
supertypes()	gibt alle direkten Supertypen zurück
allSuperTypes()	gibt alle Supertypen zurück
allInstances()	gibt alle Instanzen dieses Typen zurück

Tabelle 5.9: Operationen von OclType

5.3 Die OCL-Grammatik

Die OCL-Spezifikation schlägt eine Grammatik in EBNF[1]-Notation zum Parsen von OCL-Constraints vor. Sie ist im Anhang B1 vollständig aufgeführt und gehört zum Chomsky-Typ 2, also zur Klasse der Kontextfreien Grammatiken. Sie kann mit einem linksrekursiven Parser (LL-Parser) abgearbeitet werden. [Aho88]

Die Grammatik ist sehr unpräzise, d.h. sie beschreibt einen deutlich größere Menge von Sätzen, als in der Sprache OCL gültig sind. Beispielsweise werden die Schlüsselwort `self` und `result` als solche nicht explizit in der Grammatik aufgeführt. Die Art der Formulierung der Grammatik führt dazu, dass `self` als *Property* (also als Bezeichner eines Slots bzw. einer Relation) aufgefasst wird. Ebenso wird die in OCL-Ausdrücken notwendige Typverträglichkeit nicht von der Grammatik erfasst. Das führt dazu, dass ein großer Teil der OCL-Syntax nicht beim Parsen geprüft werden kann.

In der angegebenen OCL-Grammatik existiert ein potentieller Entscheidungskonflikt in der Produktion `propertyCallParameters` zwischen den beiden Produktionen `declarator` und `actualParameterList`, weil beide Produktionen mit einem `<name>`-Terminal beginnen können. Dadurch ist der Parser gezwungen, nicht nur den aktuellen, sondern noch einen weiteren Token im Voraus zu betrachten, um eine Entscheidung fällen zu können. Entsprechend handelt es sich bei der OCL-Grammatik um eine LL(2)-Grammatik.

[1] Erweiterte Bauckus-Naur-Form

6 Untersuchung der Eignung von OCL für den Einsatz in AKO-basierten Modellverwaltungssystemen

6.1 Überblick

Diese Kapitel wird untersuchen, inwieweit sich OCL für die Definition von Constraints in Domänenmodellen eignet, die von AKO-basierten Modellverwaltungssystemen verwaltet werden. Da die OCL für die Erweiterung von UML-Modellen konzipiert wurde, wird ein Vergleich zwischen den Metamodellen von AKO und UML als Grundlage für die Bewertung dienen.

Ein Teil der AKO-API und die Klassendiagramme der UML dienen auf den ersten Blick dem selben Zweck: der Beschreibung eines Daten- bzw. Domänenmodells. Bei näherer Betrachtung wird jedoch augenscheinlich, dass die beiden Metamodelle mit unterschiedlichen Intentionen entworfen wurden.

Die UML ist eine deklarative Sprache, bei der eine graphische Notation zur Definition von Datenmodellen genutzt wird. UML dient als Mittel, die Prozesse der Objektorientierten Analyse und des Objektorientierten Designs zu formalisieren. Das in der Analyse-Phase mit Hilfe eines UML-Klassendiagramms modellierte Datenmodell ist weitestgehend abstrakt. Es wird in der Design-Phase mit Hilfe sogenannter *Profiles* auf die Möglichkeiten der im Projekt eingesetzten Programmier- bzw. Definitionssprache abgebildet. [OMG02] Um möglichst viele Aspekte der verwendeten Programmiersprache modellieren zu können, besitzt die UML eine Fülle von Konstrukten, die Implementierungsdetails abbilden können. Instanzen von Klassen aus einem UML-Klassendiagramm sind immer Objekte der jeweiligen Programmiersprache. Nur das laufenden Programm hat Kontrolle über den Zustand eines Objektes.

Die AKO-API besteht aus einer Reihe von Interfaces, die Operationen zum sukzessiven Modellieren eines Datenmodells anbieten. Eine Unterscheidung zwischen Analyse- und Designphase wird nicht getroffen: Die Klassen des Domänenmodells können zu jeder Zeit instanziiert werden. Das AKO-Metamodell erlaubt zudem die Kontrolle über den Zustand und den Lebenszyklus der Instanzen.

Aus der Perspektive der dynamischen Modifizierbarkeit des Datenmodells und der Nutzbarkeit für die Manipulation von Instanzen sind die Metamodelle zweier mit UML verwandter Standards eher mit dem von AKO vergleichbar: Das *Common*

Warehouse Metamodell (CWM) und das in den *Meta Object Facilities* (MOF) definierte Metamodell. [OMG03c] [OMG02b] Da OCL jedoch Teil der UML-Spezifikation ist und auf dem UML-Metamodell basiert, ist letzteres Gegenstand des Vergleichs mit dem AKO-Metamodell.

Die UML definiert neben den Klassendiagrammen eine Zahl weiterer Strukturdiagramme, darunter Objektdiagramme, Komponentendiagramme und Deployment-Diagramme. Zudem existiert eine Reihe von Diagramme zum Modellieren dynamischer Aspekte eines Softwaresystems, darunter Use-Case-Diagramme, Zustandsübergangsgraphen und Kollaborationsdiagramme.

Die OCL bezieht sich jedoch nur auf Klassendiagramme und in einem Randaspekt auf Zustandsübergangsdiagramme. Daher beschränkt sich der hier vorgenommene Vergleich zwischen den Metamodellen auf Seiten des UML-Metamodells auf den Teil, der für die Klassendiagramme definiert wurde. Auf der Seite des AKO-Metamodells wird nur der „linke Teil“ untersucht werden, als die Metaklassen, die zum Modellieren eines Datenmodells dienen. (vgl. Abbildung 2.2)

6.2 Das UML-Metamodell

Die UML ist eine sehr mächtige Modelliersprache und das UML-Metamodell ist entsprechend komplex. Hier sollen nur die Teile des Metamodells näher betrachtet werden, die signifikant für OCL sind. Die folgenden Klassendiagramme sind stark vereinfacht, um die für den Vergleich mit dem AKO-Metamodell relevanten Aspekte des UML-Metamodells zu betonen.

In UML werden drei Typen von Klassifizierern unterschieden: Klassen, Datentypen und Interfaces. Ein Klassifizierer besitzt Eigenschaften (engl. features), die sich in strukturelle Eigenschaften und Verhaltenseigenschaften aufteilen. (Abbildung 6.1)

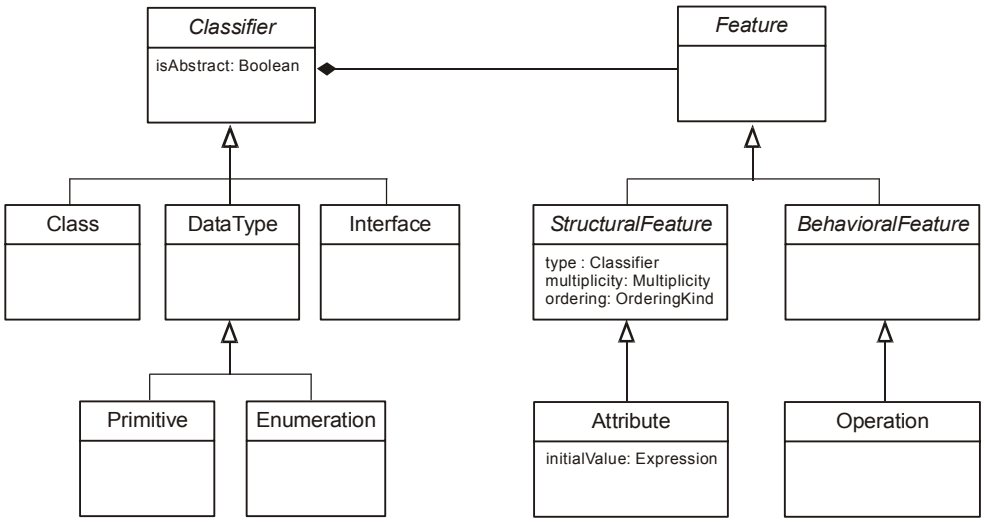

Abbildung 6.1: Die Metaklassen Classifier und Feature im UML-Metamodell

Die Klasse ist die wichtigste Art von Klassifizierern. Eine Klasse beschreibt eine Menge von Objekten, die die selben Attribute, Operationen, Relationen und die gleiche Semantik besitzen. Interfaces sind eine Sammlung von Operationen und

besitzen keine Attribute. Sie erlauben eine abstrakte Modellierung von Verhalten bzw. die Spezifikation eines Dienstes, den eine Klasse zur Verfügung stellt.

Datentypen sind Klassifizierer, deren Ausprägungen keine Identität besitzen. Dazu gehören sowohl die primitiven Datentypen wie Strings und Zahlen, als auch Aufzählungstypen wie beispielsweise *Boolean*.

Zum Modellieren von Beziehungen zwischen Klassifizierern besitzt das UML-Metamodell die Klasse `Relationship` mit den beiden Kindklassen `Association` und `Generalization` (Abbildung 6.2). Während eine Assoziation eine Beziehung zwischen zwei gleichwertigen Klassifizierern beschreibt, dient die Generalisierung zum Modellieren von Generalisierungshierarchien zwischen Klassifizierern. Zu beachten ist, dass ein Klassifizierer, wie beispielsweise eine Klasse, niemals eine direkte Verbindung mit der Assoziation besitzt, sondern nur mit einem der beiden Assoziationsenden.

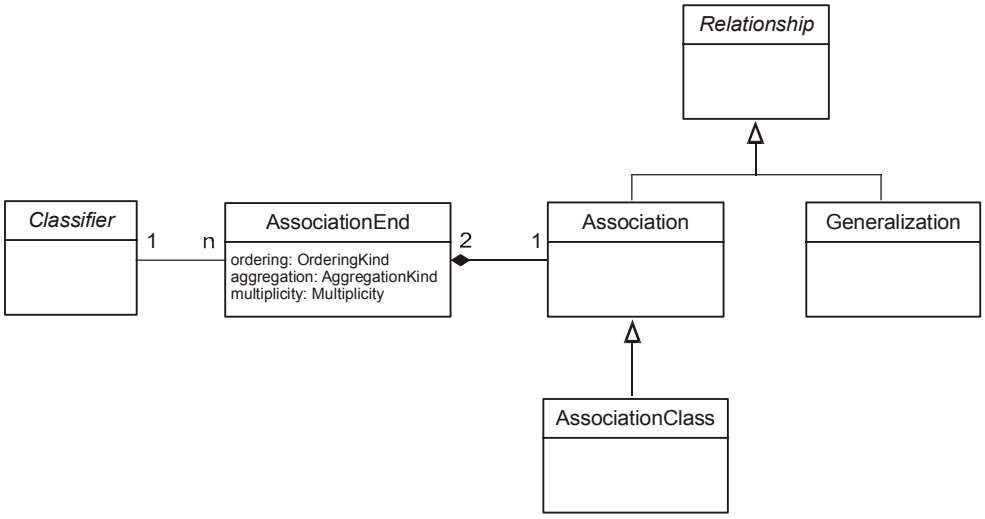

Abbildung 6.2: Die Klasse Relationship im UML-Metamodell

Die Metaklasse `Package` dient dazu, Modellelemente in Gruppen zusammenzufassen. Zu den Modellelementen gehören unter anderem Klassifizierer, Assoziationen und auch die Packages selbst. (Abbildung 6.3)

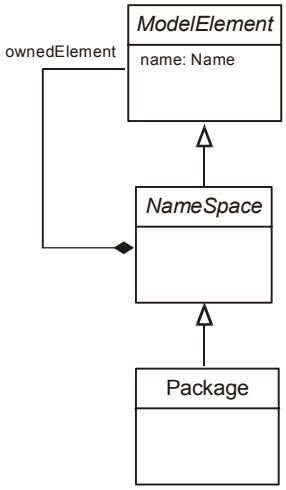

Abbildung 6.3: Die Klasse Package im UML-Metamodell

6.3 Das AKO-Metamodell

6.3.1 Überblick

Das AKO-Metamodell wurde nicht explizit spezifiziert, ergibt sich jedoch implizit aus der Beschreibung der AKO-Schnittstelle und der darin vorkommenden Konstrukte. Die Schnittstelle unterstützt sowohl die kontinuierliche Modellierung eines Domänenmodells, als auch den Zugriff auf Instanzdaten. [Kolbe98]

Sie wurde 1997 vom Arbeitskreis Objekte (AKO) erstmals spezifiziert und in verschiedenen Forschungsprogrammen weiterentwickelt. In diesem Abschnitt wird auf die derzeit vorliegende Version aus dem Jahr 2001 bezug genommen, die für das im SFB524 eingesetzte Modellverwaltungssystem zum Einsatz kommt. Den für den Vergleich mit dem UML-Metamodell relevanten Ausschnitt aus dem AKO-Metamodell zeigt die Abbildung 6.4.

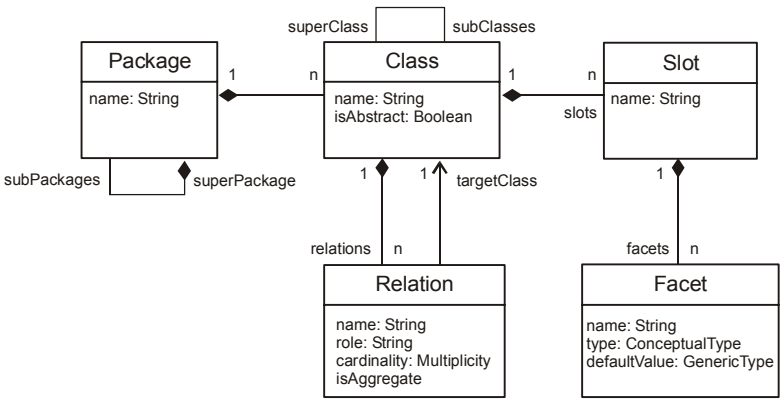

Abbildung 6.4: Ausschnitt aus dem AKO-Metamodell

Die Metaklassen `Instance`, `SlotInstance` und `RelationInstance` werden für den Zugriff auf eine Ausprägung des Datenmodells benötigt. Da jedoch die Möglichkeiten zur Definition bzw. Modellierung eines Datenmodells im Mittelpunkt des Vergleichs der Metamodelle stehen, können diese Metaklassen hier unberücksichtigt bleiben.

6.3.2 Package

Das Package (dt. Paket) dient zur Gruppierung von Klassen. Ein `Package`-Objekt kann `Class`-Objekte und `Package`-Objekte aggregieren. Letztere fungieren dann als Subpackages dieses Package.

6.3.3 Klasse

Ein Klasse repräsentiert die gemeinsamen Eigenschaften einer Menge von Objekten des Anwendungsgebietes. Sie muss einen eindeutigen Namen besitzen und kann eine beliebige Anzahl von Relationen und Slots aggregieren. Zur Abbildung von Spezialisierungsbeziehungen kann einer Klasse eine Superklasse zugeordnet werden. Die Klasse erbt alle Slots und Relationen ihrer Superklasse. Ist eine Klasse als abstrakt gekennzeichnet, so kann sie nicht instanziiert werden.

6.3.4 Relation

Eine Relation modelliert eine unidirektionale Beziehung zwischen den Objekten zweier Klassen. Sie besitzt einen eindeutigen Namen und einen Bezeichner für die Rolle, die die Relation in der Ausgangsklasse innehat, sowie eine Referenz auf die Zielklasse.

Die Anzahl der über diese Relation assoziierten Objekte der Zielklasse lässt sich mit Hilfe des Attributs `cardinality` festlegen. Diesem kann einer der durch den Aufzählungstyp `Multiplicity` festgelegten Werte zugewiesen werden, die in Tabelle 6.1 aufgeführt sind.

Multiplicity-Wert	**Bedeutung**
exactly_one	genau eine Instanz der Zielklasse
zero_or_more	keine oder genau eine Instanz
one_or_more	eine oder mehr Instanzen
zero_or_one	keine oder genau eine Instanz

Tabelle 6.1: Multiplicity-Werte

6.3.5 Slot

Ein Slot steht für eine Eigenschaft, die alle Objekte der zugehörigen Klasse besitzen. Er hat einen eindeutigen Namen und kapselt eine oder mehrere Facetten. Jeder Slot besitzt zumindest eine Wertfacette.

6.3.6 Facette

Eine Facette bildet einen Teilaspekt eines Slots ab. Beispielhaft sei hier die zu einem Maß gehörende Einheit genannt. Jede Facette besitzt einen Namen und einen Datentyp. Ihr kann zudem ein Vorgabewert zugewiesen werden.

6.3.7 Datentypen

AKO definiert eine Reihe von Datentypen, die eine Facette annehmen kann. Sie sind im Metamodell als mögliche Werte des Aufzählungstyps `ConceptualType` definiert. In Tabelle 6.2 sind die verfügbaren atomaren Datentypen aufgeführt.

Datentyp	**Wertebereich**
Boolean	Boolescher Typ mit dem Wert „wahr“ oder „falsch“
Long	Ganze Zahlen im Bereich -2^{32} bis $2^{32}-1$
Byte	Ganze Zahlen im Bereich –128 bis 127
Double	Gebrochene Zahlen mit doppelter Genauigkeit[1]
String	Zeichenkette
URL	Zeichenkette

Tabelle 6.2: Atomare Basisdatentypen

Des weiteren existieren in AKO eine Reihe von Datentypen, die eine Sequenz von atomaren Datentypen bilden (Tabelle 6.3).

Einfacher Datentyp	**Sequenz-Datentyp**
Boolean	BooleanSequence
Long	LongSequence
Byte	ByteSequence
Double	DoubleSequence
String	StringSequence

Tabelle 6.3: Sequenz-Datentypen

Schließlich definiert AKO eine Reihe von komplexen Datentypen. Das sind Typen, die aus einer Zusammensetzung von atomaren Datentyp bestehen. (Tabelle 6.4)

Komplexer Datentyp	**Komponenten**
Point3d	x : Double y : Double z : Double
Blob	format: String data: ByteSequence
Stochastic	params: DoubleSequence flags: BooleanSequence description: String

Tabelle 6.4: Komplexe Datentypen

[1] Siehe *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985

`Blob` steht für *Binary Large Object*. Dieser Datentyp dient zum Aufnehmen sehr großer binärer Datenmengen, wie sie beispielsweise beim Speichern von digitalen Bildern und Filmen auftreten. Der Datentyp `Stochastic` dient zur Aufnahme von stochastischen Werten, er kapselt beispielsweise Erwartungswert und Verteilung.

6.4 Vergleich der Metamodelle

Abbildung 6.5 wird eine mögliche Interpretation der Klassendiagramme aus Kapitel 6.2 gezeigt.

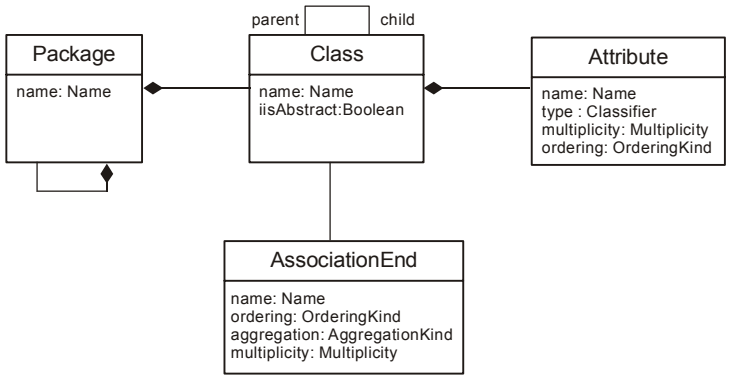

Abbildung 6.5: Vereinfachende Interpretation des UML-Metamodells

Im Vergleich mit Abbildung 6.4 ist deutlich zu erkennen, dass zwischen den Metamodellen von AKO und UML durchaus Äquivalenzen bestehen. Anhand der Tatsache, dass diese Abbildung eine starke Vereinfachung der vorhergehenden darstellt, wird jedoch auch klar, dass mit AKO nur ein kleiner Teil der mit UML formulierbaren Semantik modelliert werden kann.

6.4.1 Package

Das Package dient in beiden Metamodellen zur containerartigen Strukturierung der Klassenmenge. Die Semantik ist in den beiden Metamodellen als weitestgehend identisch anzusehen.

6.4.2 Klassifizierer

Die Semantik des Konstrukts *Klasse* ist in den beiden Metamodellen weitestgehend äquivalent. Die Klasse ist das zentrale Element, sie abstrahiert mit Hilfe objektorientierter Mechanismen die Eigenschaften von Entitäten der realen Welt.

Zum Abbilden von Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen Klassen ist das Modellieren einer Vererbungshierarchie möglich, bei der die Kindklassen die Eigenschaften der Elternklassen erben. Hierfür wird in UML die Metaklasse *Generalization* verwendet, in AKO besitzt die Metaklasse *Class* eine Referenz auf ihre Superklasse.

Mehrfachvererbung ist in UML grundsätzlich erlaubt, AKO lässt dies hingegen nicht zu. In beiden Metamodellen können Klassen als abstrakt markiert werden, wenn keine Instantiierung möglich sein soll. Interfaces sieht das AKO-Metamodell nicht vor.

Die Definition von Aufzählungsdatentypen (engl. enumerations) ist mittels UML möglich, wird von AKO hingegen nicht unterstützt.

6.4.3 UML-Attribute und AKO-Slots

Während UML zur Beschreibung von Eigenschaften die Metaklasse *Attribut* zur Verfügung stellt, nutzt AKO hierzu das Konstrukt *Slot*. Der wesentlichste Unterschied zum *Attribut* besteht darin, dass ein *Slot* aus mehreren sogenannten *Facetten* bestehen kann, die zusätzliche Teilaspekte einer Eigenschaft abbilden können.

Eine *Facette* kann lediglich einen der vordefinierten Datentypen annehmen, der Typ eines *Attributs* kann hingegen jeglicher Klassifizierer im Modell sein. UML erlaubt das Festlegen einer *Vielfalt* (engl. multiplicity) für ein Attribut. Damit wird die Anzahl an Werten bestimmt, die der Ausprägung des Attributs zugewiesen werden kann. In AKO stehen zu diesem Zweck die Sequenzdatentypen zur Verfügung (Tabelle 6.4). Ist einer Facette ein solcher Datentyp zugewiesen, kann die zugehörige Facetteninstanz eine Sequenz von Werten aufnehmen. Allerdings kann anders als beim UML-Konstrukt *Multiplicity* keine Aussage über die exakte Zahl von Werten getroffen werden.

UML lässt das überschreiben von Attributen in Subklassen zu. Dabei kann dem selben Namen ein anderer Typ zugeordnet werden. In AKO wird dieses Konzept nicht unterstützt.

6.4.4 UML-Assoziationen und AKO-Relationen

Zur Beschreibung einer Beziehung zwischen zwei Klassen verwendet UML die Metaklasse *Association* und AKO das Konstrukt *Relation* mit nahezu identischer Semantik.

Während *Relationen* in AKO grundsätzlich unidirektional sind, d.h. nur in eine Richtung traversiert werden können, können UML-Assoziationen bidirektional sein. Für eine UML-Assoziation kann angegeben werden, ob die assoziierten Objekte geordnet sein sollen. Zudem erlaubt die UML die Verwendung von *Assoziationsklassen,* die ihrerseits Attribute und Operationen zur Beschreibung von Eigenschaften und Verhalten besitzen.

Für eine Assoziation kann ein *qualifizierendes Attribut* angegeben werden. Mit dessen Hilfe kann bei Abfragen die Menge der assoziierten Objekte auf die gesuchten eingegrenzt werden. Das Konzept der *Kardinalität* von AKO-Relationen ist mit dem der *Multiplicity* von UML-Assoziationen vergleichbar. Letzteres erlaubt jedoch viel präzisere Angaben: Während in AKO lediglich das Festlegen auf eine der vier Multiplicity-Werte (siehe Tabelle 6.1) möglich ist, kann in UML auch ein spezifischer Werte (z.B. 6) bzw. einzelne Werte für obere und untere Grenzen (z.B. 6..12) angegeben werden.

Der Begriff *Relation* taucht auch in der UML-Spezifikation auf, hat dort aber eine andere Bedeutung: Er beschreibt sowohl Vererbungs- als Assoziationsbeziehungen. Dies spiegelt sich im UML-Metamodell dadurch wieder, dass die Metaklasse `Relationship` Superklasse der Metaklassen `Generalization` und `Association` ist. (siehe Abbildung 6.2)

6.4.5 Verhalten

Die UML definiert das Konstrukt *Operation*, mit dessen Hilfe das Verhalten von Objekten abstrahiert werden kann. Mit Hilfe von Operationen können den Objekten Botschaften gesandt werden, die zu einer Änderung ihres inneren Zustands führen und/oder zu einer Antwort. UML-Operationen besitzen *Parameter* und können *Exceptions* werfen.

Das AKO-Metamodell besitzt kein zur *Operation* äquivalentes Konstrukt, stellt aber implizit Methoden zum Abfragen und Setzen von Facettenwerten zur Verfügung.

6.4.6 Datentypen

Datentypen können bzw. müssen mit AKO nicht explizit modelliert werden, sondern werden implizit vom Metamodell zur Verfügung gestellt. Die UML definiert keine konkreten Datentypen, da diese meist durch die Implementierung vorgegeben werden, sondern lediglich die dazugehörige Metaklasse *Type*. Da OCL jedoch Basisdatentypen benötigt, werden diese in der OCL-Spezifikation gesondert definiert (vgl. Kap. 5.2.4.1).

6.5 Fazit und Konsequenzen

Zwischen den Metamodellen von AKO und UML bestehen einige Unterschiede, vor allem hinsichtlich des Umfangs der Modelliermöglichkeiten. Vor allem wegen des Fehlens von Facetten im UML-Metamodell ist OCL nicht direkt für AKO-basierte Modellverwaltungssysteme verwendbar.

Wegen der ebenfalls zu konstatierenden Übereinstimmungen in großen Bereichen ist es jedoch sinnvoll, eine Sprache zu definieren, die eine weitestgehende syntaktische und semantische Übereinstimmung mit der OCL aufweist, diese aber einerseits um zusätzliche Konstrukte (wie zum Zugriff auf Facetten) erweitert und andererseits die nicht benötigten Elemente von OCL (wie Prä- und Postkonditionen) nicht beinhaltet. Die im Zuge dieser Arbeit entwickelte Constraint Modeling Language (CML) erfüllt diese Anforderungen und kann daher als OCL-Dialekt aufgefasst werden. Die CML soll im folgenden Kapitel vorgestellt werden.

7 Die Constraint Modeling Language (CML)

7.1 Zielstellungen beim Entwurf

Wie aus dem vorangegangen Kapitel hervorgeht, zwingen die Unterschiede in den Metamodellen von AKO und UML zum Entwurf einer eigenen Constraint-Sprache, die für den Einsatz in AKO-basierten Modellverwaltungssystemen geeignet ist.

Beim Entwurf der Constraint-Sprache CML stellten sich folgende Prämissen in der Reihenfolge ihrer Priorität:

1. CML muss kompatibel mit dem AKO-Metamodell sein.
2. CML soll so wenig wie möglich von OCL divergieren, um eine eventuell später stattfindende Migration des MVS vom AKO- auf das UML-Metamodell zu erleichtern.
3. Ziel bei der Entwicklung der Sprache und ihrer prototypischen Umsetzung im Zuge dieser Arbeit ist weniger die Erzielung eines möglichst mächtigen Funktionsumfangs, als vielmehr der Nachweis der grundsätzlichen Eignung.

Da die Entwicklung von CML gemäß diesen Prämissen stark von OCL geprägt ist, werden zur Beschreibung des CML-Sprachumfangs in folgendem vor allem die Unterschiede zwischen den beiden Constraint-Sprachen beschrieben.

7.2 Beispiele

Die CML-Beispiele in diesem Kapitel beziehen sich auf den in Abbildung dargestellten Ausschnitt aus einem Domänenmodell.

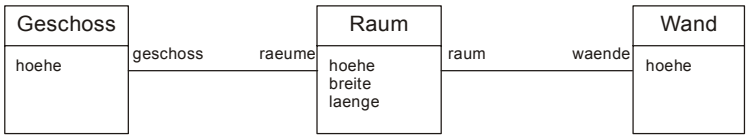

Abbildung 7.1: Ausschnitt aus einem Domänenmodell für die Beispiele

7.3 Constraint-Stereotypen

Da AKO nicht über das Konstrukt *Operation* oder ein semantisch äquivalentes Konstrukt verfügt, erübrigt sich die Notwendigkeit zur Definition von Vor- bzw.

Nachbedingungen. In CML existieren daher die Constraint-Stereotypen *precondition* und *postcondition* nicht.

OCL definiert weiterhin die Stereotypen *definition* und *invariant*. Dabei stellen die *defintion*-Ausdrücke keine Constraints im engeren Sinne dar, sondern dienen zur Definition von Pseudoattributen, auf die in regulären Constraint-Ausdrücken zugegriffen werden kann. Die Definition von Pseudoattributen mit Hilfe von `let`-Ausdrücken bzw. `defintion`-Constraints ist zwar grundsätzlich eine interessante Möglichkeit, um einfache Operationen im Bauwerksmodell abzubilden und die nötigen Berechnungen vom Modellverwaltungssystem durchführen zu lassen. Beispielsweise ist die Fläche eines Raumes ein geeigneter Kandidat für ein Pseudoattribut, dessen Berechnung sich aus den „echten“ Attributen Breite und Höhe ergibt. Wegen der semantischen Divergenz hinsichtlich der Auswertung und des Zugriffs sollten Pseudoattribute jedoch nicht als Constraints, sondern separat implementiert werden.

Im Ergebnis wird in CML zunächst nur ein Constraint-Stereotyp zugelassen: `Invariant`. Eine explizite Kennzeichnung des Constraint-Ausdrucks als `Invariant` ist daher in CML im Unterschied zu OCL nicht nötig. CML verfügt demzufolge nicht über die Schlüsselwörter `@pre`, `post`, `inv` und `def`. Entsprechend ist die Festlegung des Stereotyps zu Beginn eines Constraint-Ausdrucks nicht notwendig.

7.4 Kontext

Der `context`-Ausdruck am Anfang eines OCL-Constraints wurde eingeführt, um alle zu einem UML-Modell gehörenden Constraints in einer einzigen Datei speichern zu können. CML-Ausdrücke sind jedoch nicht für das Speichern in einer Datei bestimmt, sondern werden von einem Constraint-Objekt gekapselt, das immer mit einer konkreten Klasse bzw. Instanz assoziiert ist (siehe Kapitel 8.2). Die Angabe des Kontextes kann daher in CML entfallen.

In den Beispielen diese Kapitels wird die Kontextklasse durch einen unterstrichenen Namen angegeben. Dieser ist nicht Teil des CML-Ausdrucks.

7.5 Schlüsselwörter

Folgende Zeichenketten sind in CML als Schlüsselwörter reserviert:

```
if  then  else  endif
not  or  and  xor  implies
```

7.6 Kommentare

Kommentare im Constraint-Ausdruck entfallen, da sie ggf. gesondert in einem Attribut der Constraint-Klasse verwaltet werden können.

7.7 Basisdatentypen und –operationen

7.7.1 Typen

Für jeden in OCL verfügbaren Basisdatentyp existiert ein äquivalenter CML-Typ, wie Tabelle 7.1 zeigt. Der Wertebereich der einzelnen Typen ist für die Definition einer Constraint-Sprache unerheblich. Entsprechend werden die Typen `Byte` und `Long` gleich behandelt.

OCL	CML
Boolean	Boolean
Integer	Long, Byte
Real	Double
String	String

Tabelle 7.1: Die Basisdatentypen von CML im Vergleich mit denen von OCL

Der Zugriff auf eine Facette mit dem AKO-Typ `URL` resultiert in einen OCL-String. Die in AKO definierten komplexen Basisdatentypen `AKO_Point3d`, `AKO_Blob` und `AKO_StochValue` werden zunächst nicht in den Sprachumfang von CML aufgenommen. Ein Zugriff auf eine Facette mit einem dieser Typen gilt als undefiniert.

Da das AKO-Metamodell das explizite Ordnen von Assoziationsenden bzw. Attributen nicht vorsieht, stehen in CML die Kollektionstypen `Bag`, `Set` und `Sequence` nicht zur Verfügung, sondern nur deren Superklasse `Collection`.

CML-Ausdrücke, die auf eine Facette mit einem Sequenz-Typ zugreifen, resultieren in ein Kollektion von Werten des zugehörigen Basisdatentyps. Navigationsausdrücke entlang von Relationen mit der Kardinalität `one_or_more` oder `zero_ore_more` resultieren in eine Kollektion von Instanzen der Zielklasse.

7.7.2 Vergleichsoperationen

Die Vergleichsoperationen < > und = sind für die Typen String, Double, Byte und Long definiert. Ergebnis ist ein Boolean-Wert.

7.7.3 Addition und Subtraktion

Die Operationen Addition und Subtraktion sind für alle Zahlentypen, also Byte, Long und Double definiert. Der Ergebnistyp hängt von den Typen der Operanden ab, wie Tabelle 7.2 zeigt.

Operand 1	Operand 2	Ergebnis
Long	Long	Long
Double	Double	Double
Byte	Byte	Long
Long	Byte	Long
Long	Double	Double
Double	Byte	Double

Tabelle 7.2: Ergebnistypen bei Addition und Subtraktion

7.7.4 Multiplikation

Multiplikation und Division sind ebenfalls für alle Zahlentypen definiert. Die Ergebnistypen der Multiplikation sind so gewählt, dass kein Verlust von Präzision eintritt: sobald einer der Operanden eine Gleitkommazahl ist, ist der Ergebnistyp ebenfalls eine Gleitkommazahl. (Tabelle 7.3)

Faktor	**Faktor**	**Produkt**
Long	Long	Long
Double	Double	Double
Byte	Byte	Long
Long	Byte	Long
Long	Double	Double
Double	Byte	Double

Tabelle 7.3: Ergebnistypen bei Multiplikation

Der Ergebnistyp der Division ist immer eine Gleitkommazahl. (Tabelle 7.4)

Dividend	**Divisor**	**Quotient**
Long	Long	Double
Double	Double	Double
Byte	Byte	Double
Long	Byte	Double
Long	Double	Double
Double	Byte	Double

Tabelle 7.4: Ergebnistypen bei Division

7.7.5 Sonstige Operationen

Alle anderen in OCL für die Basisdaten definierten Operationen (Kapitel 5.2.4.1) wurden vorerst nicht in den Sprachumfang von CML aufgenommen.

7.7.6 Infix- und Postfix-Schreibweise

In CML sind wie in OCL die folgenden Operatoren in Infix-Notation zugelassen:

```
* , / , + , - , and, or , xor , not
```

Allerdings wird anders als in OCL die Postfix-Notation dieser Operatoren nicht unterstützt, da dadurch der Sprache keine zusätzliche Funktion hinzufügt wird und die Postfix-Notation dieser Operatoren ungebräuchlich ist.

7.8 Vorrangregeln

Die Vorrangregeln bei der Ausführung von Operationen sind identisch mit denen von OCL bis darauf, dass das `@pre`-Konstrukt an höchster Stelle entfällt (vgl. Kapitel 5.2.6).

7.9 Aufzählungstypen

Eine Unterstützung von Aufzählungstypen (enumerations) durch CML ist nicht notwendig, da diese nicht Bestandteil des AKO-Metamodells sind.

7.10 Eigenschaften von Instanzen

Der Zugriff auf Eigenschaften von Instanzen geschieht in CML äquivalent zu OCL mit Hilfe des Punktoperators. In CML ist ebenso wie in OCL die Angabe des Schlüsselworts `self` beim Zugriff auf eine Eigenschaft der Kontextinstanz optional.

7.10.1 Zugriff auf Slots

Der Zugriff auf einen Slot resultiert immer in dem Wert, mit dem die Value-Facette belegt ist. Der Zugriff auf einen Slot kann durch *explizite* Angabe der Instanz geschehen, gefolgt vom Punkt-Operator und dem Namen des Slots:

```
Raum
self.hoehe > 200
self.geschoss.hoehe > 200
```

Soll auf einen Slot der Kontext-Instanz zugegriffen, kann die Angabe das Schlüsselworts `self` entfallen. Der Zugriff erfolgt dann *implizit* allein durch Angabe des Slotnamens, wie zum Beispiel in folgendem Ausdruck:

```
Raum
hoehe > 200
```

7.10.2 Zugriff auf Facetten

Da das AKO-Metamodell das Konstrukt *Facette* definiert, musste in CML eine Möglichkeit geschaffen werden, um auf die Wertbelegung einer Facette zugreifen zu können. Dies geschieht analog zum Zugriff auf Slots bzw. Relationen mit Hilfe des Punkt-Operators. Vor dem Punkt muss der Name des Slots stehen, dahinter der Name der Facette. Zum Beispiel:

```
Raum
self.hoehe.max = 300
self.hoehe.min = 190
self.hoehe.units = 'cm'
```

Da in den meisten Fällen ein Zugriff auf die Value-Facette erwünscht ist, ist die Angabe eines Facettennamens optional: Wird keine Facette angegeben, so resultiert der Ausdruck automatisch im Wert der Value-Facette des zuvor angegebenen Slots.

Beispielweise gilt immer:

```
Raum
hoehe = hoehe.Value
```

7.10.3 Zugriff auf Relationen

Wegen der semantischen Nähe zwischen der Metaklasse *Association* im UML-Metamodell und der Metaklasse *Relation* im AKO-Metamodell entspricht die Art des Zugriffs auf eine Relation in CML im wesentlichen dem Zugriff auf eine Assoziation in OCL. Beispielsweise resultiert die Auswertung des folgenden Ausdrucks in einer Kollektion von `Raum`-Objekten, die über die Rolle `raeume` mit einem `Geschoss`-Objekt assoziiert sind.

```
Geschoss
self.raeume
```

Anders aber als bei OCL ist das Ergebnis einer Navigation über eine Relation, der die Kardinalität „0 oder 1“ zugewiesen ist, keine Kollektion, sondern eine einzelne Instanz. Entsprechend stehen an dieser Stelle die für Kollektionen definierten Operationen (`size()`, `exists()` usw.) nicht zur Verfügung. Das heißt, dass beispielsweise der folgende OCL-Ausdruck in CML nicht gültig ist:

```
Raum
geschoss.size > 0
```

Der Vorteil dieser Vorgehensweise ist jedoch, dass direkt auf die Slots bzw. die Relationen dieser Instanz zugegriffen werden kann, wie in folgendem Ausdruck:

```
Raum
geschoss.hoehe = self.hoehe
```

7.11 Kollektionen

Der Zugriff auf eine Facette, deren Typ einer der Sequenz-Datentypen ist (Tabelle 6.2), resultiert in eine Kollektion von atomaren Datentypen. Die Navigation entlang einer Relation, der die Kardinalität „0 oder mehr“ oder „1 oder mehr“ zugewiesen ist, resultiert in eine Kollektion von Instanzen der Zielklasse.

Für Kollektionen wurden lediglich die wichtigsten Operationen in den Sprachumfang von CML aufgenommen. Das sind im einzelnen (vgl. Kapitel 5.2.13) :

```
select
forAll
exists
size
isEmpty
```

Die folgenden Operationen aus dem Sprachumfang von OCL, die nicht in CML enthalten sind, lassen sich durch die Verknüpfung von logischen Operatoren mit den oben aufgeführten Operationen simulieren:

```
reject
count
empty
notEmpty
```

Wie Kapitel 5.2.13.3 beschreibt, sind in OCL für über Kollektionen iterierende Operationen drei verschiedene syntaktische Varianten zugelassen. In CML sind nur die beiden folgenden Varianten erlaubt:

```
collection->operation( element | <expression> )
collection->operation( <expression> )
```

Bei der dritten Variante wird der Typ der zu iterierenden Elemente explizit deklariert. Diese Variante wurde nicht in den Sprachumfang von CML aufgenommen, da der Typ bereits implizit festgelegt ist. Eine explizite Deklaration ist nicht notwendig und stellt eine potentielle Fehlerquelle dar.

Während OCL die Benutzung des Pfeilsymbols beim Aufruf einer für Kollektionen definierten Operation zwingend vorschreibt, werden in CML an dieser Stelle sowohl das Pfeilsymbol als auch der Punkt akzeptiert.

Beispiel:

```
Geschoss
raueme->forAll( hoehe>0 )
raueme.forAll( hoehe>0 )
```

7.12 Für alle Typen definierte Operationen

OCL besitzt den Typ `OclAny` als gemeinsamen Supertyp für alle UML-Klassifizierer und alle Basisdatentypen. CML besitzt keinen solchen Supertyp und damit keine für alle Typen definierten Operationen.

`OclAny` definiert die beiden Operationen `oclInState()` und `oclIsNew()`, die Auskunft über den Status des Objekts geben. Da das AKO-Metamodell die Modellierung von Objektzuständen nicht erlaubt, werden diese Operationen in CML nicht bereitgestellt.

Nebenbedingungen, die für die Metaebene gelten, werden zum großen Teil implizit vom AKO-basierten MVS geprüft und in einer Anwendungsdomäne nur selten benötigt. In CML werden daher die in OclAny definierten Operationen, die zur Typbestimmung und zum expliziten Casting eines Objektes dienen, nicht zur Verfügung gestellt. Es entfallen die folgenden Operationen:

```
oclType()
oclIsKindOf()
oclIsTypeOf()
oclAsType()
```

In einigen Fällen muss für die Formulierung einer Zwangsbedingung in einer Domäne dennoch auf die Metaebene zugegriffen werden können. Beispielhaft sei hier die unterschiedliche Behandlung von verschiedenen Subtypen der Zielklasse einer Relation genannt. Zum Beispiel resultiert der folgende OCL-Ausdruck in einer Kollektion von `Fenster`-Objekten, für die speziellere Bedingungen gelten können als generell für Aussparungen.

```
Aussenwand
aussparungen->select( oclIsKindOf(Fenster) )
```

Die Formulierung derartiger Bedingungen wird von CML im ersten Entwurf nicht unterstützt. Diese Funktionalität ist aber bei einer späteren Erweiterung des Sprachumfangs unbedingt zu berücksichtigten.

Wegen dem Wegfall der Operationen zur Typbestimmung steht ebenfalls die OCL-Metaklasse `OclType` in CML nicht zur Verfügung. Mit ihr entfallen auch die für sie definierten Attribute und Operationen (vgl. Kapitel 5.2.15).

7.13 Typangaben und Package-Pfade

Da die in CML formulierbaren Constraints nur für Instanzen gelten sollen und kein Zugriff auf die Metaebene möglich ist, ist in CML-Ausdrücken keine Typangabe notwendig. Damit kann auch die Möglichkeit zur Angabe von Package-Pfaden entfallen.

In OCL werden alle Typbezeichner groß geschrieben. Da in CML werden keine Typnamen benötigt werden, ist die Großschreibung für die Namen von Relation, Slots und Facetten grundsätzlich zugelassen.

7.14 Einschränkungen

CML ist ebenso wie OCL eine rein deklarative Sprache und verfügt demzufolge nicht über explizite Ablaufkontrollstrukturen wie Schleifen und Verzweigungen.

Weil komplexe Berechnungen jedoch genau solche Kontrollstrukturen benötigen, sind sie nicht mit Hilfe der CML durchführbar.

7.15 Die CML-Grammatik

Da die Entwicklung einer Grammatik aufwendig und kompliziert ist und der Sprachumfang von CML im Prinzip[1] als eine Teilmenge des Sprachumfangs von OCL gelten kann, wurde bei der Entwicklung der CML-Grammatik zu großen Teilen auf die OCL-Grammatik zurückgegriffen. Aus dem gegenüber OCL verringerten Sprachumfang von CML folgt jedoch, dass nur ein Teil der OCL-Grammatik verwendet wird.

Im Anhang B1 findet sich die OCL-Grammatik, zum Vergleich ist die Grammatik von CML im Anhang B2 zu finden.

Da CML-Constraints nicht in einem separaten File gespeichert werden, sondern die einzelnen Constraint-Ausdrücke immer mit einem Constraint-Objekt assoziiert sind, können die Produktionen `oclFile`, `oclExpression`, `constraint` und `contextDeclaration` entfallen. Da des weiteren in CML nur Constraints mit Stereotyp `invariant` erlaubt sind, kann die Produktion `stereotype` entfallen. Weil `let`-Ausdrücke zur Definition von Pseudoattributen in CML nicht zugelassen sind, entfallen die Produktionen `letExpression` und `oclExpression`.

Da in CML weder Vor- noch Nachbedingungen für Operationen gebraucht werden, kann weiterhin die Produktion `operationContext`, und mit ihr auch die Produktionen `operationName`, `formalParameterList`, `typeSpecifier` und `returnType` entfallen.

Im Ergebnis ist die `expression`-Produktion das Startsymbol der CML-Grammatik. Alle vorher in der OCL-Grammatik aufgelisteten Produktionen entfallen. Außerdem wurde wegen der in CML fehlenden Aufzählungstypen die Produktion `enumLiteral` gestrichen.

Die Angabe von qualifizierenden Attributen wird durch das AKO-Metamodell nicht unterstützt, entsprechend kann die Produktion `qualifiers` entfallen. Zeitlich Ausdrücke haben keine Bedeutung in CML, da die Formulierung von Vor- und Nachbedingungen nicht vorgesehen ist. Daher kann die Produktion `timeExpression` entfallen. Auch die Angabe von Pfaden ist in CML nicht notwendig, die Produktion `pathName` kann entfallen und in anderen Produktionen zu `name` vereinfacht werden. Im Ergebnis wurde die Produktion `propertyCall` stark vereinfacht zu

```
propertyCall := name "(" ( (declarator())? expression() )? ")"
```

Entsprechend konnten die Produktionen `propertyCallParameters` und `actualParameterList` entfallen. Letztere deswegen, weil in CML einem Eigenschaftszugriff immer nur genau ein Ausdruck als Parameter übergeben werden kann, in OCL hingegen eine Liste von Komma-separierten Parametern.

---

[1] ausgenommen der Möglichkeit des Zugriffs auf Facetten

Das OCL-Terminal `number` wurde in eine Produktion umgeformt, die zwischen Gleit- und Festkommazahlen unterscheidet. Entsprechend wurden die gesonderten Terminale `INTEGER_LITERAL` und `FLOATING_POINT_LITERAL` definiert.

Da der Zugriff auf die Facette eines Slots von der CML-Grammatik als `propertyCall` behandelt wird, ist die CML-Grammatik eine echte Teilmenge der OCL-Grammatik. Die CML-Grammatik ist ebenso wie die OCL-Grammatik eine LL(2)-Grammatik. Das heißt, dass beim Parsen genau ein weiteres Token im Voraus betrachtet werden muss, um an bestimmten Stellen der Grammatik eine Entscheidung fällen zu können. Dies resultiert aus dem potentiellen Entscheidungskonflikt (engl. choice conflict) zwischen den folgenden beiden Produktionen:

```
propertyCall -> <name> ( "(" ( (declarator)? expression )? ")"
declarator -> <name> "|"
```

Da die `expression`-Produktion letztlich auch mit einem `<name>`-Token beginnen kann, kann ein LL(1)-Parser beim Auffinden eines `<name>`-Tokens nicht entscheiden, ob es sich um den Beginn eines `expression`- oder eines `declarator`-Ausdrucks handelt. Um dies entscheiden zu können, muss ein weiteres Token betrachtet werden: wird ein senkrechter Strich (`|`) gefunden, kann `<name>` eindeutig dem `declarator`-Ausdruck zugeordnet werden.

7.16 Zusammenfassung

Mit der *Constraint Modeling Language* (CML) konnte eine Sprache zur Formulierung von Zwangs- bzw. Nebenbedingungen in Domänenmodellen entwickelt werden, die durch eine Ausprägung des AKO-Metamodells repräsentiert werden. Bei CML handelt es sich um einen Dialekt der Sprache OCL, der die Syntax und damit die intuitive Verwendbarkeit von seinem Vorbild OCL erbt.

In den Sprachumfang von CML wurden zunächst nicht alle Elemente von OCL übernommen; zum Teil, weil die Semantik einiger Konstrukte nicht vom AKO-Metamodell unterstützt wird, und zum Teil, weil es sich hier um einen ersten Entwurf handelt, der vor allem die grundsätzliche Eignung von CML zur Formulierung von Zwangs- bzw. Nebenbedingungen belegen soll.

CML bietet aber dennoch umfangreiche Möglichkeiten zur Formulierung von komplexen Bedingungsausdrücken. Dazu gehören der Zugriff auf facettierte Attribute, die Navigation entlang von Assoziationsbeziehungen, Verknüpfungen durch arithmetische und logische Operatoren, Vergleichsoperationen sowie konditionale Konstrukte. Besonders anwenderfreundlich sind die für Kollektionen vordefinierten Operationen `forAll()`, `select()` und `exists()`, die für jedes Element der Kollektion einen CML-Ausdruck auswerten und entweder einen booleschen Wert oder eine Untermenge der Kollektion zurückliefern.

8 Umsetzung des Constraint-Moduls

8.1 Übersicht

Im Rahmen dieser Arbeit wurde ein Constraint-Modul für das im SFB 524 im Einsatz befindliche Modellverwaltungssystem entwickelt. Das Modul dient der laufzeitdynamischen Definition von Constraints in Domänenmodellen und deren Auswertung für einzelne Instanzen.

Sowohl die Verwaltung der Constraints als auch die Interpretation der Constraint-Ausdrücke wird vom MVS-Server realisiert. Das hat zum einen konzeptionelle Gründe: Die Constraints sind Teil des Domänenwissens bzw. des Projektwissens und eng mit dem Domänenmodell verflochten. Zum anderen muss eine potentiell sehr große Menge von Instanzen auf Einhaltung der für sie gültigen Constraints geprüft werden. Wird die Constraint-Prüfung im selben Adressraum durchgeführt, in dem diese Daten liegen, kann ein ressourcenintensiver Transport über das Netzwerk entfallen. Nur durch eine hohe Performanz des Constraint-Moduls ist der mit einer Constraint-Prüfung verbundene Zeitaufwand für den Nutzer akzeptabel und eine hohe Frequenz von Constraint-Prüfungen möglich.

Dieses Kapitel wird die einzelnen Schritte der Umsetzung des Constraint-Moduls behandeln: die Erweiterung des Metamodells, die Erweiterung der AKO-Schnittstelle und die Implementation der Funktionalitäten des Constraint-Moduls im Kern des MVS-Servers. Wichtigster und aufwendigster Teil war die Umsetzung eines CML-Interpreters, dessen Aufgabe die Auswertung von CML-Ausdrücken ist.

8.2 Erweiterung des AKO-Metamodells

Ziel der Umsetzung des Constraint-Moduls war es, AKO-basierte Modellverwaltungssysteme um Funktionalitäten zur Formulierung von Constraints und zur Prüfung von Instanzdaten auf Einhaltung dieser Constraints zu erweitern. Dazu musste zunächst das AKO-Metamodell an die sich daraus ergebenden Anforderungen (vgl. Kapitel 4.7) angepasst werden.

Das AKO-Metamodell wurde um die Metaklasse `Constraint` erweitert. Ein `Constraint`-Objekt besitzt einen eindeutigen Namen und kapselt den Constraint-Ausdruck, der als String gespeichert wird. (Abbildung 8.1)

Es mussten zwei Typen von Constraints Berücksichtigung finden: Solche, die Teil des Domänenwissens und daher mit einer Klasse des Domänenmodells assoziiert sind, und solche, die Teil des projektspezifischen Wissens sind und daher mit einer

konkreten Instanz verknüpft sind. Entsprechend kann eine Instanz der Metaklasse `Constraint` entweder mit einem `Class`-Objekt oder mit einem `Instance`-Objekt assoziiert sein.

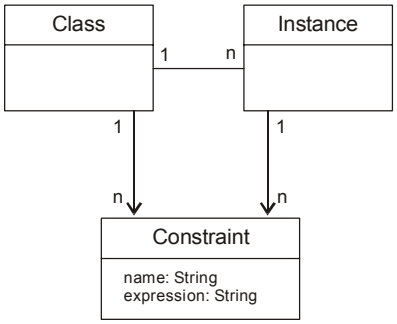

Abbildung 8.1: Erweiterung des AKO-Metamodells um die Klasse Constraint

Alle einer Klasse zugeordneten Constraints werden an deren Subklassen vererbt. Daraus folgt, dass der angegebene Name nicht nur unter den Constraints einer Klasse eindeutig sein muss, sondern auch unter den Namen der Constraints aller ihrer Superklassen.

8.3 Erweiterung der AKO-API

Die AKO-API ist eine in CORBA[1] IDL[2] definierte, verteilte Programmierschnittstelle, die das Modellverwaltungssystem seinen Clients anbietet, damit sie Ausprägungen des AKO-Metamodells (also Domänenmodelle und deren Instanzen) erzeugen und manipulieren können. Sie wurde entsprechend den Erweiterungen am Metamodell angepasst, wodurch Clients des MVS auf die Funktionalität des Constraint-Moduls zugreifen können. Bei den Erweiterungen an der API wurde besonderes Augenmerk darauf gelegt, dass sie gut mit der bereits vorhandenen Schnittstelle harmonieren.

Der AKO-API wurde das Interface `AKO_Constraint` hinzugefügt und die bereits enthaltenen Interfaces `AKO_Class` und `AKO_Instance` wurden um zusätzliche Methoden erweitert. Die exakten Signaturen der zur AKO-Schnittstelle hinzugefügten Methoden in IDL-Notation sind im Anhang B3 zu finden.

Das neue Interface `AKO_Constraint` definiert die Operationen `GetExpression()` und `SetExpression()`, mit deren Hilfe der CML-Ausdruck eines Constraints gesetzt bzw. gelesen werden kann. Beim Setzen wird der Ausdruck automatisch auf seine syntaktische und semantische Gültigkeit überprüft. Wird ein Fehler entdeckt, so wird eine `AKO_Exception` mit dem Exception-Code `EXPRESSION_NOT_VALID` geworfen.

Da Constraints mit einer Klasse assoziiert werden können, wurde das Interface `AKO_Class` um Methoden zum Erzeugen (`CreateConstraintByString`) und zum Löschen (`DeleteConstraintByString`) von Constraints erweitert. Beiden

[1] Common Object Request Broker Architecture

[2] Interface Definition Language

Operationen muss der Name des zu erzeugenden bzw. des zu löschenden Constraints übergeben werden. Sie liefern ein Constraint zurück, dass automatisch mit der Klasse assoziiert ist.

Die Operation `GetConstraints()` dient dazu, alle mit einer Klasse assoziierten Constraints aufzufinden. Wenn mit Hilfe des Constraint-Namens eine Referenz auf ein Constraint-Objekt erlangt werden soll, kann die Operation `GetConstraintByString()` verwendet werden. Beide Operationen besitzen einen Parameter, der den Scope festlegt. Wird für den diesen Parameter der Wert `GLOBAL_SCOPE` übergeben, werden auch die von den Superklassen geerbten Constraints zurückgegeben, bei Übergabe von `LOCAL_SCOPE` nur die klassen-eigenen Constraints.

Das Interface `AKO_Instance` wurde ebenso wie das Interface `AKO_Class` um die Operationen `CreateConstraintByString` und `DeleteConstraintByString` erweitert. Sie dienen zum Erzeugen und Löschen eines Constraints, das mit einer einzelnen Instanz assoziiert ist. Weiterhin wurde die Operation `evaluateConstraint()` zum Prüfen eines einzelnen Constraints und die Operation `evaluateConstraints()` zum Prüfen aller für diese Instanz gültigen Constraints eingeführt. Beim Aufruf der letzteren werden sowohl die Constraints geprüft, die mit der zugehörigen Klasse assoziiert sind, als auch jene Constraints, die nur mit dieser Instanz assoziiert sind. Beide Operationen geben einen booleschen Wert zurück.

8.4 Erweiterung des MVS-Kerns

Der Kern des Modellverwaltungssystems wurde um die Implementation der zur AKO-Schnittstelle hinzugefügten Operationen erweitert. Abbildung 8.2 zeigt die neu hinzugekommen Methoden der Klassen `Class` und `Instance` und die Klasse `Constraint`.

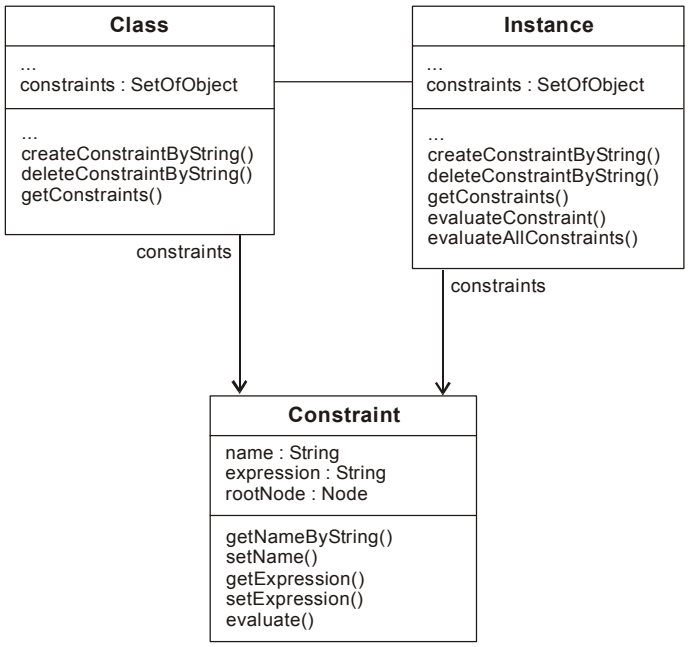

Abbildung 8.2: Erweiterungen am Kern des Modellverwaltungsservers

Die Klasse `Constraint` bildet das Kernstück des Constraint-Moduls. Sie implementiert die Operationen, die im Interfaces `AKO_Constraint` definiert wurden, und bildet die Schnittstelle zum CML-Interpreter. Eine Instanz dieser Klasse repräsentiert ein Constraint im MVS und kapselt den zugehörigen CML-Ausdruck, der als String verwaltet wird. Beim Zuweisen durch Aufruf der Methode `setExpression()` wird der Ausdruck geparst und auf seine semantische Korrektheit überprüft. Ist die Überprüfung erfolgreich, wird eine Referenz auf den Wurzelknoten des beim Parsen erzeugten Syntaxbaums im `Constraint`-Objekt gespeichert. Auf diesen wird bei der Auswertung des CML-Ausdrucks für eine Instanz durch Aufruf der Methode `evaluateConstraint()` zugegriffen. (näheres in Kapitel 8.5)

Die Klasse `Constraint` wurde als persistent deklariert, damit die `Constraint`-Objekte in der angeschlossenen objektorientierten Datenbank gespeichert werden und auch nach Herunterfahren und Neustart des Servers zur Verfügung stehen. Der Syntaxbaum wird jedoch nicht persistent gespeichert, da dies die interne Struktur der Datenbasis unnötig verkompliziert und der Aufwand zum Parsen des CML-Ausdrucks einmal nach jedem Serverstart als vertretbar anzusehen ist.

Die zur Klasse `Instance` hinzugefügten Methoden `evaluateConstraint()` und `evaluateAllConstraints()` verwenden die Methode `evaluateConstraint` der `Constraint`–Klasse.

Die mit einer Klasse assoziierten `Constraint`-Objekte werden in einem Set gespeichert, auf das das betreffende `Class`-Objekt eine Referenz besitzt. Die mit einer Instanz assoziierten `Constraint`-Objekte werden entsprechend in einem Set verwaltet, auf das das betreffende `Instance`-Objekt eine Referenz besitzt.

Wesentliche Erweiterung wurden an der Klasse `GenericType` vorgenommen. Es wurde eine ganze Reihe von Methoden zur Überprüfung der Typverträglichkeit in CML-Operationen und Methoden zur Ausführung von arithmetischen und booleschen Operationen auf `GenericType`-Werten implementiert. Diese Funktionalitäten können auch außerhalb des Constraint-Moduls, beispielsweise bei der Umsetzung von *Operationen* für Klassen des Domänenmodells verwendet werden.

8.5 Umsetzung von CML-Checker und CML-Interpreter

8.5.1 Überblick

Damit die in CML definierten Ausdrücke vom System verarbeiteten werden können, müssen diese zunächst lexikalische und die syntaktisch analysiert werden. Bei der lexikalischen Analyse wird der Strom von Eingabezeichen in einen Strom von lexikalischen Symbolen (engl. tokens) umgewandelt, der dann die Grundlage für die syntaktische Analyse (engl. parsing) bildet.

In [Aho88] wird die Syntaxanalyse wie folgt definiert:

„Syntaxanalyse ist der Prozess, der entscheidet, ob ein aus Symbolen bestehendes Wort[1] von einer Grammatik erzeugt werden kann. Während der Syntaxanalyse wird ein Syntax-Baum aufgebaut.“

Das Parsen eines CML-Ausdrucks findet auf Grundlage der CML-Grammatik statt. Der dabei entstehende Syntaxbaum wird auch genutzt, um den Ausdruck semantisch zu prüfen. Diese Prüfung wird hier als *Checking* bezeichnet und beinhaltet Typüberprüfungen für die im CML-Ausdruck verwendeten Operationen und das Prüfen der Existenz der im Ausdruck verwendeten Slots, Relationen usw. Das Zuweisen eines CML-Ausdrucks an ein Constraint findet mit Hilfe der Methode `setExpression()` statt. Der übergebene String wird geparst und gecheckt. Ist beides erfolgreich, gilt der Ausdruck als gesetzt.

Der Constraint-Ausdruck wird bei Aufruf der Methode `evaluateConstraint()` für die übergebene Instanz ausgewertet, dabei findet die *Interpretation* des Constraint-Ausdrucks statt. Beim Interpretieren wird abermals der Syntaxbaum verwendet, aber die symbolischen Ausdrücke durch konkrete Werte ersetzt und mit diesen die Operationen durchgeführt. Das Ergebnis der Interpretation eines CML-Ausdrucks ist ein boolescher Wert, der besagt, ob die betreffende Instanz das Constraint einhält oder nicht.

---

[1] Der Begriff *Wort* steht hier letztlich für einen Ausdruck.

8.5.2 Verwendete Werkzeuge

8.5.2.1 Parsergenerator JavaCC

JavaCC ist ein Werkzeug zur Generierung von Java-basierten Parsern. JavaCC steht für *Java Compiler Compiler*, wobei Compiler Compiler gleichbedeutend mit Parsergenerator ist. JavaCC war ursprünglich eine Entwicklung von Sun, wurde zwischenzeitlich von Metamata Inc. betreut und wird jetzt als Open-Source-Projekt mit BSD[1]-Lizenz unter dem Dach der java.net-Community gepflegt und weiterentwickelt. [JavaCC]

Bei der Suche nach einem geeigneten Generator für den CML-Parser fiel die Wahl auf JavaCC, weil er als der am weitesten verbreitete Parsergenerator für Java gilt, eine intuitiv verständliche Syntax für das Eingabefile verwendet und mit Grammatiken umgehen kann, für deren Parsing einen Lookahead-Wert größer als 1 nötig ist.

JavaCC generiert aus einer in EBNF definierten Grammatik einen Top-Down-Parser. Standardmäßig handelt es sich dabei um einen LL(1)-Parser. Wenn Teile der Grammatik nicht LL(1) sein sollten, gibt es die Möglichkeit, diese Teile oder die gesamte Grammatik explizit mit einem höheren Lookahead-Wert k zu markieren. Der generierte Parser ist dann entsprechend ein LL(k)-Parser. Von dieser Möglichkeit musste Gebrauch gemacht werden, da es sich, wie in Kapitel 7.15 beschrieben, bei der verwendeten CML-Grammatik um eine LL(2)-Grammatik handelt.

Das Eingabefile für den Parsergenerator beinhaltet die Definition der Tokens, die in EBNF formulierte Grammatik und den Action-Code. Der Action-Code wird in Java geschrieben und kann direkt in der Grammatik platziert werden. Im Code kann auf die vom Parser erkannten Tokens zugegriffen werden.

8.5.2.2 Präprozessor JJTree

Der Präprozessor JJTree ist Teil des JavaCC-Pakets, wird jedoch als eigenständiges Programm ausgeführt. Seine Aufgabe ist es, Action-Code so in die von JavaCC abzuarbeitende Grammatik einzufügen, dass der daraus erzeugte Parser aus dem Eingangsstring einen Syntaxbaum (engl. Abstract Syntax Tree, kurz AST) in Form von vernetzten Knotenobjekten erzeugt. (Abbildung 8.3)

[1] Berkeley Software Distribution License, siehe http://www.opensource.org/licenses/bsd-license.html

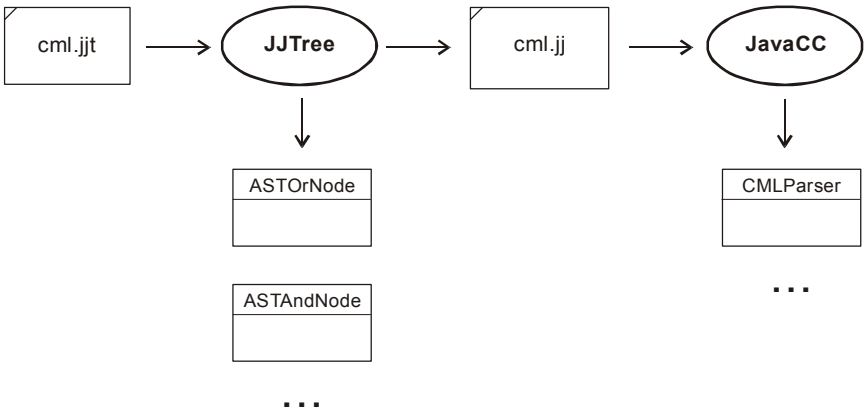

Abbildung 8.3: Ein- und Ausgabe von JJTree und JavaCC

Für die Generierung des Syntaxbaums sind zwei unterschiedliche Modi wählbar: Im einfachen Modus (simple mode) sind alle erzeugten Knoten Objekte der selben Klasse (SimpleNode). Im *multi mode* wird für jede Produktion eine spezielle Knotenklasse generiert und beim Parsen ein Objekt dieser Klasse erzeugt.

Standardmäßig generiert JJTree das Ausgabefile so, dass beim Parsen für jedes Nichtterminal der Grammatik ein Knoten im Syntaxbaum erzeugt wird. Dieses Verhalten kann vom Programmierer mit Hilfe von Präprozessoranweisungen so modifiziert werden, dass für bestimmte Nicht-Terminale keine Knoten erzeugt werden oder dass ein Knoten für einen spezifischen Teil einer Produktionsregel generiert wird.

Die Anweisungen stehen im Eingangsfile an den Stellen der Grammatik, an denen eine solche Modifikation vorgenommen werden soll. Sie werden von einem Rautezeichens eingeleitet, gefolgt vom Namen der zu erzeugenden Kontenklasse und einer optionalen Angabe zur Menge der Kindknoten oder vom Schlüsselwort `void`, wenn für eine Produktion keine Knotenklasse erzeugt werden soll.

JJTree erzeugt das Java-Interface `Node`, das alle generierten Knotenklassen implementieren. Das Interface stellt Methoden zum Setzen des Elternknotens und zum Hinzufügen von Kindknoten zur Verfügung.

Weiterhin kann der Nutzer im Actioncode-Bereich der zu verarbeitenden Eingabedatei den implizit definierten Bezeichner `jjtThis` verwenden, um beim Parsen den gerade erzeugten Knoten zu referenzieren und auf diese Weise seinen Attributen Werte zuzuweisen.

In folgendem Beispiel aus der CML-Grammatik wird dem Attribut `theString` der Klasse `StringLiteralNode` beim Parsen der Wert des aufgefundenen Tokens zugewiesen.

```
void literal() #void :
{
  Token t;
}
{
  (
    t = <STRING_LITERAL>
    {
      jjtThis.theString = t.image;
    }
```

```
  ) #StringLiteralNode
}
```

Der Syntaxbaum wird als Referenz mit dem Namen `jjtree` im Parser-Objekt gespeichert. Auf den Wurzelknoten des Syntaxbaums kann nach erfolgtem Parsing mit Hilfe der Methode `rootNode()` zugegriffen werden. Zum Beispiel:

```
CMLParser parser = new CmlParser(stream);
parser.expression();
Node rootNode = parser.jjtree.rootNode();
```

8.5.3 Implementation der Knotenklassen

Für die Implementation von CML-Checker und CML-Interpreter musste der Präprozessor JJTree und die von ihm erzeugten Knotenklassen verwendet werden. Der Versuch einer direkten Implementierung der erforderlichen Funktionalität im Actioncode-Bereich der Eingangsdatei für JavaCC scheiterte daran, dass der mit einem Token assoziierte Actioncode immer nur genau so viele Male ausgeführt wird, wie das jeweilige Token aufgefunden wird. Bei der Umsetzung von iterierenden Operationen ist jedoch ein mehrmaliges Ausführen notwendig, was nur durch expliziten Zugriff auf den Syntaxbaum realisiert werden kann.

Die Bereitstellung eines Syntaxbaums bei Verwendung von JJTree hat weitere Vorteile. Beim Zuweisen des CML-Ausdrucks an ein Constraint-Objekt findet ein einmaliges Parsing statt, danach wird der Syntaxbaum zusammen mit dem Constraint-Objekt gespeichert. Dadurch ist bei der häufig stattfindenden Interpretation dieses Ausdrucks kein erneutes Parsing notwendig. Weiterhin können geringfügige Änderungen am Verhalten des Interpreters direkt in der Implementation der Knotenklassen vorgenommen werden, eine erneute Generierung des Parsers ist nicht notwendig.

Die Eingangsdatei für JJTree ist im Anhang B2 zu finden. Im wesentlichen wurde hierin die CML-Grammatik in EBNF-Notation formuliert und um Präprozessoranweisungen zum Generieren von Knotenklassen angereichert.

8.5.3.1 Die Knotenklassen

Die erzeugten Klassen sind im einzelnen (in der Reihenfolge ihres Auftretens in der Grammatik):

```
ASTIfExpression
ASTOrNode
ASTAndNode
ASTXorNode
ASTImpliesNode
ASTIsEqualNode
ASTIsNotEqualNode
ASTIsBiggerNode
ASTIsBiggerOrEqualNode
ASTIsSmallerNode
ASTIsSmallerOrEqualNode
ASTAddNode
ASTSubstractNode
ASTMultiplicationNode
ASTDivisionNode
ASTUnaryMinusNode
ASTNotNode
ASTPostfixExpressionNode
ASTDeclaratorNode
ASTStringLiteralNode
ASTIntegerLiteralNode
ASTDoubleLiteralNode
```

Wie Abbildung 8.4 zeigt, werden alle Knotenklassen als Subklassen der Klasse `SimpleNode` erzeugt, welche wiederum das Interface `Node` implementiert. Das Interface `Node` wurde um die Deklaration der Methoden `check()` und `interpret()` erweitert, welche entsprechend von den Knotenklassen implementiert werden müssen. Diese beiden Methoden bilden die Grundlage der Interpreter- bzw. Checker-Funktionalität. Die Knotenklasse `SimpleNode` verwaltet darüber hinaus den Stack und die Symboltabelle und besitzt eine Referenz auf die aktuelle Instanz bzw. die aktuelle Klasse.

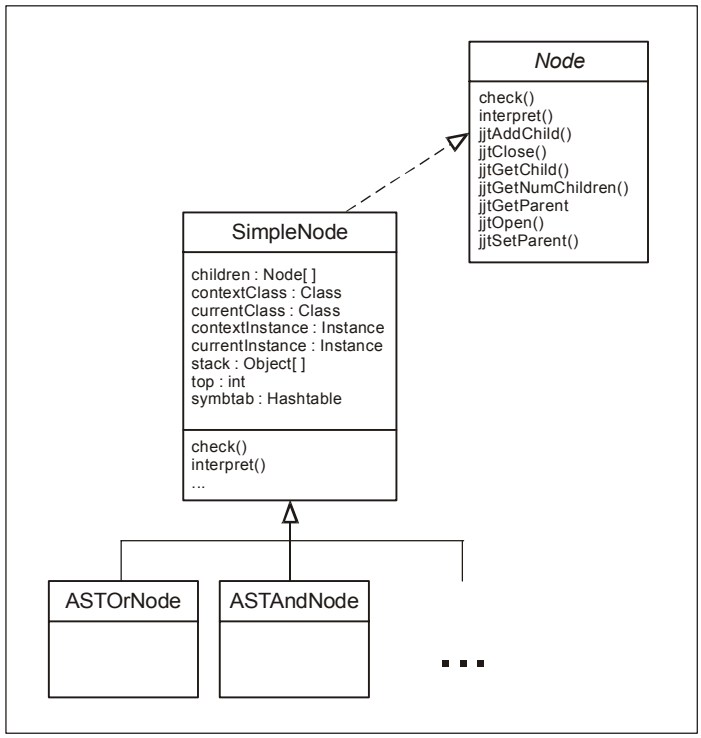

Abbildung 8.4: Klassendiagramm der Knotenklassen

Abbildung 8.5 zeigt beispielhaft den Syntaxbaum der beim Parsen des Ausdrucks „`hoehe+4=7 and 5*6>10`“ erzeugt wird. Die einzelnen Knoten des Syntaxbaums sind Instanzen der betreffenden Knotenklassen.

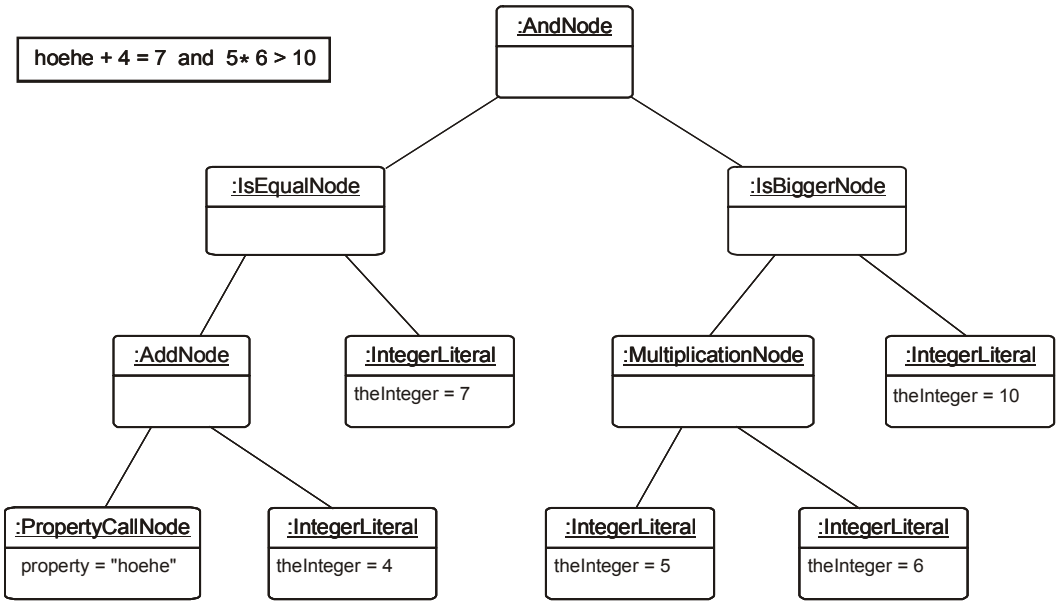

Abbildung 8.5: Syntaxbaum für den CML-Ausdruck „hoehe+4=7 and 5*6>10“

Der Aufruf der `check()`- bzw. `interpret()`-Methoden im Syntaxbaum geschieht von der Wurzel zu den Blättern, in der Abbildung also von oben nach unten. Dabei sind die Methoden jedoch häufig so implementiert, dass vor Ausführen der eigenen Operation die Ergebnisse der jeweiligen Kindknoten ermittelt werden.

8.5.3.2 CML-Checker

Beim Zuweisen eines CML-Ausdrucks an ein `Constraint`-Objekt wird dieser syntaktisch und semantisch analysiert, d.h. auf seine Korrektheit überprüft. Dabei wird die Existenz der im Ausdruck verwendeten Namen für Slots und Relationen geprüft. Weiterhin findet eine Typüberprüfung hinsichtlich der Verträglichkeit der in den Operationen involvierten Operanden statt.

Wird eine Unverträglichkeit festgestellt bzw. der Slot, die Facette oder die Relation mit dem im Ausdruck verwendeten Namen nicht aufgefunden, wird eine `ExpressionCheckException` mit detaillierten Angaben zur Art des Fehlers geworfen. Der Constraint-Ausdruck gilt damit als nicht gesetzt und der Nutzer kann zu einer Berichtigung aufgefordert werden.

Für die beim Checking stattfindenden Typprüfungen stellt die Klasse `GenericType` die statischen Methoden `isNumber()`, `multiplication()`, `addition()` und `division()` zur Verfügung.

Bei der Implementierung der `check()`-Methoden wurde auf eine Optimierung der Performanz Wert gelegt. Beispielsweise springt die `check()`-Methode des OR-Knotens bereits zurück, nachdem die Auswertung des ersten Operanden `true` ergeben hat.

Im Unterschied zum Interpreter arbeitet der Checker ausschließlich mit den Klassen des AKO-Metamodells, die das Domänenmodell beschreiben (`Class`, `Slot`, `Relation`), und nicht mit denen, die Zugriff auf die Ausprägungen einzelner Klassen erlauben (`Instance`, `SlotInstance`, `RelationInstance`). Beispiels-

weise wird bei der `check()`-Methode des PropertyCall-Knotens versucht, von der *aktuellen Klasse* auf den *Slot* mit dem angegebenen Namen zuzugreifen. Die `interpret()`-Methode des selben Knotens greift hingegen von der *aktuellen Instanz* aus auf die *Slotinstanz* mit diesem Namen zu.

8.5.3.3 CML-Interpreter

Der Interpreter wertet den CML-Ausdruck aus. Er arbeitet daher nicht wie der Checker mit Typen, sondern mit konkreten Werten. `GenericType`-Objekte kapseln einen Wert und den Typ dieses Wertes. Sie bilden innerhalb des Interpreters die Grundlage der Kalkulationen, in dem sie Methoden für arithmetische Operationen und Vergleichsoperationen anbieten.

Die für Zahlen und Strings definierten Operationen (siehe Kapitel 7.7) werden durch folgende Methoden der Klasse `GenericType` umgesetzt:

```
GenericType add ( GenericType value )
GenericType substract ( GenericType value )
GenericType divideBy ( GenericType value )
GenericType multiplyBy ( GenericType value )
GenericType isSmallerThan ( GenericType value )
GenericType isSmallerOrEqual ( GenericType value )
GenericType isBiggerThan ( GenericType value )
GenericType isBiggerOrEqual ( GenericType value )
GenericType unaryMinus()
GenericType not()
```

Rückgabewert jeder dieser Methoden ist ein neu erzeugtes `GenericType`-Objekt, das den Ergebniswert der Operation kapselt

Wird beim Interpretieren ein Fehler festgestellt, beispielsweise weil sich beispielsweise der Name eines Slots geändert hat und dadurch eine Slotinstanz mit dem im CML-Ausdruck angegebenen Namen nicht auffindbar ist, wird eine `InterpreterException` mit detaillierten Angaben zur Art und möglichen Ursache des Fehlers geworfen.

Das während der Interpretation notwendige Zwischenspeichern von Werten bzw. Objekten geschieht ebenso wie beim Interpreter mit Hilfe eines Stacks.

8.5.3.4 Stack

Während des Prüfens und während des Interpretierens eines CML-Ausdrucks kommt ein Stapelspeicher (engl. stack) zum Einsatz. Wesentlichste Eigenschaft eines Stapelspeichers ist, dass nur an oberster Stelle Elemente hinzugefügt bzw. entfernt werden können. Der Stack hat die Aufgabe, beim Prüfen bzw. Interpretieren Werte zwischenzuspeichern, die Ergebnis der Abarbeitung der `check()`- bzw. `interpret()`-Methode eines Knotens des Syntaxbaums sind und von einem anderen, weiter oben liegenden Knoten als Eingangswert benötigt werden. Er dient damit letztlich als Mittel der Kommunikation zwischen den Knotenobjekten.

Der Stack wird als statisches Attribut der Superklasse `SimpleNode` verwaltet, wodurch alle Knotenobjekte Zugriff auf ihn besitzen. Er besteht aus einem Feld von Objekten der Klasse `Object`, die die Superklasse aller Java-Klassen ist. Dadurch können Objekte jedweder Klasse auf dem Stack verwaltet werden. `SimpleNode` besitzt außerdem das statische Attribut `top` vom Typ Integer, das dazu dient, die

aktuelle Höhe des Stapels und damit die Position des höchsten Elements anzeigen. (vgl. Abbildung 8.6)

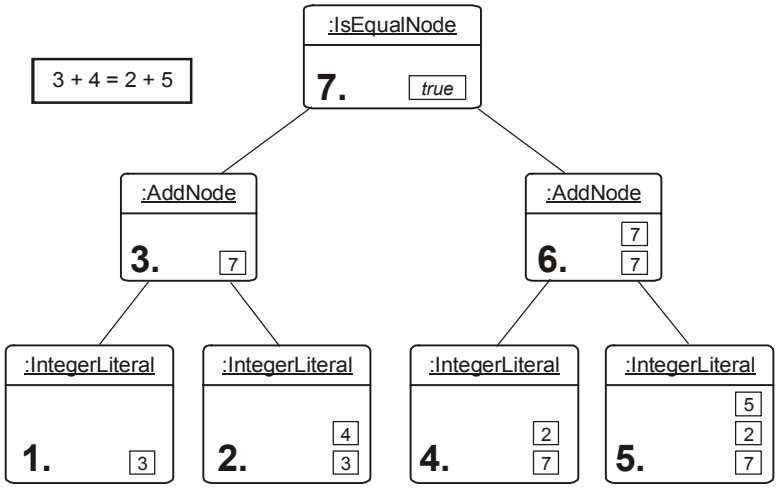

Abbildung 8.6: Reihenfolge der Abarbeitung von Knoten des Syntaxbaums und Stackbelegung

Zwar werden die `check()`- bzw. `interpret()`-Methoden der Knoten im Prinzip von der Wurzel aus hin zu den Blättern des Syntaxbaums aufgerufen, jedoch wird von der Implementation eines Knotens meist erst die jeweilige Methode der Kindknoten ausgeführt, bevor die eigene Operation umgesetzt wird. Der Grund hierfür liegt darin, dass im allgemeinen die Operanden einer Operation erst durch die Kindknoten zur Verfügung gestellt werden. Abbildung 8.6 zeigt für den Beispiel-Ausdruck „`3+4=2+5`“ die Reihenfolge der Abarbeitung der Knoten des Syntaxbaums beim Interpretieren und die Belegung des Stacks nach erfolgter Beendigung der `interpret()`-Methode eines Knotens.

Alle in der Abbildung dargestellten Zahlen sind genaugenommen `GenericType`-Objekte, die den Wert und den dazugehörigen Typ kapseln. Daneben werden von den verschiedenen Knoten des Syntaxbaums während des Interpretierens auch `Instance`-, `SlotInstance`- und `Collection`-Objekte auf den Stack gelegt. Da bei der Typüberprüfung nicht mit konkreten Werten gearbeitet wird, befinden sich während des Checkings nur Objekte vom Typ `ConceptualType`, `Class` oder `Slot` auf dem Stack.

Nach vollständiger Abarbeitung des Syntaxbaums liegt das Ergebnis der Prüfung bzw. des Interpretierens allein auf dem Stack. Nach dem Prüfen wird erwartet, dass es sich dabei um ein `ConceptualType`-Objekt mit der Belegung `BOOLEAN_TYPE` handelt. Ist dies nicht der Fall, wird eine `ExpressionCheckException` geworfen. Nach dem Interpretieren wird hingegen erwartet, dass sich auf dem Stack ein `GenericType`-Objekt vom selben Typ befindet. Wenn nicht, wird eine `InterpreterException` geworfen.

8.5.3.5 Literale

Literale werden durch Instanzen der Knotenklassen `ASTStringLiteralNode`, `ASTIntegerLiteralNode` bzw. `ASTDoubleLiteralNode` repräsentiert. Alle

drei Klassen besitzen ein Attribut (`theString`, `theInteger` bzw. `theDouble`) mit dem Wert des beim Parsen aufgefundenen Literals.

check()

Für die Implementation der `check()`-Methode wird dieser Wert nicht benötigt. Je nach Typ des Knotens legt die `check()`-Methode ein `ConceptualType`-Objekt mit dem Wert STRING_TYPE, DOUBLE_TYPE oder INTEGER_TYPE auf den Stack.

interpret()

Es wird ein `GenericType`-Objekt mit dem betreffenden Typ erzeugt, diesem der Wert des Attributs zugewiesen und zuoberst auf den Stack gelegt.

8.5.3.6 Eigenschaftszugriffe

Eigenschaftszugriffe (engl. property calls) werden durch Instanzen der Klasse `ASTpropertyCall` repräsentiert. Sie besitzt ein Attribut `property`, dem beim Parsen der aufgefundene Name der Eigenschaft zugewiesen wird.

Da die Grammatik von CML beim Parsing keine Unterscheidung zwischen Zugriffen auf Relationen, Slots bzw. Facetten und Aufrufen von Operationen gestattet, sondern alle derartigen Zugriffe als *property call* behandelt werden, gestalten sich sowohl die `check()`- als auch die `interpret()`-Methode des PropertyCall-Knotens recht komplex.

Erschwerend kommt hinzu, dass Zugriff auf Eigenschaften der Kontextinstanz ohne explizite Angabe des Schlüsselworts `self` geschehen können (siehe Kapitel 7.10). Zu beachten ist außerdem, dass sich implizite Zugriffe auf Eigenschaften im Parameterteil einer über Kollektionen iterierenden Operation nicht auf die Kontextklasse, sondern auf die Klasse der Kollektionselemente beziehen. (siehe Kapitel 5.2.13.3)

Die Art der Interpretation des in `property` gespeicherten Namens hängt maßgeblich vom Typ des zu oberst auf dem Stack liegenden Objektes ab. Anhand diesem wird entschieden, wie der übergebene Bezeichner zu interpretieren ist: als Slotname, als Rollenbezeichner einer Relationen oder als Name einer für Kollektionen definierten Operation. Auch das Quasi-Schlüsselwort `self` muss an dieser Stelle interpretiert werden. (vgl. Kapitel 7.15)

Die Prüfung und die Interpretation der für Kollektionen definierten Operationen `size()`, `isEmpty()`, `forAll()`, `select()` und `exists()` wurden ebenfalls in der Knotenklasse `ASTpropertyCall` implementiert. (vgl. Kapitel 7.11)

check()

In Abbildung 8.7 ist der Entscheidungsbaum zu sehen, der Grundlage der Implementation der `check()`-Methode des PropertyCall-Knotens ist. Die unterschiedlichen Reaktionen sind nicht abgebildet, sondern sollen im folgenden beschrieben werden.

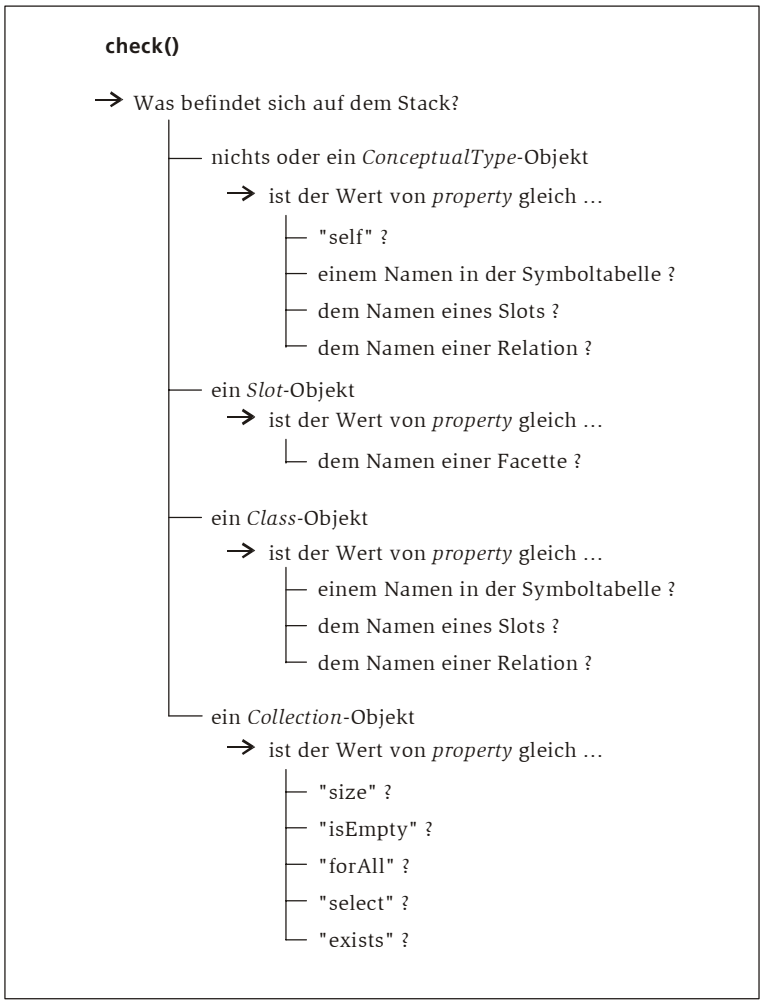

Abbildung 8.7: Entscheidungsbaum der check()-Methode des propertyCall-Knotens

Ist der Stack leer oder befindet sich ein `ConceptualType`-Objekt an oberster Stelle, so wird zunächst der Wert des Attributs `property` mit dem Schlüsselwort `self` verglichen. Ist der Vergleich positiv, wird die Kontextklasse an oberster Stelle auf den Stack gelegt. Wenn nicht, wird in der Symboltabelle nachgeschaut, ob ein Bezeichner mit dem `property`-Namen deklariert wurde. Wenn ja, wird das zugehörige `Class`-Objekt auf die oberste Stelle des Stacks gelegt. Wenn nicht, wird geprüft, ob es sich um den Namen eines Slots der *aktuellen Klasse* handelt. Wenn ja, wird das betreffende `Slot`-Objekt an oberster Stelle auf den Stack gelegt. Wenn nicht, wird geprüft, ob es sich um den Namen einer Relation der *aktuellen Klasse* handelt. Wenn ja, wird nach der Anzahl der über die Relation assoziierbaren Instanzen differenziert: Ist die Relation mit der Kardinalität `zero_or_one` oder

`exactly_one` versehen, wird das `Class`-Objekt, das die Zielklasse repräsentiert, auf den Stack gelegt. Bei anderen Kardinalitäten wird ein `Collection`-Objekt erzeugt und diesem das `Class`-Objekt der Zielklasse zugewiesen. Das `Collection`-Objekt wird an oberster Stelle auf den Stack gelegt.

Befindet sich ein `Slot`-Objekt an oberster Stelle des Stacks, wird versucht, auf eine Facette mit dem im `property`-Attribut gespeicherten Namen zuzugreifen. Gelingt das, wird ein `ConceptualType`-Objekt zuoberst auf den Stack gelegt, dem als Wert der Typ der Facette zugewiesen wird.

Ist ein `Class`-Objekt an oberster Stelle des Stacks, wird versucht, auf ein Slot bzw. eine Relation *dieser Klasse* mit dem in `property` gespeicherten Namen zuzugreifen. Bei Erfolg wird äquivalent zu oben ein `Slot`-, ein `Instance`- oder ein `Collection`-Objekt auf den Stack gelegt.

Befindet sich ein `Collection`-Objekt an oberster Stelle des Stacks, wird `property` mit den Namen der für Kollektionen definierten Operationen verglichen. Wird „`size`“ erkannt, wird ein `ConceptualType`-Objekt mit dem Wert `Long` erzeugt und zuoberst auf den Stack gelegt. Wird „`isEmpty`“ erkannt, wird ein `ConceptualType`-Objekt mit dem Wert `Boolean` erzeugt und zuoberst auf den Stack gelegt.

Die Prüfung von iterierenden Operationen ist in folgenden Punkten identisch: Vor der Prüfung wird die aktuelle Klasse zwischengespeichert, um sie nach Beendigung wieder zur aktuellen Klasse zu erklären. Wenn vorhanden, wird die `check()`-Methode des ersten Kindknotens (Deklarator-Knoten) ausgeführt, um ggf. die Klasse der Kollektion mit einem Bezeichner zu assoziieren. Danach wird die `check()`-Methode des zweiten Kindknotens ausgeführt. Hiernach liegt ein `ConceptualType`-Objekt auf dem Stack, dessen Wert `Boolean` sein muss. Im Fall von `exists` und `select` wird dieses Objekt zuoberst auf dem Stack belassen, im Fall von `select` ein `Collection`-Objekt dorthin geschoben.

interpret()

Abbildung 8.8 zeigt den Entscheidungsbaum, der der Implementation der interpret()-Methode des PropertyCall-Knotens zugrunde liegt. Er ähnelt stark dem Entscheidungsbaum der `check()`-Methode in Abbildung 8.7, Unterschiede bestehen jedoch in den erwarteten Typen des auf dem Stack liegenden Objekts.

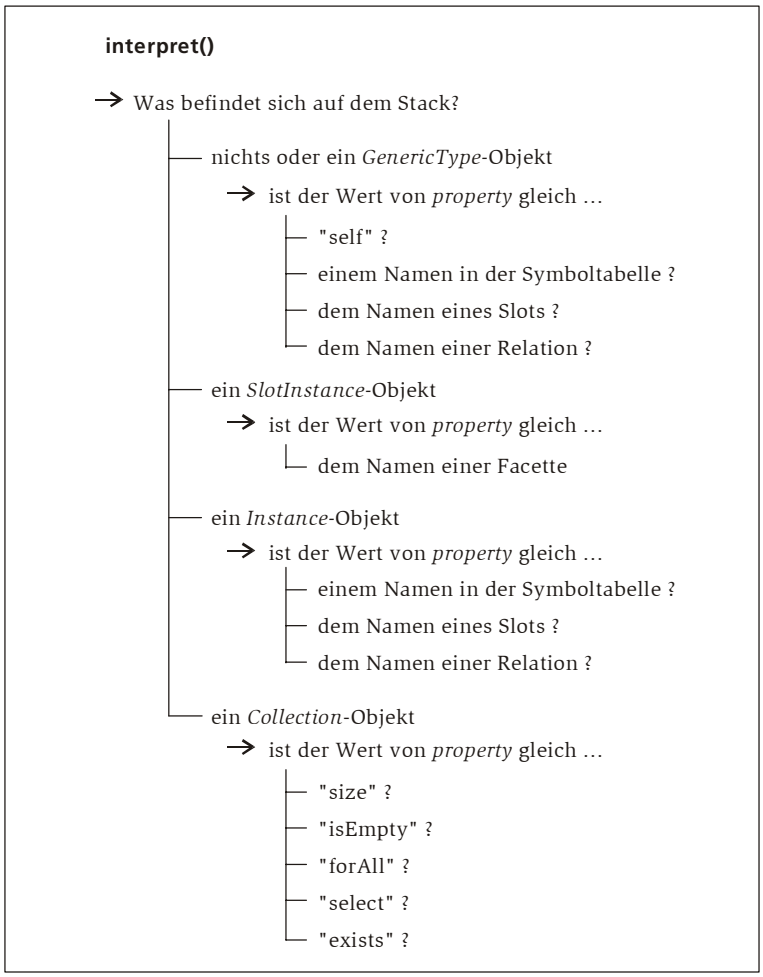

Abbildung 8.8: Entscheidungsbaum der interpret()-Methode des propertyCall-Knotens

Ist der Stack leer oder befindet sich ein `GenericType`-Objekt an oberster Stelle, so wird zunächst der Wert des Attributs `property` mit dem Schlüsselwort `self` verglichen. Ist der Vergleich positiv, wird die Kontextinstanz an oberster Stelle auf den Stack gelegt. Wenn nicht, wird in der Symboltabelle nachgeschaut, ob ein Bezeichner mit dem `property`-Namen deklariert wurde. Wenn ja, wird das zugehörige `Instance`-Objekt auf die oberste Stelle des Stacks gelegt. Wenn nicht, wird geprüft, ob es sich um den Namen eines Slots an der *aktuellen Instanz* handelt. Wenn ja, wird die betreffende Slotinstanz an oberster Stelle auf den Stack gelegt. Wenn nicht, wird geprüft, ob es sich um den Namen einer Relation an der aktuellen Instanz handelt. Wenn ja, wird nach der Anzahl der über die Relation assoziierbaren Instanzen differenziert: Ist die Relation mit der Kardinalität `zero_or_one` oder `exactly_one` versehen, wird die assoziierte Instanz auf den

Stack gelegt. Bei anderen Kardinalitäten wird ein `Collection`-Objekt erzeugt, das alle assoziierten Instanzen aufnimmt, und an oberster Stelle auf den Stack gelegt.

Befindet sich eine Slotinstanz an oberster Stelle des Stacks wird versucht, auf eine Facette mit dem im `property`-Attribut gespeicherten Namen zuzugreifen. Gelingt das, wird das `GenericType`-Objekt, das die Facetteinstanz repräsentiert, zuoberst auf den Stack gelegt.

Ist ein `Instance`-Objekt an oberster Stelle des Stacks, wird versucht, auf eine Slot- bzw. Relationeninstanz mit dem in `property` gespeicherten Namen an *dieser Instanz* zuzugreifen. Bei Erfolg wird äquivalent zu oben beschriebenen Verfahren eine Slotinstanz, eine Instanz oder ein `Collection`-Objekt auf den Stack gelegt.

Befindet sich ein `Collection`-Objekt an oberster Stelle des Stacks, wird `property` mit den Namen der für Kollektionen definierten Operationen verglichen und bei Erfolg die entsprechende Operation ausgeführt.

`size()`: Die Anzahl der in der Kollektion befindlichen Elemente wird bestimmt, ein `GenericType`-Objekt vom Typ `Long` erzeugt und diesem der Wert zugewiesen.

`isEmpty()`: Die Anzahl der in der Kollektion befindlichen Elemente wird bestimmt. Ein `GenericType`-Objekt vom Typ `Boolean` wird erzeugt und diesem der Wert `true` bzw. `false` zugewiesen.

Die Implementation von iterierenden Operationen ist in folgenden Punkten identisch: Vor dem Ausführen einer iterierenden Operationen wird die aktuelle Instanz zwischengespeichert, um sie nach Beendigung wieder zur aktuellen Instanz zu erklären. Es wird über alle in der Kollektion gespeicherten Instanzen iteriert. Dabei wird die jeweils aktuelle Instanz mit `currentInstance` assoziiert. Wenn vorhanden, wird der erste Kindknoten (Deklarator-Knoten) interpretiert, um ggf. die aktuelle Instanz der Kollektion mit einem Bezeichner zu assoziieren. Danach wird der zweite Kindknoten interpretiert, der den für die Instanz zu prüfenden Ausdruck repräsentiert. Hiernach liegt ein `GenericType`-Objekt vom Typ `Boolean` auf dem Stack, das besagt, ob der Ausdruck für die aktuelle Instanz `wahr` oder `falsch` ist.

`forAll()`: Wenn die Auswertung des Ausdrucks für eine der Instanzen `false` ergibt, ist das Ergebnis der `forAll()`-Operation `false`, sonst `true`. Es wird ein `GenericType`-Objekt vom Typ `Boolean` erzeugt, mit dem entsprechenden Wert belegt und an Stelle der Kollektion auf den Stack gelegt.

`select()`: Es wird eine `Collection`-Objekt erzeugt, dass die ausgewählten Instanzen aufnimmt. Alle Instanzen, für die die Auswertung des Ausdrucks `true` ergibt, werden dieser Kollektion hinzugefügt. Diese wird am Ende auf den Stack gelegt.

`exists()`: Wenn die Auswertung des Ausdrucks für eine der Instanzen `true` ergibt, ist das Ergebnis der `exists()`-Operation `true`, sonst `false`. Es wird ein `GenericType`-Objekt vom Typ `Boolean` erzeugt, mit dem entsprechenden Wert belegt und an Stelle der Kollektion auf den Stack gelegt.

8.5.3.7 Deklarationen

Wenn im CML-Ausdruck Iterator-Variablen deklariert werden (siehe Kapitel 7.11), wird diese Deklaration durch eine Instanz der Knotenklasse `ASTdeclarator` repräsentiert. Sie besitzt ein Attribut `variableName` vom Typ String, dem beim Parsing der Name des deklarierten Bezeichners zugewiesen wird.

check()

Die `check()`-Methode trägt den deklarierten Bezeichner in die Symboltabelle ein und weist ihm die aktuelle Klasse zu.

interpret()

Die `interpret()`-Methode trägt den deklarierten Bezeichner in die Symboltabelle ein und weist ihm die aktuelle Instanz zu.

8.5.3.8 Postfix-Ausdrücke

Wie in Kapitel 7.10 beschrieben, kann in CML ist die Angabe der Facette eines Slots optional. Wird kein Facettenname angegeben, wird implizit auf die Value-Facette zugegriffen. Die Knotenklasse `ASTpostfixExpression` übernimmt die Aufgabe, für Flexibilität hinsichtlich der Angabe einer Facette zu sorgen.

check()

Befindet sich nach Abarbeitung des Subknotens ein Slot an oberster Stelle des Stacks, wird der Typ seiner Value-Facette ermittelt und ein `ConceptualType`-Objekt mit dem entsprechenden Wert an Stelle des Slots auf den Stack gelegt. Im anderen Fall wird der Stack nicht manipuliert.

interpret()

Befindet sich nach Abarbeitung des Subknotens eine Slotinstanz auf dem Stack, wird der Wert ihrer Value-Facette ermittelt und das `GenericType`-Objekt an Stelle der Slotinstanz auf den Stack gelegt. Im anderen Fall wird der Stack nicht manipuliert.

8.5.3.9 Unäre Operationen

Zu den unären Operationen gehören das für Zahlen definierte unäre Minus und das auf boolesche Werte anwendbare `Not`. Die beiden unären Operationen werden durch Instanzen der Klassen `ASTUnaryMinusNode` bzw. `ASTNotNode` repräsentiert. Die Knotenobjekte besitzen jeweils nur einen Kindknoten.

check()

Die von `check()`-Methode von `ASTUnaryMinus` prüft, ob es sich beim Operanden um eine Zahl handelt, die von `ASTNotNode` ob ein Boolean-Wert zuoberst auf dem Stack liegt.

interpret()

Für die Implementation der `interpret()`-Methoden wird auf die Methoden `unaryMinus()` bzw. `unaryNot()` der Klasse `GenericType` zurückgegriffen. Die Methode `unaryMinus()` gibt den negativen Wert einer Zahl und die Methode `unaryNot()` den invertierten Wert eines Booleans zurück.

8.5.3.10 Arithmetische Operationen

Das Auftreten der arithmetischen Operationen * , / , + und – in CML-Ausdrücken wird durch Instanzen der Knotenklassen `ASTAddNode`, `ASTSubtractNode`, `ASTMultiplicationNode` und `ASTDivisionNode` repräsentiert. Arithmetische Operationen sind nur für Zahlen definiert.

check()

Für die Typüberprüfung der Operanden von arithmetischen Operationen werden die statischen Methoden `multiplication()`, `addition()` und `division()` der Klasse `GenericType` genutzt. Nach der Abarbeitung der beiden Kindknoten liegen die Typen der beiden Operanden als `ConceptualType`-Objekte zuoberst auf dem Stack. Diese werden der jeweiligen Methode übergeben, die gemäß Tabelle 7.2, Tabelle 7.3 bzw. Tabelle 7.4 den Ergebnistyp der zugehörigen Operation zurückliefert. Für die Subtraktion wurde keine gesonderte Hilfsmethode implementiert, da für sie die gleichen Regeln gelten wie für die Addition.

interpret()

Zunächst werden die Operanden der Operation durch Ausführen der `interpret()`-Methode der beiden Kindknoten bestimmt. Sie liegen als `GenericType`-Objekte zuoberst auf dem Stack. Dann wird zum Durchführen der Operation die Methoden `add()`, `substract()`, `divideBy()` bzw. `multiplyBy()` am `GenericType`-Objekt ausgeführt, das den ersten Operanden repräsentiert. Der jeweiligen Methode wird das `GenericType`-Objekt des zweiten Operanden übergeben. Zurückgegeben wird ein `GenericType`-Objekt, dass den Ergebniswert und dessen Typ kapselt. Dieses wird anstelle des ersten Operanden auf den Stack gelegt und die Höhe des Stacks um 1 verringert.

8.5.3.11 Vergleichsoperationen

Das Auftreten der Vergleichsoperationen > , < , = , < = , > = und < > in CML-Ausdrücken wird durch Instanzen der Knotenklassen

```
ASTIsEqualNode,
ASTIsBiggerNode,
ASTIsSmallerNode,
ASTIsBiggerOrEqualNode,
ASTIsSmallerOrEqualNode bzw.
ASTIsNotEqualNode
```

repräsentiert. Wie aus Kapitel 7.7 hervorgeht, sind die Vergleichsoperationen = und < > sowohl für Zahlen als auch für Instanzen definiert. Alle anderen Vergleichsoperationen gelten nur für Zahlen.

check()

Nach Abarbeitung der `check()`-Methode der beiden Kindknoten liegen die Typen der beiden Operanden als `ConceptualType`- oder `Class`-Objekte zuoberst auf dem Stack. Mit Hilfe der statischen Methode `isNumber()` der Klasse `GenericType` wird geprüft, ob es sich beim Typ der Operanden um einen Zahlentyp handelt. Ist dies nicht der Fall, wird in den Klassen `ASTIsEqualNode` und `ASTIsNotEqualNode` zusätzlich geprüft, ob die beiden zuoberst auf dem Stack liegenden Objekte Instanzen von `Class` sind.

Ist die Prüfung erfolgreich, wird ein `ConceptualType`-Objekt mit der Belegung `BOOLEAN_TYPE` auf den Stack gelegt und die `check()`–Methode springt zurück. Wenn nicht, wird eine `ExpressionCheckException` geworfen.

interpret()

Nach Abarbeitung der `interpret()`-Methode der beiden Kindknoten liegen die beiden Operanden als `GenericType`- bzw. `Instance`-Objekte zuoberst auf dem Stack. Im ersten Fall wird der Vergleich durch Aufruf der Methoden `isEqualTo()`, `isSmallerThan()`, `isSmallerOrEqual()`, `isBiggerThan()` bzw. `isBiggerOrEqual()` am ersten Operanden durchgeführt und der zweite Operand als Parameter übergeben. Diese Methoden liefern ein `GenericType`-Objekt vom Typ `Boolean` zurück, das anstelle des ersten Operanden auf den Stack gelegt wird. Außerdem wird die Höhe des Stacks um 1 verringert.

Beim Vergleich von `Instance`-Objekten wird deren ID verglichen und das Ergebnis als `GenericType`-Objekt vom Typ `Boolean` auf den Stack gelegt.

8.5.3.12 Operationen zur logischen Verknüpfung

Die Operationen `and`, `or`, `xor` und `implies` dienen zur logischen Verknüpfung von booleschen Werten. Ihr Auftreten im CML-Ausdruck wird durch Instanzen der Knotenklassen `ASTOrNode`, `ASTAndNode`, `ASTXorNde`, `ASTImpliesNode` repräsentiert.

check()

Die Operanden und das Ergebnis einer logischen Verknüpfung sind immer vom Typ Boolean. Entsprechend wird nach Abarbeitung der `check()`-Methode der beiden Kindknoten geprüft, ob die beiden zuoberst auf dem Stack liegenden Objekte `ConceptualType`-Objekte mit der Belegung `BOOLEAN_TYPE` sind. Ist dies der Fall, wird die Höhe des Stacks um 1 verringert, wenn nicht, wird eine `ExpressionCheckException` geworfen.

interpret()

`ASTAndNode`: Wenn der Wert des ersten Operanden `false` ist, muss der zweite Operand nicht ermittelt, d.h. der zweite Kindknoten nicht abgearbeitet werden. Es wird ein `GenericType`-Objekt mit der Belegung `false` auf den Stack gelegt. Sonst werden erster und zweiter Operand durch den Java-Operator „`&&`“ verknüpft und das Ergebnis auf den Stack gelegt.

`ASTOrNode`: Wenn der Wert des ersten Operanden `true` ist, muss der zweite Operand nicht ermittelt, d.h. der zweite Kindknoten nicht abgearbeitet werden. Es wird ein `GenericType`-Objekt mit der Belegung `true` auf den Stack gelegt. Sonst werden erster und zweiter Operand durch den Java-Operator „`||`“ verknüpft und das Ergebnis auf den Stack gelegt.

`ASTXorNode`: Für die Bestimmung des Ergebnisses einer XOR-Verknüpfung müssen beide Operatoren ermittelt, d.h. beide Kindknoten abgearbeitet werden. Erster und zweiter Operand werden durch den Java-Operator „`^`“ verknüpft. Es wird ein `GenericType`-Objekt mit dem Ergebnis auf den Stack gelegt.

`ASTImpliesNode`: Wenn der Wert des ersten Operanden `false` ist, muss der zweite Operand nicht ermittelt, d.h. der zweite Kindknoten nicht abgearbeitet werden. Es wird ein `GenericType`-Objekt mit der Belegung `true` auf den Stack gelegt. Sonst werden erster und zweiter Operand durch den Java-Operator „`&&`“ verknüpft und das Ergebnis auf den Stack gelegt.

8.5.3.13 Konditionale Ausdrücke

Ein Bedingungsausdruck wird mit Hilfe der Schlüsselwörter `if`, `then`, `else` und `endif` formuliert. Sein Auftreten im CML-Ausdruck wird durch eine Instanz der Knotenklasse `ASTifExpression` repräsentiert, die drei Kindknoten besitzt: der erste repräsentiert den Bedingungsausdruck nach dem Schlüsselwort `if`, der zweite den Ausdruck des `then`-Zweigs und der dritte den des `else`-Zweigs.

check()

Die `check()`-Methode prüft, ob der Bedingungsausdruck einen `Boolean`-Wert zurückliefert, und ob der Rückgabetyp von `then`- und `else`-Ausdruck gleich ist. Ist eine dieser Bedingungen nicht erfüllt, wird eine Exception geworfen.

interpret()

Zunächst wird die `interpret()`-Methode des ersten Kindknotens aufgerufen. Danach befindet sich ein `GenericType`-Objekt vom Typ `Boolean` an oberster Stelle des Stacks. Ist es mit dem Wert `true` belegt, wird die `interpret()`-Methode des zweiten Kindknotens aufgerufen, andernfalls die des dritten Kindknotens.

8.6 Erweiterungen am MVS-Browser[1]

Der MVS-Browser dient zur Anzeige und Modifikation der durch den MVS-Server verwalteten Domänenmodelle und der zugehörigen Instanzdaten. Er wurde um die Möglichkeit zur Definition von Constraints durch Formulieren von CML-Ausdrücken und zum Auslösen von Constraint-Prüfungen für einzelne Instanzen erweitert. Beide Funktionalitäten wurden durch interaktive Dialogboxen realisiert, die durch einen Klick mit der rechten Maustaste auf die betreffende Klasse bzw. Instanz aktiviert werden können.

Die Ergebnisse einer Constraint-Prüfung werden durch eine Tabelle visualisiert, in deren linker Spalte die Namen der geprüften Constraints aufgeführt sind. In der rechten Spalte steht entweder ein „ok“, wenn das Constraint eingehalten ist, oder ein „X“, wenn eine Verletzung des Constraints festgestellt wurde. (Abbildung 8.9)

---

[1] Die Erweiterung des MVS-Browsers war nicht Teil der Aufgabenstellung und dient lediglich der Präsentation der auf Serverseite implementierten Funktionalität. Die Erweiterungen am MVS-Browser wurden durch Katrin Wender vorgenommen.

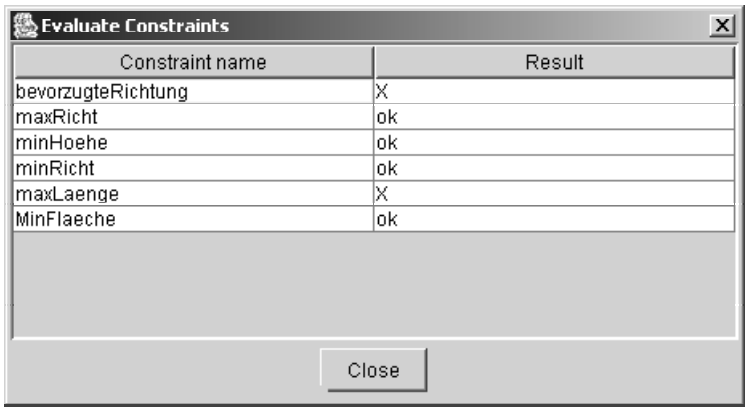

Abbildung 8.9: Dialogbox, die die Ergebnisse der Constraint-Prüfungen für eine Instanz zeigt

8.7 Zusammenfassung

Dieses Kapitel behandelte die erfolgreiche Umsetzung eines Constraint-Moduls für das im SFB 524 im Einsatz befindliche Modellverwaltungssystem. Hierfür wurde zunächst auf konzeptioneller Ebene die notwendige Erweiterung des AKO-Metamodells um die Klasse `Constraint` erörtert. Um den Clients des Modellverwaltungssystems die Funktionalität des Constraint-Moduls zur Verfügung zu stellen, mussten Erweiterungen an der AKO-Schnittstelle vorgenommen werden, die detailliert beschrieben wurden.

Der aufwendigste Teil bei der Implementierung des Constraint-Moduls lag in der Umsetzung des CML-Checkers und des CML-Interpreters. Bei der dafür notwendigen Entwicklung eines Parsers für CML-Ausdrücke wurde der Parsergenerator JavaCC und dessen Präprozessor JJTree eingesetzt. JJTree gestattet den expliziten Zugriffs auf den beim Parsen erzeugten Syntaxbaum. Die von JJTree erzeugten Knotenklassen wurden jeweils um die Methoden `check()` und `interpret()` erweitert, deren Implementation die Grundlage der Checker- bzw. Interpreter-Funktionalität bildet.

9 Zusammenfassung und Ausblick

9.1 Zusammenfassung

Es ist gelungen, ein tragfähiges Konzept für die Integration von Zwangsbedingungen (Constraints) in dynamische Bauwerksmodelle zu entwickeln. Constraints können dazu eingesetzt werden, Domänenwissen und projektspezifisches Wissen im Informationssystem zu repräsentieren. Während Constraints der ersten Art mit Klassen des Domänenmodells assoziiert werden, werden projektspezifische Constraints im allgemeinen mit einzelnen Instanzen verknüpft.

Constraints können auf unterschiedlichste Art und Weise eingesetzt werden: Zum Festlegen des Wertebereichs eines Attributs, zum Beschreiben von Abhängigkeiten zwischen Attributen, zum Festlegen des Typs der über eine Relation assoziierbaren Instanzen oder zur näheren Beschreibung der Semantik von Assoziationsklassen. Das in dieser Arbeit verfolgte Konzept sieht vor, dass temporäre Unstimmigkeiten zwischen den Instanzdaten und den für sie geltenden Bedingungen vom Modellverwaltungssystem toleriert werden. Damit die bei der Planung notwendige Flexibilität gewährleistet werden kann, findet die Prüfung von Instanzen zu einem vom Nutzer festzulegenden Zeitpunkt statt.

Für die Formulierung von Zwangsbedingungen muss eine formale Sprache zum Einsatz kommen. Bei der Untersuchung der *Object Constraint Language* (OCL) auf Eignung für AKO-basierte Modellverwaltungssysteme musste festgestellt werden, dass die zwischen dem UML- und dem AKO-Metamodell bestehenden Differenzen dazu führen, dass OCL nicht direkt eingesetzt werden kann. Mit der *Constraint Modeling Language* (CML) konnte ein OCL-Dialekt entwickelt werden, der die intuitive Verwendbarkeit und einen großen Teil der Funktionalität von seinem Vorbild OCL erbt und mit dem AKO-Metamodell kompatibel ist.

Der Sprachumfang von CML ist zwar gegenüber dem von OCL leicht verkleinert, bietet aber dennoch umfangreiche Möglichkeiten zur Formulierung von komplexen Bedingungsausdrücken. Dazu gehören der Zugriff auf facettierte Attribute, die Navigation entlang von Assoziationsbeziehungen, Verknüpfungen durch arithmetische und logische Operatoren, Vergleichsoperationen sowie konditionale Konstrukte. Zu einer hohen Ausdruckmächtigkeit tragen vor allem auch die für Kollektionen definierten Operationen der Mengenalgebra bei.

Die Tragfähigkeit des entwickelten Konzepts konnte durch die prototypische Umsetzung eines Constraint-Moduls für das im SFB 524 eingesetzte Modellverwaltungssystem nachgewiesen werden.

9.2 Ausblick

Der Philosophie eines dynamischen Modellverwaltungssystems gemäss sollten bei laufzeitdynamischen Modifikationen am verwalteten Domänenmodell die gespeicherten Constraint-Ausdrücke automatisch vom System aktualisiert werden, um dem Modellierer lästige Änderungen an den Namen von Klassen, Relationen und Slots bzw. Facetten zu ersparen. Um das umzusetzen, müssen im Knotenobjekt `propertyCall` des beim Parsen erzeugten Syntaxbaums statt der Namen von Relationen, Slots bzw. Facetten, Referenzen auf die jeweiligen Metaobjekte gespeichert werden. Der CML-Ausdruck selbst wird dann nicht mehr vom Constraint-Objekt verwaltet, sondern kann bei Bedarf aus dem Syntaxbaum abgeleitet werden. Wird der Name eines Slots oder die Rolle einer Relation geändert, behält das Constraint seine Semantik; wird der Constraint-Ausdruck angefordert, beinhaltet er die aktualisierten Namen.

Wird beim Prüfen der Constraints festgestellt, dass ein oder mehrere Bedingungen nicht erfüllt sind, muss die Behebung derzeit vollständig manuell erfolgen. Vor allem dann, wenn die Constraints komplex vernetzt sind, d.h. wenn sich mehrere Constraints auf die gleichen Facetten beziehen, kann sich die Lösung dieser Aufgabe als äußerst schwierig gestalten. In diesem Fall sollte die Entwurfsumgebung dem Anwender assistieren. Zur Umsetzung dieser Funktionalität können die im Bereich der Künstlichen Intelligenz entwickelten Techniken des Constraint-basierten Lösens zum Einsatz kommen (vgl. Kapitel 3.2.1). Die Syntaxbäume der Constraints bilden eine gute Grundlage für die Anwendung von Propagierungsalgorithmen. Vorrausetzung für das Erkennen von Abhängigkeiten zwischen Constraints ist jedoch die oben beschriebene Verwaltung von Referenzen auf Facetteninstanzen. Auf diese Weise kann auch geprüft werden, ob die Constraints untereinander verträglich sind, d.h. ob sie evtl. in Konflikt miteinander stehen und es gar keine Lösung des Constraint-Problems gibt.

Weiterhin muss die Anwenderfreundlichkeit beim Festlegen von Constraints erhöht werden. Hierbei soll vor allem auf die Signifikanz räumlicher Operatoren (*„liegt in“*, *„darüber“*, etc.) für Constraints in Bauwerksmodellen verwiesen werden. Derzeit müssen räumliche Typen und deren Relationen als Klassen im Domänenmodell definiert (vgl. Kap. 4.4.5) werden. Die Formulierung von topologischen Zwangsbedingungen ist mit Hilfe von Constraints für diese Klassen grundsätzlich möglich, die dafür zu verwendenden CML-Ausdrücke sind jedoch sehr komplex. Eine andere Möglichkeit besteht darin, den AKO-Typ `Point3d` in den Sprachumfang von CML aufzunehmen und für ihn gültige räumliche Operationen zu definieren. Außerdem ist eine Ergänzung des AKO-Typsystems um weitere räumliche Datentypen (wie z.B. *3D-Bereich*) mit entsprechenden Operationen denkbar.

Das AKO-Metamodell sieht bislang keine Möglichkeit zur Abbildung des Verhaltens von Objekten vor. Zur Implementierung einfacher Operationen ohne aufwendige Kontrollstrukturen bietet sich die Verwendung von Pseudo-Attributen an. Zur Formulierung der Berechnungsformel kann die Syntax von CML verwendet werden, so wie in den *definition*-Constraints der OCL (siehe Kapitel 5.2.5). Bei der Modellierung des Domänenmodells kann zur Visualisierung von Constraints im MVS-Browser eine graphische Notation von CML verwendet werden, ähnlich der in [Kiesner02] für OCL vorgeschlagenen. In den Fachapplikation müssen die für einzelne Instanzen geltenden Constraints auf eine der jeweiligen Entwurfsumgebung angepasste Art angezeigt werden. In räumlichen bzw. geometrischen Pla-

nungswerkzeugen können entsprechende Constraints beispielsweise durch Markieren des für ein Bauteil gültigen Bereichs visualisiert werden.

Schließlich kann CML auch zum Formulieren von Anfragen an das Modellverwaltungssystem genutzt werden. Hierzu muss CML allerdings um Operationen zur Bildung von Schnitt- und Vereinigungsmengen, wie sie in OCL definiert sind, erweitert werden. Das Ergebnis einer Anfrage muss anders als bei Constraint-Ausdrücken nicht notwendigerweise ein Boolean-Wert sein, sondern kann ebenso ein Zahlenwert oder eine spezifizierte Teilmenge der verfügbaren Instanzen sein.

Anhang A Literaturverzeichnis

[Aho88] Aho, A. V.; Sethi R.; Ullmann D. U.: Compilerbau, Addison-Wesley, 1988

[AKO97] Ranglack, D.; Kolbe, P.; Steinmann, F.: Eine Schnittstelle für dynamische Objektstrukturen für Entwurfsanwendungen, IKM, Weimar 1997

[Booch93] Booch, G. Object-Oriented Analysis and Design with Application, Addison Wesley, 1993

[Booch99] Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language User Guide, Addison Wesley, 1999

[Borrmann02] Borrmann, A.: Entwurf und prototypische Umsetzung eines Benachrichtigungsmoduls für das Modellverwaltungssystem des SFB 524. Studienarbeit. Bauhaus-Universität Weimar, 2002

[Cook94] Cook, St.; Daniels, J.: Designing Object Systems: Object-Oriented Modelling with Syntropy , Prentice Hall, 1994

[Firmenich02] Firmenich, B.: CAD im Bauplanungsprozess: Verteilte Bearbeitung einer strukturierten Menge von Objektversionen. Shaker Verlag, 2002. Zugl.: Dissertation, Bauhaus-Universität Weimar, 2001

[Frühwirth97] Frühwirth, Th., Abdennadher, S.: Constraint-Programmierung. Grundlagen und Anwendungen. Springer-Verlag, 1997

[Graham95] Graham, I.: Migration to Object Technology, Addison-Wesley, 1995

[Hauschild00] Hauschild, Th.; Hübler,R.; Schleinitz, M.: Unterstützung von Gruppenarbeit im Bauwerksentwurf auf Basis eines dynamischen objektorientierten Bauwerks-Modelliersystems, CAD 2000, Berlin 2000

[Hauschild02] Hauschild, Th., Hübler, R.: Distributed, Collaborative Management of Building Models for Revification Projects, IX. International Conference on Computing in Civil and Building Engineering (ICCCBE), Taipeh, 2002

[Hannus98] Hannus, M.: The Islands of Automation. http://cic.vtt.fi/projects/ratas/islands.html

[Hovestadt94] Hovestadt, L.: A4 – Digitales Bauen – Ein Modell für die weitgehende Computerunterstützung von Entwurf, Konstruktion und Betrieb von Gebäuden. VDI-Fortschrittsberichte Reihe 20 Nr. 120, VDI-Verlag, 1994. Zugl.: Dissertation, Universtät Karlsruhe, 1994

[Hübler02] Hübler, R.; Hauschild, Th.; Willenbacher, H.: Distributed Cooperative Building Models for Revivification of Buildings, International Association for Bridge and Structural Engineering (IABSE) Symposium 2002, Melbourne 2002

[IFC] http://eetd.lbl.gov/btd/iai/index.html

[Jaffar87] Jaffar, J.; Lassez, J. L.: Constraint Logic Programming. In: Proceedings of the 14th ACM Symposium on Principles of Programming Languages POPL-87, München, 1987

[JavaCC] https://javacc.dev.java.net/

[Kiesner02] Kiesner, Ch.; Taentzer, G.; Winkelmann, J.: Visual OCL : Eine visuelle Notation der Object Constraint Language. Technischer Bericht. Technische Universität Berlin, Fakultät Elektrotechnik und Informatik, 2002

[Kolbe98] Kolbe, P.: AKO – Arbeitskreis Objekte, Eine Schnittstelle für Modellverwaltungssysteme, Forum Bauinformatik 1998
A. Grosche, U. Schneider, R. Schumann (Hrsg.), VDI-Verlag, 1998

[Kuper00] Kuper, G.; Libkin, L.; Paredaens, J. (Hrsg.): Constraint Databases, Springer-Verlag, 2000

[Schulze96] Schulze, H. H.: PC-Lexikon – Fachbegriffe schlüssig erklärt. rororo Taschenbuch, 1996

[D'Souza99] D'Souza, D. F.; Wills, A. C.: Objects, Components and Frameworks with UML: The Catalysis Approach. Addision Wesley Logman, 1999

[Stallman77] Stallman, R.; Sussman, G: Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis. AI-Journal 9, 1977

[Steinmann97] Steinmann, F.: Modellbildung und computergestütztes Modellieren in frühen Phasen des architektonischen Entwurfs. Dissertation, Bauhaus-Universität Weimar, 1997

[Sturm97] Sturm, R.: Dynamische Regelmengen zur Beschreibung von Entwurfsspielräumen. VDI-Fortschrittsberichte Reihe 10 Nr. 495, VDI-Verlag, 1997.
Zugl.: Dissertation, Universtät Karlsruhe, 1994.

[Sussman80] Sussman, G; Steele, G.: Constraints – a Language for Expressing Almost-Hierarchical Descriptions. AI-Journal 14, 1980

[Spivey88] Spivey, J. M.: Understanding Z. Cambridge U Press, 1988

[Meyer88] Meyer, B.: Object-Oriented Software Construction. Prentice Hall, 1988

[OMG97] The Object Management Group: The Object Constraint Language (OCL) Specification, Version 1.1. OMG Document ad/97-08-08, Framingham MA, 1997

[OMG02a] The Object Management Group: UML Profile for CORBA Specification, Version 1.0. OMG Document formal/02-04-01, Framingham MA, 2002

[OMG02b] [OMG02] The Object Management Group: Meta Object Facilities (MOF) Specification 1.4, OMG Document formal/02-04-03, Framingham MA, 2002

[OMG03a] The Object Management Group: The Unified Modeling Language Specification, Version 1.5. Chapter 6: The Object Constraint Language Specification. OMG Document ad/03-03-13, Framingham MA, 2003

[OMG03b] The Object Management Group: UML 2.0 OCL 2nd revised submission. OMG Document ad/03-03-13, Framingham MA, 2003

[OMG03c] The Object Management Group: Common Warehouse Metamodel (CWM) Specification, Version 1.1. OMG Document formal/03-03-02, Framingham MA, 2003

[Puppe88] Puppe, F.: Einführung in Expertensysteme, Springer, 1988

[Voss88] Voss, A.; Voss, H.: A Uniform View on Local Constraint Propagation Methods. Tagungsband der KIFS-87, Springer, Informatik-Fachberichte, 1988

[Willenbacher02] Willenbacher, H.: Interaktive verknüpfungsbasierte Bauwerksmodellierung als Integrationsplattform für den Bauwerkslebenszyklus. Dissertation. Bauhaus-Universität Weimar, 2002

[Warmer99] Warmer, J. B.; Kleppe, A. G.: The object constraint language: precise modelling with UML, Addison Wesley Longman, 1999

Anhang B Quellcode

B1 OCL-Grammatik

Die folgende Grammatik der *Object Constraint Language* wurde Kapitel 6 der UML-Spezifikation Version 1.5 entnommen. [OMG03b]

```
oclFile                 := ( "package" packageName oclExpressions
                             "endpackage" )+

packageName             := pathName

oclExpressions          := ( constraint )*

constraint              := contextDeclaration
                           ( ( "def" name? ":" letExpression* )
                            |
                            ( stereotype name? ":" oclExpression)
                           )+

contextDeclaration      := "context"
                           ( operationContext | classifierContext )

classifierContext       := ( name ":" name )
                           | name

operationContext        := name "::" operationName
                           "(" formalParameterList ")"
                           ( ":" returnType )?

stereotype              := ( "pre" | "post" | "inv" )

operationName           := name | "=" | "+" | "-" | "<" | "<=" |
                           ">=" | ">" | "/" | "*" | "<>" |
                           "implies" | "not" | "or" | "xor" | "and"

formalParameterList     := ( name ":" typeSpecifier
                           ("," name ":" typeSpecifier )*
                           )?

typeSpecifier           := simpleTypeSpecifier
                           | collectionType

collectionType          := collectionKind
                           "(" simpleTypeSpecifier ")"

oclExpression           := (letExpression* "in")? expression

returnType              := typeSpecifier

expression              := logicalExpression
```

```
letExpression           := "let" name
                           ( "(" formalParameterList ")" )?
                           ( ":" typeSpecifier )?
                           "=" expression

ifExpression            := "if" expression
                           "then" expression
                           "else" expression
                           "endif"

logicalExpression       := relationalExpression
                           ( logicalOperator
                             relationalExpression
                           )*
relationalExpression    := additiveExpression
                           ( relationalOperator
                             additiveExpression
                           )?

additiveExpression      := multiplicativeExpression
                           ( addOperator
                             multiplicativeExpression
                           )*

multiplicativeExpression:= unaryExpression
                           ( multiplyOperator
                             unaryExpression
                           )*

unaryExpression         := ( unaryOperator
                             postfixExpression
                           )
                           | postfixExpression

postfixExpression       := primaryExpression
                           ( ("." | "->")propertyCall )*

primaryExpression       := literalCollection
                           | literal
                           | propertyCall
                           | "(" expression ")"
                           | ifExpression

propertyCallParameters  := "(" ( declarator )?
                           ( actualParameterList )? ")"
                           literal := string
                           | number
                           | enumLiteral

enumLiteral             := name "::" name( "::" name )*

simpleTypeSpecifier     := pathName

literalCollection       := collectionKind "{"
                           ( collectionIte m
                           ("," collectionItem )*
                           )?
                           "}"

collectionItem          := expression (".." expression )?

propertyCall            := pathName
                           ( timeExpression )?
                           ( qualifiers )?
                           ( propertyCallParameters )?

qualifiers              := "[" actualParameterList "]"
```

```
declarator              := name ( "," name )*
                           ( ":" simpleTypeSpecifier )?
                           ( ";" name ":" typeSpecifier "="
                             expression
                           )?
                           "|"

pathName                := name ( "::" name )*

timeExpression          := "@" "pre"

actualParameterList     := expression ("," expression)*

logicalOperator         := "and" | "or" | "xor" | "implies"

collectionKind          := "Set" | "Bag" | "Sequence" | "Collection"

relationalOperator      := "=" | ">" | "<" | ">=" | "<=" | "<>"

addOperator             := "+" | "-"

multiplyOperator        := "*" | "/"

unaryOperator           := "-" | "not"

typeName                := charForNameTop charForName*

name                    := charForNameTop charForName*

charForNameTop          := /* Characters except inhibitedChar
                           and ["0"-"9"]; the available
                           characters shall be determined by
                           the tool implementers ultimately.*/

charForName             := /* Characters except inhibitedChar; the
                           available characters shall be determined
                           by the tool implementers ultimately.*/

number                  := ["0"-"9"] (["0"-"9"])*
                           ( "." ["0"-"9"] (["0"-"9"])* )?
                           ( ("e" | "E") ( "+" | "-" )? ["0"-"9"]
                             (["0"-"9"])*
                             "|"\"|"#"|"\'"|"("|")"|"*"|"+"|","|
                             "|"."|"/"|":"|";"|"<"|"="|">"|"@"|
                             ["|"\\"|"]"|"{"|"|"}"
                           )?
string                  := "'"
                           (( ~["’","\\","\n","\r"] )
                             |("\\"
                             ( ["n","t","b","r","f","\\","’","\""]
                              | ["0"-"7"]
                             ( ["0"-"7"] ( ["0"-"7"] )? )?
                             )
                            )
                           )*
```

B2 CML-Grammatik

Im folgenden ist der Inhalt der Datei `cml.jjt` abgedruckt, die als Eingabe für den Präprozessor JJTree verwendet wurde. Neben der Grammatik in EBNF-Notation enthält die Datei nutzerdefinierten Action-Code in geschweiften Klammern und Präprozessoranweisungen beginnend mit einem Rautezeichen.

```
SKIP:
{
      " "
    | "\t"
    | "\n"
    | "\r"
    | "\f"
    | <comment : "--" (~["\n","\r"])* ("\n"|"\r\n")>
}

/*************************************************************************
 Terminals.
*************************************************************************/
TOKEN :
{
    //< SELF: "self" > |


    // if-then-else-endif
    < IF: "if" > |
    < THEN: "then" > |
    < ELSE: "else" > |
    < ENDIF: "endif" > |

    //logical operators
    < AND: "and"  >  |
    < OR: "or" >     |
    < XOR: "xor" >   |
    < IMPLIES : "implies" > |

    < EQUALS :          "="    > |
    < BIGGER:           ">"    > |
    < SMALLER:          "<"    > |
    < BIGGER_OR_EQUAL:  ">="   > |
    < SMALLER_OR_EQUAL: "<="   > |
    < UNEQUAL:          "<>"   > |

    < PLUS:   "+" >  |
    < MINUS:  "-" >  |

    < TIMES:     "*" >  |
    < DIVIDED_BY: "/" >  |

    //unaryOperators
    < NOT : "not" >     |


    < name           :  ["a"-"z", "A"-"Z"] ( ["a"-"z", "A"-"Z", "0"-"9", "_"
       ] )* > |

    < INTEGER_LITERAL: ["0"-"9"] (["0"-"9"])* > |

    < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
    > |
```

```
    < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ > |


    < STRING_LITERAL:   // from Java1.1 grammar
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "'"
    >
}


/*****************************************************************************
  Productions.
*****************************************************************************/

void expression() #void :
{}
{
  logicalExpression()
}


void ifExpression() :
{}
{
  <IF>    expression()
  <THEN>  expression()
  <ELSE>  expression()
  <ENDIF>
}

void logicalExpression() #void :
{}
{
  relationalExpression()
  (
    <OR> relationalExpression()  #OrNode(2)
    |
    <AND> relationalExpression() #AndNode(2)
    |
    <XOR> relationalExpression() #XorNode(2)
    |
    <IMPLIES> relationalExpression() #ImpliesNode(2)
  )*
}

void relationalExpression() #void :
{}
{
  additiveExpression()
  (
    <EQUALS> additiveExpression() #IsEqualNode(2)
    |
    <BIGGER> additiveExpression() #IsBiggerNode(2)
    |
    <SMALLER> additiveExpression() #IsSmallerNode(2)
    |
    <BIGGER_OR_EQUAL> additiveExpression() #IsBiggerOrEqualNode(2)
    |
    <SMALLER_OR_EQUAL> additiveExpression() #IsSmallerOrEqualNode(2)
    |
```

```
    <UNEQUAL> additiveExpression() #IsNotEqualNode(2)
  )?
}

void additiveExpression() #void :
{}
{
  multiplicativeExpression()
  (
    <PLUS>  multiplicativeExpression() #AddNode(2)
    |
    <MINUS> multiplicativeExpression() #SubtractNode(2)
  )*
}

void multiplicativeExpression() #void :
{}
{
  unaryExpression()
  (
    <TIMES> unaryExpression() #MultiplicationNode(2)
    |
    <DIVIDED_BY> unaryExpression() #DivisionNode(2)
  )*
}

void unaryExpression() #void :
{}
{
  <MINUS> primaryExpression() #UnaryMinusNode(1)
  |
  <NOT>   primaryExpression() #NotNode(1)
  |
  postfixExpression()
}

void postfixExpression() :
{}
{
  primaryExpression()
  ( ("." | "->") propertyCall() )*
}

void primaryExpression() #void :
{}
{
  (
    literal()
    | propertyCall()
    | "(" expression() ")"
    | ifExpression()
  )
}

void literal() #void :
{
  Token t;
}
{
  (
    t = <STRING_LITERAL>
    {
      jjtThis.theString = t.image;
    }
  ) #StringLiteralNode
  |
  number()
}
```

```
void number() #void : {
  Token t;
}
{
  (
    t = <INTEGER_LITERAL>
    {
      jjtThis.theInteger = Integer.parseInt(t.image);
    }
  ) #IntegerLiteralNode
  |
  (
    t = <FLOATING_POINT_LITERAL>
    {
      jjtThis.theDouble = Double.parseDouble(t.image);
    }
  ) #DoubleLiteralNode
}

void propertyCall() #PropertyCallNode : {
  Token t;
}
{
  t=<name>
  {
    jjtThis.property = t.image;
  }
  (
   "(" ( (declarator())? expression() )? ")"
  )?
}

void declarator() :
{
  Token variableName;
}
{
  variableName=<name>
  "|"
  {
    jjtThis.variableName = variableName.image;
  }
}
```

B3 Erweiterungen der AKO-Schnittstelle

```
module AKO_2001
{
  …
  typedef sequence<AKO_Constraint> AKO_ConstraintSequence;
  …
  enum AKO_ExceptCode {
    …
    ,EXPRESSION_NOT_VALID, INTERPRETER_ERROR, CONSTRAINT_NOT_FOUND
  };

  interface AKO_Class
  {
    …
    AKO_Constraint CreateConstraintByString( in string aConstraintName, in
        string anExpression ) raises (AKO_Exception);
    AKO_Constraint DeleteConstraintByString( in string aConstraintName )
        raises (AKO_Exception);
    AKO_Constraint GetConstraintByString( in string aConstraintName, in
        ScopeType aScope) raises (AKO_Exception);
    AKO_ConstraintSequence GetConstraints( in ScopeType aScope) raises
        (AKO_Exception);
  };

  interface AKO_Instance
  {
    …
    AKO_Constraint CreateConstraintByString( in string aConstraintName, in
        string anExpression ) raises (AKO_Exception);
    AKO_Constraint DeleteConstraintByString( in string aConstraintName )
        raises (AKO_Exception);
    AKO_Constraint GetConstraintByString( in string aConstraintName, in
        ScopeType aScope) raises (AKO_Exception);
    AKO_ConstraintSequence GetConstraints() raises (AKO_Exception);
    boolean EvaluateConstraint( in AKO_Constraint theConstraint ) raises
        (AKO_Exception);
    boolean EvaluateAllConstraints() raises (AKO_Exception);
  };

  interface AKO_Constraint
  {
    AKO_ObjectID GetID() raises (AKO_Exception);
    void SetNameByString(in string aName) raises (AKO_Exception);
    string GetNameAsString() raises (AKO_Exception);
    void SetExpression( in string s ) raises (AKO_Exception);
    string GetExpression() raises (AKO_Exception);
    boolean Evaluate(in AKO_Instance anInst ) raises (AKO_Exception);
  }
}
```