Universität Stuttgart
Fakultät Informatik

Prüfer: Prof. Erhard Plödereder
Betreuer: Thomas Eisenbarth

Beginn: 15. Mai 2000
Ende: 15. November 2000

CR-Klassifikation D.2.2, D.2.7, D.2.11

Diplomarbeit Nr. 1858

Beschreibung einer halbautomatisch abgeleiteten Architektur mit UML-Ausdrucksmitteln

Gregor Schiele

Institut für Informatik

Breitwiesenstraße 20-22

D-70565 Stuttgart

Dieses Dokument wurde unter Verwendung der Vorlage *Rob-Style* erstellt.
©1998 Robert Sakretz

Kurzfassung

Zur effektiven Wartung eines Softwaresystems ist es nötig, die Architektur des Systems zu erkennen und zu verstehen. Werkzeuge und Methoden zur Unterstützung dieses Architekturerkennungsprozesses sind Gegenstand intensiver Forschung. Diese Diplomarbeit konzentriert sich auf die Modellierung und Präsentation der im Rahmen des Projekts *Bauhaus* ermittelten Architekturbeschreibung eines Softwaresystems mit UML-Ausdrucksmitteln. Innerhalb des Projekts *Bauhaus* wird eine Architekturbeschreibung durch einen *Resource Flow Graphen* repräsentiert. Dieser stellt die Quelldarstellung der zu entwickelnden Transformationen dar. Das in dieser Diplomarbeit erarbeitete Konzept ermöglicht es, eine Architekturbeschreibung in die UML zu transformieren, in einer geeigneten Visualisierungskomponente darzustellen und Änderungen, die ein Benutzer an der UML-Modellierung vornimmt, zurück zu transformieren und in die Architekturbeschreibung zu integrieren. Eine prototypische Realisierung dieses Konzepts wird beschrieben. Als Visualisierungskomponente können existierende UML-Werkzeuge Verwendung finden, da das Standardformat *XML Metadata Interchange* unterstützt wird.

Inhaltsverzeichnis

1 Einleitung 1

1.1 Projekt Bauhaus . . . 1
1.2 Aufgabe . . . 3
1.3 Verwendete Notationen . . . 3
1.4 Kapitelübersicht . . . 4

2 Grundlegende Informationen 5

2.1 Resource Flow Graph . . . 5
2.1.1 Definition und Aufbau . . . 5
2.1.2 Inhalt und Verwendung . . . 7
2.1.3 Basis-RFG . . . 8
2.1.4 Erweiterungsmöglichkeiten . . . 13

2.2 Unified Modeling Language . . . 13
2.2.1 Geschichte der UML . . . 13
2.2.2 Grundlagen und Ziele der UML . . . 14
2.2.3 UML-Metamodell . . . 15
2.2.4 UML-Notation . . . 16
2.2.5 Erweiterungsmöglichkeiten . . . 22

3 Rahmenkonzeption 23

3.1 Anforderungen . . . 24
3.1.1 Anforderungen an den Resource Flow Graphen . . . 24
3.1.2 Anforderungen an die Modellierung . . . 26

3.2 Grobentwurf . . . 27
3.2.1 Grundlegende Festlegungen . . . 27
3.2.2 Prinzipielle Modellierung einer Architekturbeschreibung . . . 30
3.2.3 Integration mit einem UML-Werkzeug . . . 31
3.2.4 Genaue Modellierung einer Architekturbeschreibung . . . 39
3.2.5 Weiteres Vorgehen . . . 40

4 Modellierung atomarer Komponenten 41

4.1 Vorliegende Informationen im RFG . . . 42

4.2 Semantisch äquivalente UML-Sprachmittel . . . 44
4.2.1 Modellierung von atomaren Komponenten . . . 44
4.2.2 Modellierung der in ACs enthaltenen Elemente . . . 46
4.2.3 Modellierung von Elementen, die keiner AC angehören . . . 50

4.3 Vorgeschlagene Modellierung 50
4.3.1 Transformation nach UML 50
4.3.2 Rücktransformation in einen RFG 62
4.3.3 Erlaubte Änderungen und Integration mit dem RFG 65

5 Weitere Modellierungen 67

5.1 Modellierung von Subsystemen 67
5.1.1 Vorliegende Informationen im RFG 67
5.1.2 Semantisch äquivalente UML-Sprachmittel 68
5.1.3 Vorgeschlagene Modellierung 69

5.2 Modellierung von Konnektoren 72
5.2.1 Vorliegende Informationen im RFG 73
5.2.2 Semantisch äquivalente UML-Sprachmittel 75
5.2.3 Vorgeschlagene Modellierung 77

5.3 Modellierung von Protokollen 79
5.3.1 Derzeitige Form der vorliegenden Informationen im RFG 80
5.3.2 Semantisch äquivalente UML-Sprachmittel 82
5.3.3 Vorgeschlagene Modellierung 84

6 Prototypische Realisierung 87

6.1 Realisierung der Transformationswerkzeuge 88
6.1.1 Notwendige Anpassungen der Transformationen 89
6.1.2 Realisierungsalternativen 90
6.1.3 Realisierung 93

6.2 Realisierung des Integrationswerkzeugs 94

6.3 Realisierte Hilfswerkzeuge 95
6.3.1 Erstellen der Architektur-View 96
6.3.2 Auftrennen der Architektur-View in User- und Base-View 96
6.3.3 Einfügen zusätzlicher Abhängigkeiten 96
6.3.4 Minimieren des UML-Modells 97
6.3.5 Korrigieren der XMI-Ausgabe von *Together Control Center* 97

7 Abschluß 99

7.1 Zusammenfassung 99

7.2 Bewertung des Projekts und der Ergebnisse 100

7.3 Ausblick 101

Literaturverzeichnis 105

Tabellenverzeichnis

Tabelle 2-1: Knoten im Basis-RFG 8
Tabelle 2-2: Kanten im Basis-RFG 11
Tabelle 2-3: Reference-Kanten im Basis-RFG 12
Tabelle 2-4: Signature-Kanten im Basis-RFG 12
Tabelle 2-5: Of_Type-Kanten im Basis-RFG 12
Tabelle 2-6: Kanten für Module im Basis-RFG 13
Tabelle 4-1: Knoten der AC-Erkennung 43
Tabelle 4-2: Kanten der AC-Erkennung 44
Tabelle 4-3: Transformation ACs und Basis-RFG nach UML 51
Tabelle 4-4: Zuordnung von Knotensichtbarkeiten zu Sichtbarkeiten in der UML 55
Tabelle 4-5: Modellierung der Klassifikation eines Unterprogramms 59
Tabelle 4-6: Rücktransformation von Klassen und ihrem Inhalt in einen RFG . 62
Tabelle 5-1: Knoten der Protokollerkennung 81
Tabelle 5-2: Kanten der Protokollerkennung 82
Tabelle 7-1: Zuordnung zwischen Konstrukten im RFG und in der UML 99

Abbildungsverzeichnis

Abbildung 1-1: Ziel des Projekts Bauhaus 2
Abbildung 2-1: Ansichten eines RFGs 6
Abbildung 2-2: Konzeptionelles RFG-Metamodell 6
Abbildung 2-3: Generierung eines RFGs 7
Abbildung 2-4: Klassenhierarchie der Knoten im Basis-RFG 8
Abbildung 2-5: Beispiele für die Modellierung von Typen im RFG 9
Abbildung 2-6: Klassenhierarchie der Kanten im Basis-RFG 10
Abbildung 2-7: Ausschnitt aus dem UML-Metamodell 15
Abbildung 2-8: Klassendiagramm 17
Abbildung 2-9: Objektdiagramm 18
Abbildung 2-10: Anwendungsfalldiagramm 18
Abbildung 2-11: Sequenzdiagramm 19
Abbildung 2-12: Kollaborationsdiagramm 19
Abbildung 2-13: Zustandsdiagramm 20
Abbildung 2-14: Aktivitätsdiagramm 20
Abbildung 2-15: Komponentendiagramm 21
Abbildung 2-16: Einsatzdiagramm 21
Abbildung 3-1: Modellierung mehrerer Architekturen eines SW-Systems . . . 30
Abbildung 3-2: Anwendungsfall bei enger Integration 31
Abbildung 3-3: Anwendungsfall bei loser Integration 32
Abbildung 3-4: Prinzipielle Integration 37
Abbildung 3-5: Integration der Änderungen mit dem ursprünglichen RFG . . 38
Abbildung 4-1: Beispiel für eine AC im RFG 42
Abbildung 4-2: Klassenhierarchie der Knoten der AC-Erkennung 43
Abbildung 4-3: Klassenhierarchie der Kanten der AC-Erkennung 43
Abbildung 4-4: Alternative Darstellungen eines Pakets in UML 45
Abbildung 4-5: Alternative Darstellungen einer Klasse in UML 45
Abbildung 4-6: Darstellung einer Utility-Klasse in UML 46
Abbildung 4-7: Verfeinerte Abbildung von Typen 48
Abbildung 4-8: Realisierung einer ungerichteten Assoziation mit Attributen . 49
Abbildung 4-9: Vorgeschlagene Stereotyphierarchie 51
Abbildung 4-10: Verschiedene Darstellungsformen für Stereotypen am Beispiel einer Datenbank 52
Abbildung 4-11: Modellierung eines Atomic_Component-Knotens in UML . . . 52
Abbildung 4-12: Verschiedene Darstellungen eingebetteter Typen 54
Abbildung 4-13: Modellierung eines Typs in UML 56
Abbildung 4-14: Verschiedene Darstellungen eingebetteter Typen 58
Abbildung 4-15: Klassenhierarchie der Kanten der AC-Erkennung 65
Abbildung 5-1: Beispiel für ein Subsystem im RFG 67
Abbildung 5-2: Verschiedene Darstellungen eines UML-Subsystems 69

Abbildung 5-3: Beispiel einer Subsystemabbildung 70
Abbildung 5-4: Beispiel für einen Konnektor 73
Abbildung 5-5: Klassenhierarchie der Knoten der Konnektorenerkennung . 74
Abbildung 5-6: Klassenhierarchie der Kanten der Konnektorenerkennung . 74
Abbildung 5-7: Beispiel für einen ADO-Konnektor im RFG 74
Abbildung 5-8: Darstellung einer Assoziationsklasse in UML 75
Abbildung 5-9: Darstellung einer Kollaboration in UML 76
Abbildung 5-10: Darstellung einer generischen Kollaboration in UML 76
Abbildung 5-11: Generische Kollaboration für die Modellierung von ADO-Konnektoren 77
Abbildung 5-12: Beispielprotokoll 80
Abbildung 5-13: Beispiel für ein Protokoll im RFG 82
Abbildung 5-14: Vorbedingungen für Operationen in UML 84
Abbildung 5-15: Modellierung eines Beispielprotokolls in UML 85
Abbildung 6-1: Zusammenspiel der beteiligten Werkzeuge 87
Abbildung 6-2: Aufbau der Transformationswerkzeuge in *Ada* *90*
Abbildung 6-3: Aufbau der Transformationswerkzeuge mit XSLT 92
Abbildung 6-4: Aufbau der Transformationswerkzeuge in Java 93
Abbildung 6-5: Zur Realisierung verwendete Java-Pakete 93
Abbildung 6-6: Einfügen einer neuen Kante durch *addDependsOn* *97*
Abbildung 6-7: Fehler beim XMI-Export mit *Together Control Center* *98*

Listingverzeichnis

Listing 3-1: Transformation einer View nach UML 39
Listing 3-2: Rücktransformation eines UML-Modells in einen RFG 39
Listing 4-1: Transformation eines AC-Knotens nach UML 53
Listing 4-2: Transformation eines Type-Knotens nach UML 54
Listing 4-3: Transformation von Is_Part_Of_Type- und Typedefed_By-Kanten nach UML 55
Listing 4-4: Transformation eines Member-Knotens nach UML 57
Listing 4-5: Transformation von Variable- und Constant-Knoten nach UML . . 58
Listing 4-6: Transformation eines Subprogram-Knotens nach UML 60
Listing 4-7: Transformation von Kanten zu bzw. von Unterprogrammen nach UML 61
Listing 4-8: Transformation einer Reference-Kante nach UML 62
Listing 4-9: Rücktransformation einer Klasse in den RFG 63
Listing 5-1: Erweiterte Transformation eines Atomic_Component-Knotens nach UML 70
Listing 5-2: Rücktransformation eines Subsystems in UML in einen RFG . . . 71
Listing 5-3: Transformation von ADO-Konnektoren nach UML 78
Listing 5-4: Rücktransformation von Konnektor-Kollaborationen in einen RFG 79
Listing 6-1: Ablauf des Werkzeugs *gxl2uml* *94*
Listing 6-2: Ablauf des Werkzeugs *mergeRFGs* *95*

Abkürzungen

AC	Atomic Component
AdaJNI	Ada Java Native Interface
ADL	Architecture Description Language
ADO	Abstract Dataobject
ADT	Abstract Datatype
API	Application Programming Interface
ASE	Abstract State Encapsulation
CASE	Computer Aided Software Engineering
DOM	Document Object Model
DTD	Document Type Definition
FhG IESE	Fraunhoferinstitut für experimentelles Software Engineering
GXL	Graph Exchange Language
HC	Hybrid Component
IML	Intermediate Language
JNI	Java Native Interface
MB	Megabyte
OLE	Object Linking and Embedding

OMG	Object Management Group
OMT	Object Modeling Technique
OOA&D	Objektorientierte Analyse und Design
OOSE	Object-Oriented Software Engineering
RFG	Resource Flow Graph
RS	Related Subprograms
SAX	Simple API for XML
SW	Software
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Einleitung

Kapitel 1

Die Bedeutung der Wartung von Software (SW) ist allgemein bekannt. Über den gesamten Lebenszyklus eines SW-Systems betrachtet, verursacht die Wartung den größten Kostenanteil. Für die effektive Wartung eines SW-Systems ist es erforderlich, daß ein Wartungsingenieur deren Ist- und Soll-Architektur kennt. Ohne dieses Wissen ist es nicht möglich, die Folgen einer Wartungsaktivität abzuschätzen. Die Einführung von Fehlern durch unerkannte Seiteneffekte, Code Clones, bzw. eine allgemein schlechtere Softwarequalität und damit eine sinkende Wartbarkeit könnten die Folgen sein. Dieser von jedem Wartungsingenieur initial zu durchlaufende Prozeß des Programmverstehens ist im allgemeinen sehr zeit- und damit kostenintensiv. Eine eventuell vorhandene Dokumentation der Systemarchitektur veraltet oftmals, soweit sie nicht ständig aktualisiert wird, was wiederum sehr aufwendig ist.

Werkzeuge, die den Wartungsingenieur bei diesen Tätigkeiten unterstützen, sind daher erstrebenswert und derzeit intensives Forschungsgebiet. Einen initialen Überblick gibt z. B. [KosPlö96].

1.1 Projekt Bauhaus

In diesem Zusammenhang führt das *Institut für Informatik der Universität Stuttgart, Abteilung für Programmiersprachen und Compilerbau*, in Kooperation mit dem *Fraunhoferinstitut für experimentelles Software Engineering* (FhG IESE) *Kaiserslautern*, das Projekt *Bauhaus* durch.

Dieses Projekt verfolgt das Ziel, Wartungsingenieure bei der Erkennung der Ist- und Soll-Architektur eines SW-Systems, der Nachführung der Architektur bei Änderungen, der Abschätzung der Folgen einer Änderung sowie deren Durchführung zu unterstützen. Forschungsgegenstand des Projekts sind "Beschreibungsmittel sowie Analysemethoden und -werkzeuge, die es ermöglichen, Architekturen zu extrahieren, darzustellen und zu manipulieren." ([Eis99], S. 17)

Ausgangspunkt der Untersuchungen ist ausschließlich der Quelltext des Systems, um auch Systeme mit fehlerhafter, bzw. veralteter oder fehlender Dokumentation bearbeiten zu können. Dies zeigt Abbildung 1-1 bildhaft.

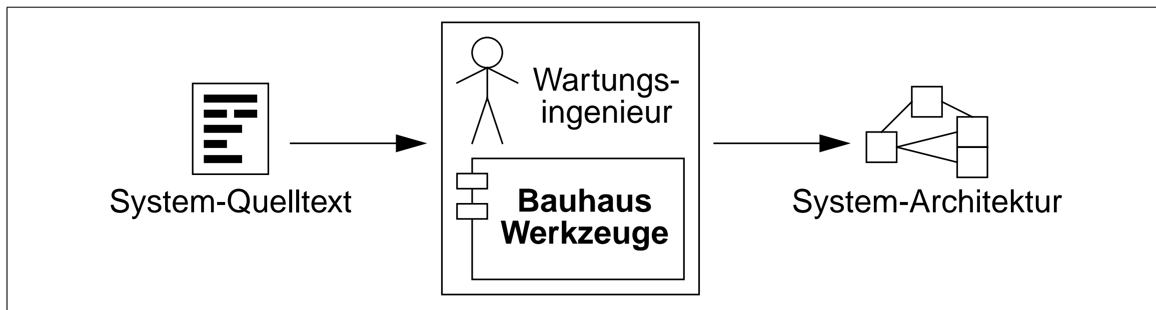


Abbildung 1-1: Ziel des Projekts Bauhaus

Aus dem Quelltext eines SW-Systems wird zunächst eine Darstellung in einem weitgehend programmiersprachenunabhängigen Zwischencode, der *Intermediate Language* (IML), erzeugt. Auf dieser Darstellung können im weiteren Verlauf Analysen durchgeführt werden. Sie enthält ausschließlich Informationen auf Implementierungsebene, mit deren Hilfe z. B. Datenflußanalysen durchgeführt werden können. Weitere Informationen zu IML finden sich in [Roh98].

Abgeleitet aus der detaillierten IML-Darstellung wird zudem eine stärker abstrahierte Systemsicht in Form eines *Resource Flow Graphen* (RFG) erstellt. Alle derzeitigen Analysen zur Erkennung der Architektur arbeiten mit dieser Sicht, in der Zukunft sind auch Analysen denkbar, die eine Kombination beider Sichten verwenden, beispielsweise, um die Ergebnisse von Datenflußanalysen einzubeziehen. Die Ergebnisse der Architekturerkennung werden in einem RFG gespeichert. Da RFGs für diese Arbeit von zentraler Bedeutung sind, werden sie im Kapitel „Resource Flow Graph“ auf Seite 5 genauer behandelt.

Da sich gezeigt hat, daß automatische Verfahren den Grad der menschlichen Erkennungsqualität nicht erreichen, erfolgt die Architekturerkennung in weiten Teilen in Form eines interaktiven, iterativen Prozesses. Dem Wartungsingenieur bzw. Benutzer wird in einer modifizierten Version des Grapheneditors *Rigi* ([Til93]) der RFG präsentiert. Der Benutzer wählt die jeweils durchzuführenden Analysen aus, bewertet die Ergebnisse, führt gegebenenfalls Korrekturen durch und liefert zusätzlich benötigte Informationen. Dadurch entsteht schrittweise eine Architekturbeschreibung im verwendeten RFG. Genauere Informationen über den semiautomatischen Prozeß der Architekturerkennung sind in [Kos00], Kapitel 9 zu finden.

Die Architekturerkennung in Bauhaus besteht derzeit grob aus vier Bereichen:

1. Die **Erkennung atomarer Komponenten** befaßt sich mit dem Auffinden der grundlegenden (atomaren) Komponenten eines Softwaresystems.
2. Die **Subsystemerkennung** versucht darauf aufbauend Komponenten höherer Ordnung zu ermitteln.
3. Bei der **Konnektorenerkennung** wird versucht, Verbindungen bzw. Kommunikationskanäle zwischen Komponenten aufzufinden und zu klassifizieren.
4. Die **Protokollerkennung** extrahiert Informationen über die zulässige Ver-

wendung von Komponenten und Konnektoren, insbesondere in welcher Reihenfolge Unterprogramme aufgerufen werden dürfen.

Während die Komponentenerkennung als Basis der anderen Bereiche bereits relativ weit fortgeschritten ist, sind die übrigen Teilbereiche noch relativ neu. Alle werden in späteren Kapiteln näher betrachtet.

Weitere Informationen zum Projekt *Bauhaus* sind unter anderem in [Eis99] zu finden. [Cze00b] beschreibt eine mit Hilfe der *Bauhaus*-Werkzeuge durchgeführte Architekturerkennung. Einen Ausblick über zukünftig geplante Aktivitäten gibt [Cze00]. Für genauere Informationen über verschiedene Ansätze des Programmverstehens siehe [KosPlö96].

1.2 Aufgabe

Nach Abschluß der Architekturerkennung, bzw. nach einem hinreichend großen Teil der dazu notwendigen Arbeiten, liegt die abgeleitete Architektur des SW-Systems als RFG vor. Sie kann derzeit lediglich über die unmittelbare Anzeige des Graphen visualisiert werden. Angestrebt wird die Darstellung der Architektur in einer verbreiteten, vorzugsweise standardisierten Form, um die Ergebnisse einfach an andere Stellen weitergeben, bzw. mit anderen Werkzeugen weiterbearbeiten zu können. Neben Architekturbeschreibungssprachen (Architecture Description Language, ADL) (siehe [MedTay00]) bietet sich hierfür die *Unified Modeling Language* (UML) an.

Aufgabe dieser Arbeit ist es, die Ausdrucksmittel von UML daraufhin zu untersuchen, ob und wie sie die Modellierung der als RFG vorliegenden Ist-Architektur eines Systems zulassen. Dazu soll einerseits festgestellt werden, wozu sich die derzeit vorliegende Information im RFG und anderen, durch Analysen erzeugten Informationsquellen, benutzen läßt. Andererseits ist zu untersuchen, welche Information gegebenenfalls darüberhinaus notwendig bzw. wünschenswert ist.

In einem zweiten Schritt sollen verschiedene UML-fähige Computer Aided Software Engineering (CASE) Werkzeuge daraufhin untersucht werden, inwieweit sie sich für eine Anbindung an die *Bauhaus*-Werkzeuge als manipulatives Backend eignen. Dazu sind die Schnittstellen der einzelnen CASE-Werkzeuge auf ihre Tauglichkeit hin zu betrachten. Die aus dem RFG abgeleitete und mit UML modellierte Architektur soll dargestellt und manipuliert werden können. Änderungen sollen sowohl zurück in die RFG-Darstellung übertragen als auch von RFG-Seite aus in das CASE-Werkzeug übermittelt werden können. Die notwendige Anbindung für ein geeignetes Werkzeug ist prototypisch zu implementieren.

1.3 Verwendete Notationen

Soweit wie möglich wurde versucht, englische Begriffe zu vermeiden, bzw. diese ins Deutsche zu übersetzen. Ausgenommen hiervon sind Eigennamen, die zudem kursiv geschrieben werden (z. B. *Object Management Group*), Programmtext, und feststehende Begriffe aus der Informatik, für die kein verbreiteter deutscher Begriff existiert (z. B. Software). Programmtext wird zur besseren Unterscheidung in Courier gesetzt (z. B. `if true then`). Teilweise werden hinter den deutschen Bezeichnungen

für UML-Konstrukte - bzw. im RFG vorliegende Knoten und Kanten - deren offizielle Namen im UML-Metamodell - bzw. in der Implementation des RFGs - angegeben (z. B. Klasse (`Class`)).

Abbildungen sind im Regelfall in UML-Notation angegeben. Wenn es für eine übersichtliche Darstellung nötig war, wurde von dieser Regel jedoch abgewichen. Eine Kurzübersicht über die UML-Notation findet sich im Kapitel „UML-Notation" auf Seite 16. Für eine genauere Beschreibung sei auf die dort angegebenen Quellen verwiesen. Klar unterschieden werden muß zwischen Abbildungen, die zur Erläuterung angegebener Sachverhalte eingesetzt werden, und solchen, die die Notation bestimmter UML-Konstrukte bzw. Beispiele für die gewählte Modellierung zeigen. Obwohl beide in UML-Notation angegeben werden, dürfen sie nicht vermischt werden. Beispielsweise zeigt Abbildung 4-1 auf Seite 42 einige in einem RFG vorkommende Objekte und ihre Verbindungen, während Abbildung 4-11 auf Seite 52 verdeutlicht, wie eine Modellierung in UML aussehen könnte.

1.4 Kapitelübersicht

Im folgenden Kapitel („Grundlegende Informationen" auf Seite 5) werden grundlegende Informationen, insbesondere über den RFG und die UML gegeben, die zum Verständnis dieser Arbeit benötigt werden. Danach werden im Kapitel „Rahmenkonzeption" auf Seite 23 die gegebenen Anforderungen an die Arbeit untersucht und ein Rahmenkonzept für die Modellierung einer Architekturbeschreibung mit der UML erarbeitet. Dieses wird in den Kapiteln vier („Modellierung atomarer Komponenten" auf Seite 41) und fünf („Weitere Modellierungen" auf Seite 67) erweitert, indem die Modellierung der in den Teilbereichen der Architekturerkennung in Bauhaus ermittelten architektonischen Informationen entworfen wird. Die Beschreibung der prototypischen Realisierung dieser Modellierungen ist in Kapitel sechs („Prototypische Realisierung" auf Seite 87) enthalten. Kapitel sieben („Abschluß" auf Seite 99) gibt abschließend eine Bewertung der Arbeit und ihrer Ergebnisse sowie einen Ausblick auf weitere sinnvolle Untersuchungsbereiche, die sich im Verlauf dieser Diplomarbeit ergaben.

Grundlegende Informationen

Kapitel 2

Ziel dieser Arbeit ist die Modellierung der als RFG vorliegenden Ist-Architektur eines SW-Systems mit Ausdrucksmitteln der UML.

Es existieren verschiedene Definitionen für die **Architektur eines Softwaresystems**. Den folgenden Betrachtungen wird die Definition von *Shaw* und *Garlan* zugrundegelegt. Demnach beinhaltet eine Software-Architektur abstrakt betrachtet die Beschreibung der Elemente, aus denen ein System aufgebaut ist, der Interaktionen zwischen diesen Elementen, der Muster nach denen die Elemente zusammengefaßt werden sowie der Eigenschaften bzw. Einschränkungen dieser Muster. Ein konkretes System wird durch eine Menge von Komponenten und den Interaktionen zwischen diesen definiert. (vgl. [ShaGar96], S. 1)

Die Zielgruppe dieser Arbeit besteht vor allem aus Wartungsingenieuren, deren Aufgabe die Anpassung und Weiterentwicklung des untersuchten SW-Systems ist. Die UML wird unter anderem zur Dokumentation und Präsentation eingesetzt. Im Rahmen dieser Arbeit ist dies sogar ihr primärer Zweck, da für größere Arbeiten an der Architektur des SW-Systems in *Bauhaus* andere Darstellungen des RFGs gewählt werden. Aus diesem Grund sollten auch computertechnisch weniger erfahrene Benutzer, wie z. B. Manager, zur Zielgruppe gezählt werden.

Bevor eine genauere Diskussion sinnvoll ist, muß ein grober Überblick über Struktur und Inhalt des RFGs und die mit UML gegebenen Möglichkeiten zur Modellierung gewonnen werden. Dazu werden kurz die grundlegenden Konzepte des RFGs vorgestellt und im zweiten Teil des Kapitels eine Übersicht über die UML gegeben, die die wichtigsten Sachverhalte zusammenfaßt. Detailliertere Betrachtungen bleiben späteren Kapiteln vorbehalten.

2.1 Resource Flow Graph

Bei der Erkennung der Architektur eines Softwaresystems werden verschiedene Abstraktionsebenen verwendet. Während IML das System auf relativ konkreter Ebene darstellt, werden die abstrakteren Ebenen mittels des RFG modelliert.

2.1.1 Definition und Aufbau

Ein RFG ist ein gerichteter, bipartiter Graph. Er besteht aus typisierten Knoten und Kanten in Form eines erweiterten, attributierten, abstrakten Syntaxbaums. Knoten stellen architektonisch relevante Elemente des Softwaresystems dar. Beziehungen

zwischen diesen Elementen werden mit Kanten modelliert. Sowohl Knoten als auch Kanten können Attribute enthalten. Zulässige Verbindungen zwischen Knoten und Kanten werden durch deren Typen eingeschränkt. (vgl. [Kos00], S. 47) Ein Beispiel-RFG besteht aus drei Knoten, je einem für die Unterprogramme *X* und *Y* und einem weiteren für eine globale Variable *A*. Über entsprechende Kanten wird modelliert, daß *Y* von *X* aufgerufen wird. Beide Unterprogramme greifen auf *A* zu, *X* lesend, *Y* schreibend. Diesen RFG zeigt Abbildung 2-1 in zwei alternativen Darstellungsformen. Darstellungsform (a) entspricht der üblichen Graphendarstellung, (b) zeigt den gleichen RFG als UML-Objektdiagramm. Im weiteren Verlauf wird die Darstellung als Objektdiagramm gewählt. Informationen über diese Darstellungsform sind im Kapitel „Objektdiagramme (Object Diagrams)“ auf Seite 18 zu finden.

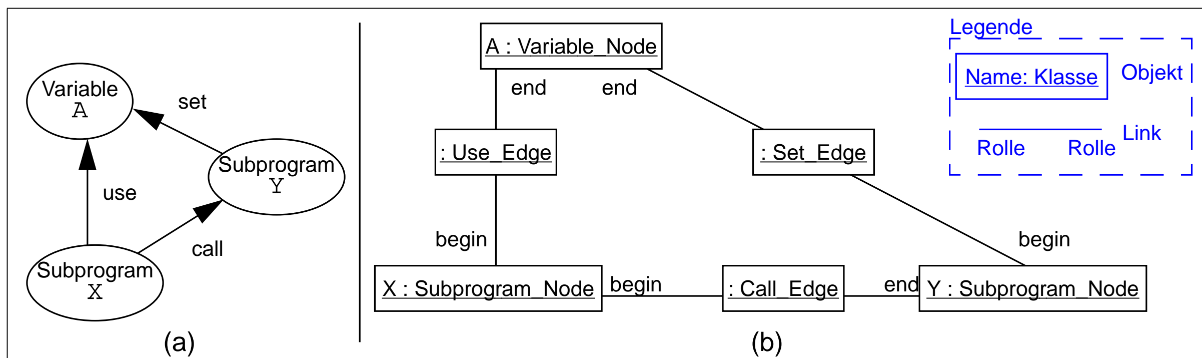


Abbildung 2-1: Ansichten eines RFGs

Jeder Knoten im RFG ist über seine eindeutige Node_Id systemweit identifizierbar. Diese besteht aus dem Namen des zugrundeliegenden Konstrukts im Quelltext (`Object_Name`), dem Namen der Quelltextdatei (`Filename`) und dem Namen des untersuchten Systems (`Program_Name`).

Es ist möglich, verschiedene Sichten (Views) auf einen RFG zu definieren. Eine Sicht stellt konzeptionell einen Teilgraphen des RFGs dar, in dem nur bestimmte Elemente enthalten sind. Knoten und Kanten können in beliebig vielen Sichten enthalten sein. Das resultierende konzeptionelle Metamodell des RFG zeigt Abbildung 2-2.

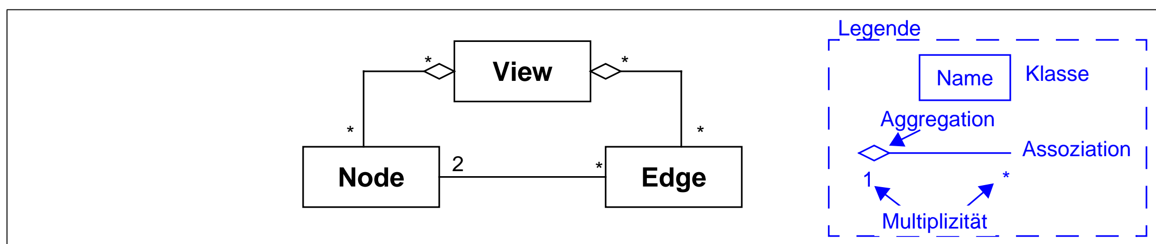


Abbildung 2-2: Konzeptionelles RFG-Metamodell

Sichten haben nichts mit der graphischen Darstellung eines RFGs am Bildschirm zu tun, sondern stellen logische Gruppierungen von Knoten und Kanten im RFG dar.

Sie werden eingesetzt, um (vgl. [GirKos98]):

- Analysen einen einheitlichen und effizienten Mechanismus zu bieten, ihnen die zu analysierenden Teile des RFGs zu übergeben und ihre Ergebnisse zurückzuliefern,
- Zwischenergebnisse, die von verschiedenen Analysen weiterverarbeitet werden, zu sichern, so daß diese nicht mehrmals berechnet werden müssen,
- mehreren Benutzern unterschiedliche Sichten auf ein Systems zu ermöglichen, z. B. unterschiedlich detaillierte und
- verschiedene Aspekte des Systems, z. B. Ist- und Soll-Architektur im gleichen RFG modellieren zu können.

Verschiedene Eigenschaften bzw. Attribute von Knoten werden nicht für den gesamten RFG global sondern pro View angegeben, so z. B. die oben genannte Sichtbarkeit eines Knotens im umschließenden Kontext. Es ist jederzeit möglich, neue Sichten zu erstellen und in einen RFG einzufügen. Einige Sichten sind jedoch vordefiniert. Für diese Arbeit sind vor allem zwei Sichten von Bedeutung,

1. die **Base-View** und
2. die **User-View**,

deren Verwendung im folgenden kurz erläutert wird.

2.1.2 Inhalt und Verwendung

Ein RFG enthält zunächst grundlegende, aus dem Quelltext abgeleitete Informationen über die statische Struktur eines Softwaresystems, insbesondere globale Variablen, Unterprogramme und benutzerdefinierte Typen in abstrakter Form. Bestimmte Details, die für die Architekturerkennung nicht wichtig sind, sind nicht enthalten (z. B. die genaue Signatur eines Unterprogramms). Diese Informationen werden bei der Erzeugung des RFGs aus der IML-Darstellung des Systems gewonnen und in der **Base-View** abgelegt. Abbildung 2-3 zeigt den groben Ablauf der Generierung eines RFGs. Derzeit dürfen in der Base-View enthaltene Knoten und Kanten nicht mehr modifiziert werden. Der resultierende RFG wird auch Basis-RFG genannt. Er wird im nächsten Teilkapitel genauer behandelt. (vgl. [Eis98], S. 15)

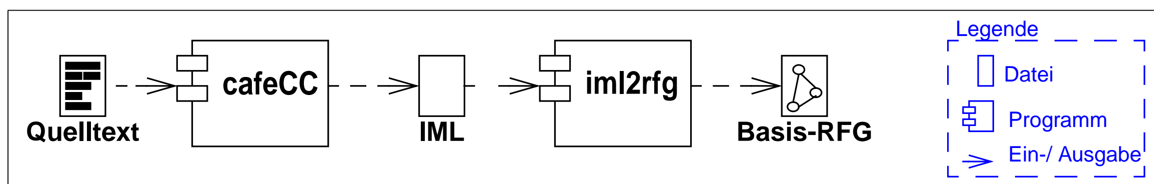


Abbildung 2-3: Generierung eines RFGs

Darüberhinaus enthält ein RFG übergeordnete architektonische Informationen über das modellierte Softwaresystem bzw. die in ihm enthaltenen architektonischen Komponenten und Konnektoren. Diese Informationen werden im Laufe der Architekturerkennung mit Hilfe von Analysen in Form neuer Knoten und Kanten hinzugefügt. Die Ergebnisse werden dem Benutzer präsentiert und von diesem akzeptiert bzw. angepaßt oder verworfen. Akzeptierte Teile der Architekturbeschreibung werden der

User-View hinzugefügt. Nach Abschluß der Arbeiten liegt im RFG die Ist-Architektur des untersuchten Softwaresystems in Form von Knoten- und Kantenbeziehungen vor.

2.1.3 Basis-RFG

Der direkt aus dem Quelltext bzw. der IML-Darstellung des Systems gewonnene Basis-RFG enthält bereits eine Vielzahl verschiedener Knoten und Kanten.

Enthaltene Knoten

Der Basis-RFG enthält Knoten für benutzerdefinierte Typen und deren Teilkomponenten, globale Variablen und Konstanten, Unterprogramme und Module. Alle Knoten sind Teil einer Hierarchie, die Abbildung 2-4 zeigt.

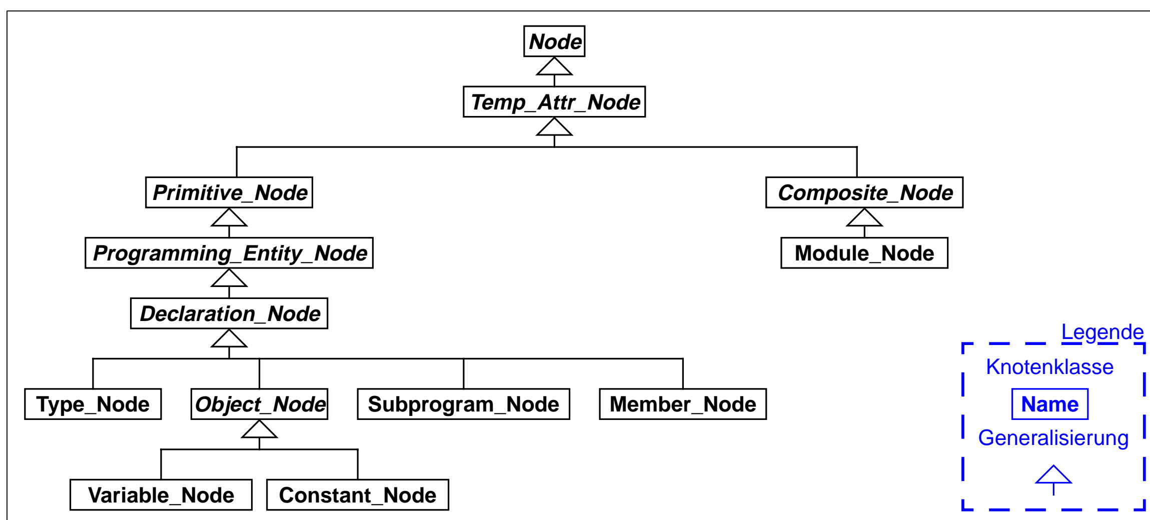


Abbildung 2-4: Klassenhierarchie der Knoten im Basis-RFG

Eine kurze Beschreibung der Knotentypen kann Tabelle 2-1 entnommen werden, genauere Informationen folgen.

Bezeichnung	Beschreibung
`Type_Node`	benutzerdefinierter Typ
`Variable_Node`	globale Variable
`Constant_Node`	globale Konstante
`Subprogram_Node`	Unterprogramm
`Member_Node`	Teilkomponente eines zusammengesetzten Typs bzw. Variable
`Module_Node`	physische Gruppierung von Elementen im untersuchten System

Tabelle 2-1: Knoten im Basis-RFG

Ein wichtiges Attribut (des Typs `Visibility_Type`), das verschiedene im Basis-RFG enthaltene Knoten (Type-, Object-, Subprogram- und Member-Knoten) enthalten, gibt die Sichtbarkeit des Knotens im umschließenden Kontext an, beispielsweise die

Sichtbarkeit eines Unterprogramms im Rahmen eines Moduls. Ein Knoten kann als vollkommen öffentlich (`Public_Visibility`), als eingeschränkt öffentlich (`Restricted_Visibility`) oder als privat (`Private_Visibility`) definiert werden. Alternativ ist es auch möglich, eine unbekannte Sichtbarkeit anzugeben (`Unknown_Visibility`).

Der Begriff des Moduls wird in der Informatik für verschiedene Sachverhalte verwendet. In *Bauhaus* stellen Module im Quelltext vorgefundene, physische Gruppierungen von Programmelementen (z. B. Typen und Unterprogramme) dar. Beispiele hierfür sind eine Datei in einem in der Programmiersprache *C* realisierten System, ein Modul in einem *Modula-2* System oder ein Package in einem *Ada* System.

Die Modellierung von Typen unterliegt einigen Beschränkungen. So sind ausschließlich benutzerdefinierte Typen im RFG enthalten. Für vordefinierte Datentypen, wie z. B. Integer, werden keine Knoten eingefügt. In Programmiersprachen existieren verschiedene Arten von Typen, z. B. primitive Datentypen, Aufzählungstypen, zusammengesetzte Typen, Vereinigungstypen und andere. Im Detail unterscheiden sich teilweise auch prinzipiell gleichartige Typarten von Programmiersprache zu Programmiersprache. In *C* kann z. B. auf einen Vereinigungstyp (union) ungeprüft zugegriffen werden, was in anderen Sprachen nicht möglich ist. Derzeit wird im RFG von dieser Situation abstrahiert. Es existiert nur eine einzige Art von Typknoten, auf den alle Arten von Typen abgebildet werden. Die Teilkomponenten von zusammengesetzten Typen und Vereinigungstypen werden über Member-Knoten modelliert. Weitere Informationen über Art und Struktur des Typs werden nicht übernommen, insbesondere gehen Zeiger- oder Feldinformationen verloren. (vgl. [Kos00], S. 68 ff.) Abbildung 2-5 zeigt zwei Beispiele. Enclosing-Kanten werden in Tabelle 2-2 erläutert.

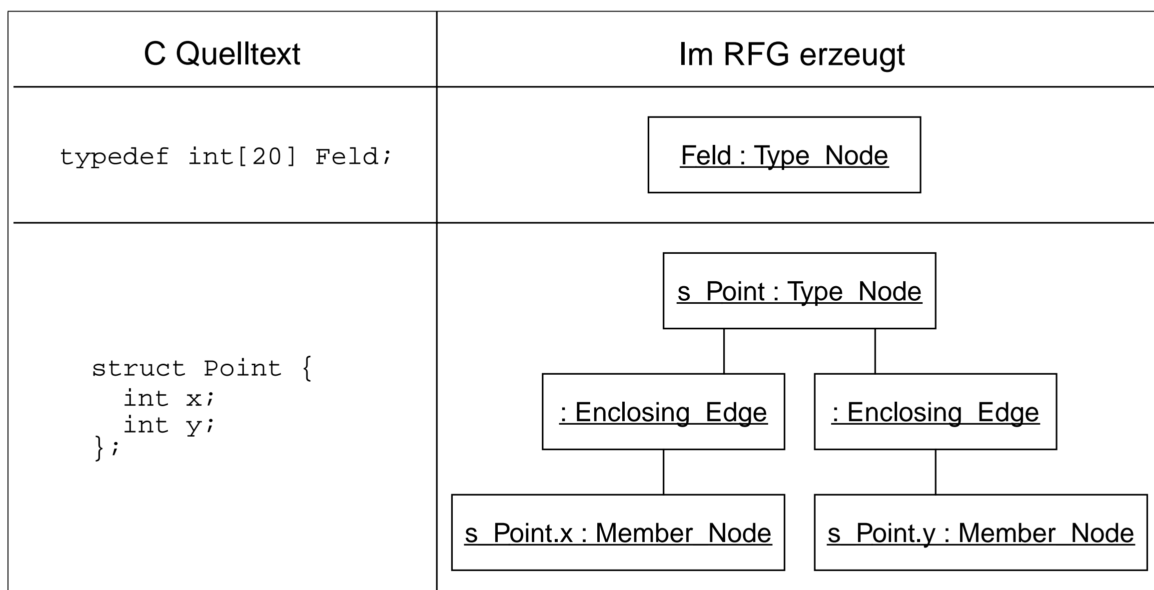


Abbildung 2-5: Beispiele für die Modellierung von Typen im RFG

Wurde der RFG aus dem Quelltext eines in *C* geschriebenen Systems abgeleitet, kann am Namen des Typknotens erkannt werden, ob es sich um einen zusammengesetzten Typ (der Typname beginnt mit ‘s’, gefolgt von einem Leerzeichen), einen Aufzählungstyp (der Typname beginnt mit ‘e’ und einem Leerzeichen), einen Vereinigungstyp (der Typname beginnt mit ‘u’ und einem Leerzeichen) oder eine andere Art von

Typ handelt (der Name beginnt mit einer anderen Zeichenkette). Dies resultiert daraus, daß in *C* diese Typarten verschiedene Namensräume besitzen, die auf diese Weise simuliert werden. Eine allgemeine Möglichkeit, die Art eines Typs zu erkennen, ist dies nicht.

Variable-Knoten werden nur für globale Variablen erzeugt. Lokale Variablen oder Parameter werden nicht als Knoten sondern als Kanten abgebildet. Die Teilkomponenten zusammengesetzter Variablen werden initial nicht als eigene Member-Knoten modelliert. Es gibt aber die Möglichkeit diese Teilkomponenten nachträglich einfügen zu lassen, indem die entsprechenden Member-Knoten des Typs der zusammengesetzten Variable kopiert und an den Variable-Knoten angehängt werden. Neben globalen Variablen sind auch globale Konstanten im RFG vorhanden. Sie ergeben Constant-Knoten. Variable- und Constant-Knoten werden unter dem Begriff Object-Knoten zusammengefaßt.

Es können verschiedene Arten von Unterprogrammen mit Subprogram-Knoten modelliert werden. Welche Art jeweils vorliegt, gibt ein entsprechendes Attribut (vom Typ `Subprogram_Classification_Type`) im Subprogram-Knoten an. Es können Konstruktoren (`Is_Constructor`), Destruktoren (`Is_Destructor`), den Zustand eines Systems ändernde Modifikatoren (`Is_Modifier`) und rein lesende Inspektoren (`Is_Accessor`) unterschieden werden. Ein Subprogram-Knoten ohne Angabe der Unterprogrammart ist ebenfalls zulässig.

Enthaltene Kanten

Eine Vielzahl von Informationen wird im Basis-RFG in Form von Kanten abgelegt, die im folgenden beschrieben werden. Eine Übersicht über diese Kanten zeigt Abbildung 2-6.

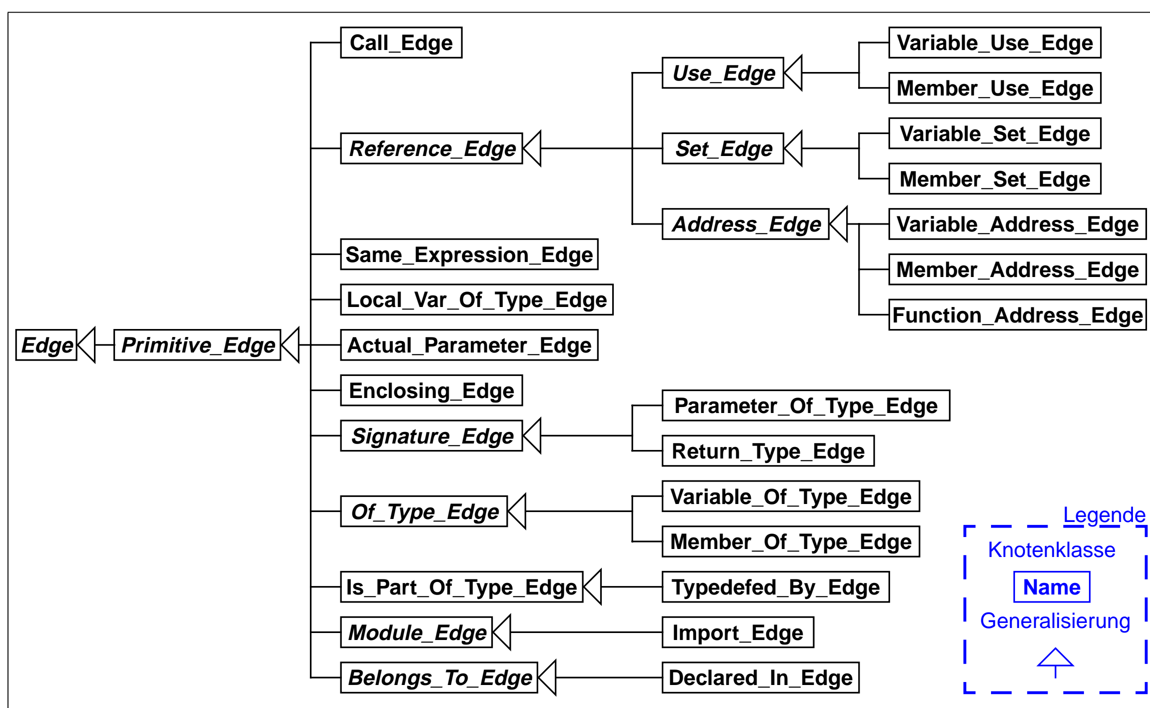


Abbildung 2-6: Klassenhierarchie der Kanten im Basis-RFG

Tabelle 2-2 zeigt die grundlegenden Kanten im Basis-RFG. Zu beachten ist, daß Call-

Bezeichnung	Quelle s	Ziel d	Beschreibung
Actual_Parameter	Subprogram	Object	d wird an s als aktueller Parameter übergeben
Call	Subprogram	Subprogram	s ruft d auf
Local_Var_Of_Type	Subprogram	Type	s hat eine lokale Variable vom Typ d
Is_Part_Of_Type	Type	Type	s wird in der Deklaration von d verwendet
Typedefed_By	Type	Type	s wird strukturell identisch aus d abgeleitet oder ist ein Alias auf d
Enclosing	Member	Type | Object	s ist Teilkomponente von d
Same_Expression	Object	Object	s und d werden im selben Ausdruck verwendet

Tabelle 2-2: Kanten im Basis-RFG

Kanten nur für direkte Aufrufe eines Unterprogramms eingefügt werden. Erfolgt ein Aufruf über einen Funktionszeiger oder durch das Betriebssystem in Form eines Callbacks, führt dies zunächst nicht zu einer Call-Kante. Erst im Verlauf der weiteren Arbeiten könnten entsprechende Analysen solche Call-Kanten einfügen, was bisher allerdings noch nicht erfolgt.

Der Kantentyp *Typedefed_By_Edge* wurde in den RFG aufgenommen, um die Deklaration `typedef` der Programmiersprache *C* abzubilden. Diese erzeugt einen Alias auf einen existierenden Typ. Sie wird aber auch als Möglichkeit verwendet, einen neuen Typ aus einem bestehenden abzuleiten. Eine Typedefed_By-Kante zeigt an, daß der Quelltyp entweder ein Alias auf den Zieltyp ist, oder ein strukturell identischer Typ, der mit Hilfe von d deklariert wurde. Für Sprachen, in denen diese Fälle unterschieden werden können, z. B. *Ada*, sind zusätzliche Unterarten dieses Kantentyps angedacht. Die Typedefed_By-Kante wird unter dem Namen *delineate* in [Kos00], Seite 69 f. beschrieben. (vgl. [Kos00], S.69 f.)

Kanten geben im allgemeinen nicht an wie oft ein bestimmter Zusammenhang besteht, da pro Kantenart immer nur eine Kante ein gegebenes Knotenpaar verbinden kann. So geben z. B. Local_Var_Of_Type-Kanten nicht an wieviele lokalen Variablen eines Typs ein Unterprogramm hat, sondern nur, daß es überhaupt welche besitzt.

Zugriffe auf Programmteile werden - neben Call-Kanten - über Reference-Kanten abgebildet, die Tabelle 2-3 auflistet. Einen Sonderfall bilden Zugriffe auf Teilkomponenten zusammengesetzter Variablen bzw. Konstanten. Wie im letzten Teilkapitel beschrieben, sind diese Teilkomponenten normalerweise in einem RFG nicht vorhanden. Daher ergeben Zugriffe auf sie Reference-Kanten, die auf die Gesamtvariable bzw. -konstante, anstatt auf die entsprechende Teilkomponente, verweisen. Die Infor-

Bezeichnung	Quelle s	Ziel d	Beschreibung
Variable_Use	Subprogram	Object	s greift auf d lesend zu
Member_Use	Subprogram	Member | Object	s greift auf d lesend zu
Variable_Set	Subprogram	Variable	s greift auf d schreibend zu
Member_Set	Subprogram	Member | Variable	s greift auf d schreibend zu
Variable_Address	Subprogram	Object	s ermittelt Adresse von d
Member_Address	Subprogram	Member | Object	s ermittelt Adresse von d
Function_Address	Subprogram	Subprogram	s ermittelt Adresse von d

Tabelle 2-3: Reference-Kanten im Basis-RFG

mation, auf welchen genauen Teil der Variable bzw. Konstante zugegriffen wird, ist in der Reference-Kante enthalten. Werden die Teilkomponenten zusammengesetzter Variablen oder Konstanten durch die oben angesprochene Projektion in den RFG eingefügt, so werden aus dieser Information Reference-Kanten abgeleitet, die direkt auf die zugegriffenen Teilkomponenten verweisen. Ignoriert werden derzeit Zugriffe auf lokale Variablen oder Parameter. Sie ergeben keine Kanten im RFG. (vgl. [Kos00], S. 40)

Die Signatur eines Unterprogramms wird über Signature-Kanten in den RFG übernommen, die Tabelle 2-4 zeigt. Die Parameternamen gehen verloren. Außerdem werden mehrere Parameter des gleichen Typs auf dieselbe Kante abgebildet.

Bezeichnung	Quelle s	Ziel d	Beschreibung
Parameter_Of_Type	Subprogram	Type	s hat einen formalen Parameter vom Typ d
Return_Type	Subprogram	Type	s hat einen Rückgabewert vom Typ d

Tabelle 2-4: Signature-Kanten im Basis-RFG

Tabelle 2-5 beschreibt die im Basis-RFG verwendeten Of_Type-Kanten, mit deren Hilfe der Typ einer Variablen oder der Teilkomponente eines zusammengesetzten Typs angegeben wird.

Bezeichnung	Quelle s	Ziel d	Beschreibung
Variable_Of_Type	Variable	Type	s ist vom Typ d
Member_Of_Type	Member	Type	s ist vom Typ d

Tabelle 2-5: Of_Type-Kanten im Basis-RFG

Schließlich existieren Kanten, um den Inhalt von Modulen bzw. Beziehungen zwischen Modulen festzulegen. Diese zeigt Tabelle 2-6.

Bezeichnung	Quelle s	Ziel d	Beschreibung
Declared_In	Programming_Entity	Module	s wird in d deklariert
Import	Module	Module	s importiert Teile von d

Tabelle 2-6: Kanten für Module im Basis-RFG

2.1.4 Erweiterungsmöglichkeiten

Zusätzliche Informationen können im RFG auf zwei Arten abgelegt werden:

1. Es können neue Knoten- bzw. Kantentypen eingeführt werden, wie dies beispielsweise die Dominanzanalyse tut.
2. Bestehende Knoten und Kanten können zusätzliche statische bzw. dynamische Attribute erhalten. Dynamisch hinzugefügte Attribute können zur Zeit allerdings noch nicht persistent gemacht werden.

Für die Arbeit mit RFGs existiert eine umfangreiche Werkzeugsammlung, die in [Eis98] erläutert wird. Hier finden sich auch weitere Informationen über die Implementierung des RFGs. Zusatzinformationen sind auch in [Kos00] und [GirKos98] enthalten.

2.2 Unified Modeling Language

2.2.1 Geschichte der UML

Beginnend Anfang der 80er Jahre bis Mitte der 90er Jahre wurden eine Vielzahl von Methoden für objektorientierte Analyse und Design (OOA&D) vorgeschlagen. 1994 existierten etwa 50 verschiedene Methoden. Jede dieser Methoden hatte Vor- und Nachteile, so daß sich keine wirklich durchsetzen konnte. Neue Methoden waren oft auf älteren aufgebaut und änderten diese in bestimmten Aspekten ab bzw. erweiterten sie. Unter den wichtigen Methoden waren z. B. die objektorientierte Analyse nach *Coad* und *Yourdan* ([CoaYou91]), *Shlaer/ Mellor* ([ShlMel88]), *Object-Oriented Software Engineering* (OOSE) nach *Jacobson* ([Jac92]), *Rumbaughs Object Modeling Technique* (OMT) ([Rum91]) und die *Booch*-Methode ([Boo94]).

1994 begannen *James Rumbaugh* und *Grady Booch* damit, ihre Methoden unter dem Namen *Unified Method* zu kombinieren. Ein Jahr später stieß *Ivar Jacobson* mit seiner Methode zu ihnen. Anfang 1997 veröffentlichten sie Version 1.0 ihrer *Unified Modeling Language*. Unter starker Beteiligung anderer wurde diese weiterentwikkelt. Ende 1997 erklärte die *Object Management Group* (OMG) Version 1.1 zum offiziellen Standard. Die Weiterentwicklung der UML obliegt seitdem einer eigenen OMG Revision Task Force. Inzwischen wurde Version 1.3 der UML veröffentlicht. Auf diese beziehen sich alle im folgenden gegebenen Angaben. Eine im wesentlichen der Berei-

nigung von Fehlern dienende Version 1.4 ist bereits in Arbeit. Größere Veränderungen sind erst von Version 2.0 zu erarbeiten, die derzeit in Vorbereitung ist. (vgl. [Rum99], S. 4 ff.)

2.2.2 Grundlagen und Ziele der UML

Die UML ist eine graphische Modellierungs*sprache* für die Darstellung, Spezifikation, Konstruktion und Dokumentation softwareintensiver Systeme. Sie ermöglicht es, die 'Baupläne' eines Systems auf standardisierte Art und Weise darzustellen. Dabei unterstützt sie konzeptionelle Aspekte, wie z. B. Geschäftsprozesse ebenso wie konkrete, z. B. Klassen in bestimmten objektorientierten Programmiersprachen. (vgl. [Boo99], S. xv) Wie ihr Name (*Unified Modeling* ***Language***) bereits sagt, ist die UML keine Entwurfs*methode*, d.h. sie definiert kein Vorgehensmodell und ist an keinen bestimmten Entwurfsprozeß gebunden, auch wenn inzwischen *Jacobson* et al. einen speziell für die Verwendung mit UML gedachten Prozeß vorgestellt haben ([Jac99]).

Mit Hilfe der UML können verschiedene Informationsarten ausgedrückt werden:

- Informationen auf logischer Ebene, d.h. über
  - die statische Struktur und
  - das dynamische Verhalten eines Systems,
- Informationen auf Implementierungsebene, über
  - die tatsächlich vorhandenen Programmteile und
  - deren Verteilung auf reale Anlagen zur Laufzeit,
- Informationen über die Organisation der Modelle, d.h.
  - in welche Teile (Pakete) die Modellierungsinformationen unterteilt sind,
  - wie diese Teile zusammenhängen um ein konsistentes Bild zu ergeben,
- Informationen über die zur Modellierung nötigen Erweiterungen der UML selbst, d.h.
  - welche zusätzlichen Konstrukte nötig sind und
  - welche Semantik diese aufweisen.

Die UML kann für drei Perspektiven eingesetzt werden (vgl. [FowSco99], S. 51 f.):

1. Ziel der **Konzeptionsperspektive** ist es, in der Analysephase Konzepte und Zusammenhänge des Zielgebiets zu verdeutlichen. Dies erhöht das Verständnis für den modellierten Bereich. Eine direkte Abbildung auf eine Programmiersprache findet nicht statt.
2. Die **Spezifikationsperspektive** zeigt Schnittstellen der Software und ihre Verantwortlichkeiten, nicht die tatsächliche Realisierung.
3. Die **Implementierungsperspektive** konzentriert sich auf die konkrete Realisierung eines Systems.

Teilweise werden diese Perspektiven mit den gleichen Mitteln modelliert. Die UML legt keine semantischen Unterschiede fest. Für das Verständnis ist es aber wichtig zu wissen, welche Perspektive gewählt wurde.

Grob kann die UML in zwei Teile eingeteilt werden (siehe [uml1.3]):

1. Das **UML-Metamodell** legt fest, welche Sprachelemente UML enthält und wie diese zusammengesetzt werden können (abstrakte Syntax). Außerdem wird die Semantik der einzelnen Sprachelemente, bzw. ihrer Kombination definiert. Dieser Teil ist in Kapitel 2 des UML-Standards enthalten.
2. Die **UML-Notation** beschreibt, wie diese Elemente dargestellt werden und welche möglichen Alternativen dabei existieren. Dies wird in Kapitel 3 des UML-Standards behandelt.

Häufig werden diese Teile nicht klar unterschieden und vermischt dargestellt (so in [Boo99], [FowSco99]), was zu Unklarheiten oder Ungenauigkeiten führen kann.

2.2.3 UML-Metamodell

Das UML-Metamodell definiert die zulässigen UML-Sprachelemente und deren Beziehungen zueinander. Es ist selbst in UML geschrieben, ähnlich einem Compiler, der mit sich selbst übersetzt wurde. Da die UML nicht auf die Modellierung bestimmter Softwareklassen eingeschränkt ist, enthält sie eine Vielzahl von Elementen. Um die resultierende Komplexität des Metamodells herabzusetzen, ist es in hierarchische Teile, bzw. Pakete aufgeteilt:

- Das *Foundation Package* enthält unter anderem den Sprachkern (enthalten im *Core Package*) und die Mechanismen zur Erweiterung der UML (enthalten im *Extension Mechanisms Package*). ([uml1.3], S. 2-11 ff.)
- Das *Behavioral Elements Package* gruppiert alle Sprachelemente, die zur Modellierung des Systemverhaltens eingesetzt werden. ([uml1.3], S. 2-85 ff.)
- Das *Model Management Package* beinhaltet alle Sprachelemente für die Organisation von Modellierungsinformationen, z. B. Pakete und Modelle. ([uml1.3] S. 2-171 ff.)

Einen Ausschnitt aus dem UML-Metamodell zeigt Abbildung 2-7.

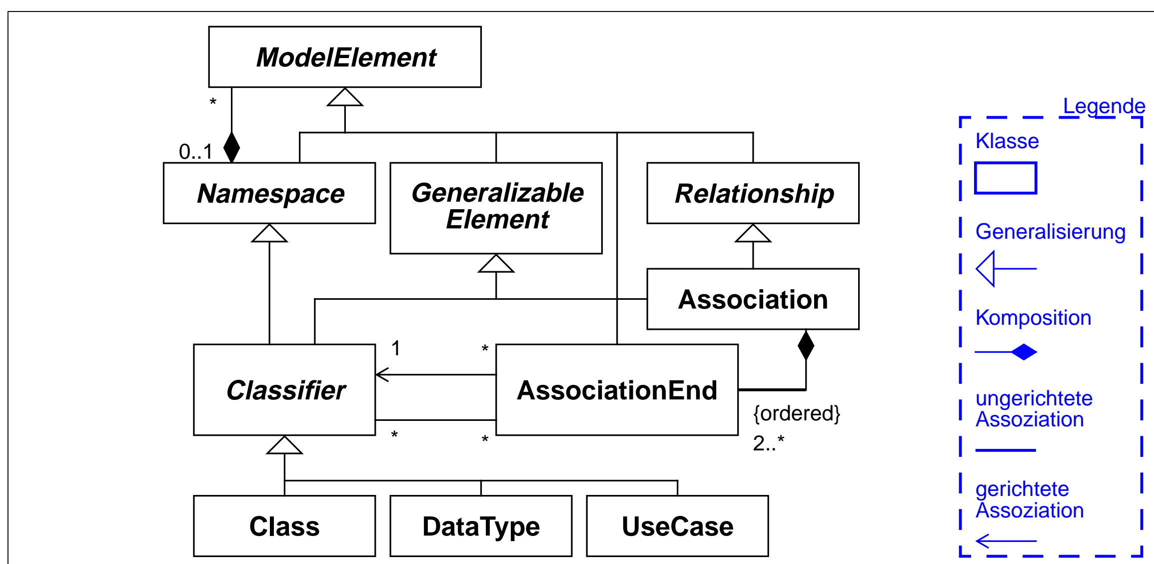


Abbildung 2-7: Ausschnitt aus dem UML-Metamodell

Bei der folgenden kurzen Beschreibung einiger im Metamodell enthaltenen Metaklassen wird nach der deutschen Bezeichnung jeweils die Bezeichnung der entsprechenden Metaklasse im Metamodell aufgeführt. Eine vollständige Beschreibung der im UML-Metamodell enthaltenen Elemente und ihrer Semantik ist in [uml1.3], Kapitel 2 enthalten. Weitere UML-Sprachelemente, die in diesem Bericht benötigt werden, werden jeweils am Ort ihrer Verwendung erklärt.

Modelle

Ein Grundkonzept der UML sind Modelle (`Model`). Ein Softwaresystem wird in ein oder mehrere Modelle abgebildet (oder modelliert). Ein Modell stellt eine semantisch vollständige Abstraktion eines Systems dar. (vgl. [Rum99], S. 342)

Modelle dienen dazu, die Komplexität eines Systems zu vereinfachen, indem sie alle für einen gegebenen Zweck unnötigen Teile eines Systems ausblenden, ohne im gegebenen Kontext falsch zu werden. Bekannte Modelle sind z. B. das Atommodell, das Quantenmodell, etc. Ein Modell beschreibt ein System aus einem bestimmten Blickwinkel und auf einer bestimmten Detaillierungsstufe. Ein Modell enthält typischerweise Modellelemente, die das Modell näher beschreiben.

Modellelemente

Ein Modellelement (`ModelElement`) abstrahiert einen Teil eines Systems. Im Metamodell ist es die direkte oder indirekte Elternklasse für die allermeisten anderen Klassen. Auch Modelle sind spezielle Modellelemente. Andere Beispiele sind Klassifizierer (`Classifier`), Beziehungen (`Relationship`), Objekte (`Object`) oder Aktionen (`Action`).

2.2.4 UML-Notation

Der für die meisten Anwender auffälligere Teil der UML ist ihre Notation. Diese legt fest, wie die im Metamodell definierten Sprachelemente visualisiert werden. Für die meisten Elemente geschieht das über graphische Elemente (Icons, zweidimensionale Symbole, Pfade), die in Diagrammen angeordnet sind.

Diagramme

Ein Diagramm ist eine graphische Darstellung einer Menge von Modellelementen. Die Bedeutung dieser Elemente bzw. ihre Beziehungen zueinander werden von der Art wie sie präsentiert werden nicht berührt. (vgl. [Rum99], S.260 f.) Insbesondere sind Informationen, wie Anordnung, Farbe oder Größe von Elementen in den meisten Fällen in der UML ohne semantische Bedeutung und daher auch nicht im Metamodell enthalten. Eine Ausnahme bilden Sequenzdiagramme, bei denen die zeitliche Abfolge einzelner Interaktionen über deren Position auf der Zeitachse dargestellt wird (siehe Kapitel „Sequenzdiagramme (Sequence Diagrams)“ auf Seite 19). Für den menschlichen Betrachter liefern diese Informationen dagegen teilweise wichtige Hinweise. Beispielsweise legt die örtliche Nähe zwischen zwei Elementen eine Beziehung zwischen ihnen nahe, ihre Farbe kann Klassifizierungshinweise geben (siehe auch [Coa99]). Dies kann in UML derzeit nicht modelliert werden und liegt in der Verantwortung des verwendeten CASE-Werkzeugs. Im Metamodell ist für Darstellungsinformationen eine eigene Klasse `PresentationElement` vorgesehen, die im UML-Standard allerdings bisher nicht näher definiert wurde.

Einzelne Diagramme zeigen bestimmte Sichten oder Aspekte eines Systems bzw. Modells, nicht jedes Detail. Erst einige Diagramme zusammen ergeben ein konsistentes Gesamtbild. Beispielsweise kann ein Diagramm nur das Verhalten einer einzigen Klasse zeigen (siehe Kapitel „Zustandsdiagramme (Statechart Diagrams)“ auf Seite 20) oder den Ablauf eines bestimmten Anwendungsfalls (siehe Kapitel „Sequenzdiagramme (Sequence Diagrams)“ auf Seite 19). Eine in einem Klassendiagramm dargestellte Klasse kann ohne oder nur mit ganz bestimmten Operationen gezeigt werden, obwohl sie mehr besitzt, wenn nur die gezeigten hier wichtig sind. D. h. es ist nicht möglich aus einem einzelnen Diagramm auf alle Eigenschaften eines visualisierten Modellelements zu schließen.

Theoretisch kann ein Diagramm dazu verwendet werden, beliebige Kombinationen von Modellelementen und deren Beziehungen zueinander zu visualisieren. In der Praxis haben sich aber neun Diagrammarten ergeben, die auch im UML-Standard aufgeführt werden. (vgl. [Boo99], S. 24 ff., vgl. [uml1.3]) Sie werden im folgenden kurz mit einigen Notationsbeispielen erläutert. Im weiteren Verlauf dieser Arbeit wird die Kenntnis dieser Diagramme vorausgesetzt und bei in UML-Notation dargestellten Abbildungen auf eine Legende verzichtet. *Fowler* verwendet zusätzlich zu den hier aufgeführten Diagrammen Paketdiagramme, die nur Pakete und deren Beziehungen enthalten. (vgl. [FowSco99], S. 108) Da sich die englischen Bezeichnungen für diese Diagramme inzwischen weit verbreitet haben, werden sie hinter den deutschen Namen jeweils in Klammern mit angegeben. Weiterhin werden auch Metaklassenbezeichnungen in Klammern angegeben.

Klassendiagramme (Class Diagrams)

Ein Klassendiagramm zeigt die *statische* Struktur eines modellierten Systems, bzw. Teile dessen. In ihm werden neben Klassen (`Class`), Schnittstellen (`Interface`) und Paketen (`Package`) auch Beziehungen zwischen diesen Elementen und Instanzen von Klassen, d. h. Objekte (Object) und Instanzen von Assoziationen, sogenannte Verknüpfungen (`Link`) visualisiert. Neben passiven Elementen können auch aktive Elemente, z. B. aktive Klassen, die einen eigenen Kontrollfluß besitzen, visualisiert werden. Elemente, die das *dynamische* Verhalten eines Systems modellieren, werden nicht in Klassendiagrammen dargestellt. Eine Übersicht zeigt Abbildung 2-8. Auf die Darstellung von Instanzen wird bei der Besprechung von Objektdiagrammen im nächsten Abschnitt eingegangen.

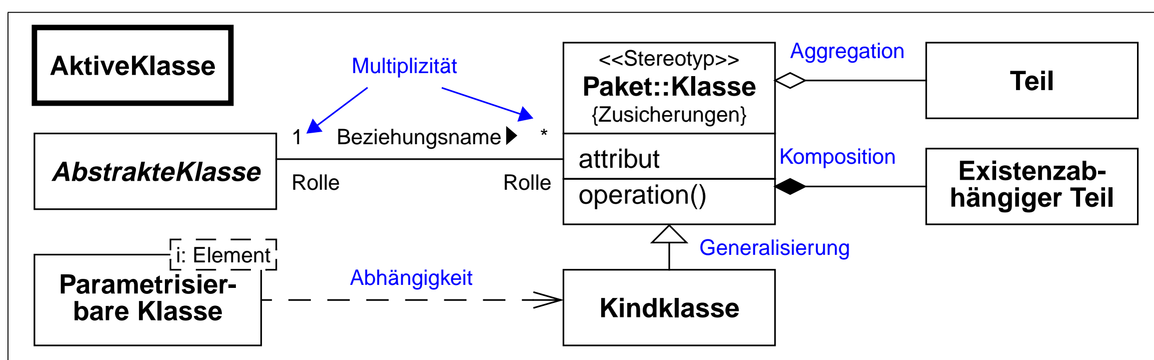


Abbildung 2-8: Klassendiagramm

Klassendiagramme werden sehr häufig verwendet, sind allgemein verbreitet und bieten sehr umfangreiche Darstellungsmöglichkeiten. (vgl. [uml1.3], S. 3-31)

Objektdiagramme (Object Diagrams)

Sehr eng mit Klassendiagrammen verwandt sind Objektdiagramme. Sie zeigen Momentaufnahmen eines Systems zur Laufzeit, ohne jedoch dynamische Aspekte, z. B. wie die dargestellte Situation erreicht wurde, zu verdeutlichen. Sinnvoll sind sie vor allem, um komplizierte Datenstrukturen zu verdeutlichen. Genau betrachtet, sind Objektdiagramme nur eine Teilmenge der Klassendiagramme. Ein Klassendiagramm mit Objekten und ohne Klassen oder anderen Klassifizierern kann als Objektdiagramm angesehen werden. Objekte können mit oder ohne Namen angegeben werden. Wird der Name angegeben, kann alternativ die Klasse, deren Instanz das Objekt ist, weggelassen werden, wie dies Abbildung 2-9 zeigt. (vgl. [uml1.3], S. 3-32)

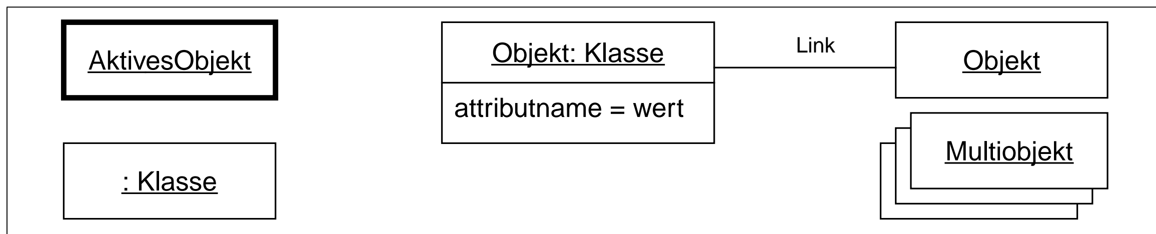


Abbildung 2-9: Objektdiagramm

Anwendungsfalldiagramme (Use Case Diagrams)

Auf einer abstrakteren Ebene angesiedelt sind Anwendungsfalldiagramme wie das in Abbildung 2-10 dargestellte. Sie zeigen die Beziehungen zwischen Anwendungsfällen und Akteuren. Ein Anwendungsfall repräsentiert eine zusammenhängende Menge an Funktionalität des betrachteten Systems, Teilsystems, bzw. der Klasse an der Schnittstelle des Systems zu seiner Umgebung. Die Umgebung wird als eine Menge von Akteuren abgebildet. Ein typisches Beispiel ist die Eingabe neuer Kundendaten (Anwendungsfall) durch einen Sachbearbeiter (Akteur). Anwendungsfalldiagramme werden vor allem in den frühen Phasen der Softwareentwicklung eingesetzt, z. B. bei der Anforderungsdefinition. Eine Diskussion über die Grenzen dieser Diagrammart findet sich in [Oes99b]. (vgl. [uml1.3], S. 3-87 ff.)

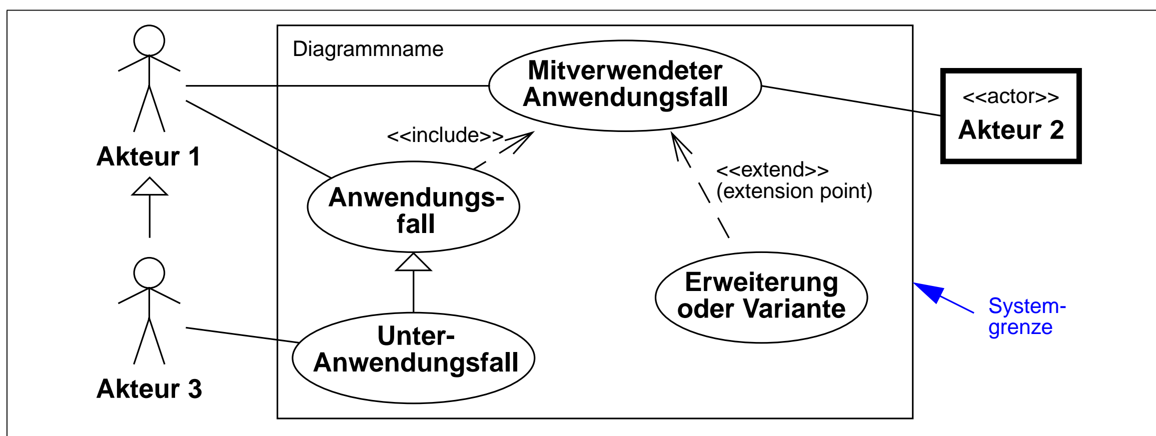


Abbildung 2-10: Anwendungsfalldiagramm

Während die bisher behandelten Diagrammarten verwendet werden, um die logische, statische Struktur eines Systems darzustellen, verdeutlichen die nächsten vier Diagrammarten das dynamische Verhalten eines Systems.

Sequenzdiagramme (Sequence Diagrams)

Sequenzdiagramme zeigen Interaktionen zwischen Mengen von Instanzen. Eine Interaktion besteht aus der zeitlichen Abfolge mehrerer Kommunikationsnachrichten. Die beteiligten Instanzen können Objekte aber auch Akteure sein. Oft werden Sequenzdiagramme daher eingesetzt, um Anwendungsfälle genauer zu spezifizieren. Ein weiterer wichtiger Einsatzzweck ist die Verdeutlichung komplizierter Kommunikationsfolgen zwischen Objekten, z. B. in einem verteilten System, bei denen auch Echtzeitanforderungen eine Rolle spielen. Sequenzdiagramme sind in der Lage, alle

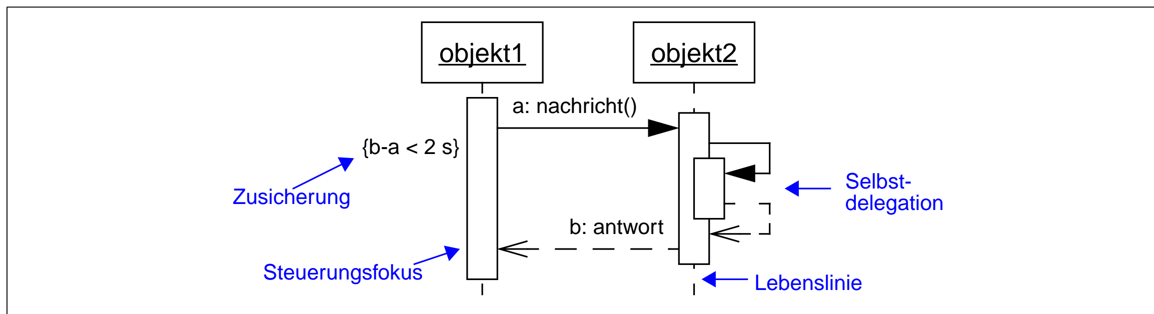


Abbildung 2-11: Sequenzdiagramm

in einer bestimmten Situation zwischen Instanzen möglichen Interaktionen darzustellen. Normalerweise werden verschiedene Interaktionen (z. B. für eine erfolgreiche oder fehlgeschlagene Anfrage) jedoch auf mehrere Diagramme verteilt. Der Hauptfokus bei der Abbildung von Interaktionen liegt bei dieser Diagrammart auf der Verdeutlichung der zeitlichen Abfolge. Abbildung 2-11 zeigt die verwendete Notation. (vgl. [uml1.3], S.3-93 ff.)

Kollaborationsdiagramme (Collaboration Diagrams)

Ebenso wie Sequenzdiagramme zeigen Kollaborationsdiagramme Interaktionen zwischen Instanzen. Daher werden beide Diagrammarten auch unter dem Begriff 'Interaktionsdiagramm' vereint. Tatsächlich sind sie nur zwei verschiedene Visualisierungsmöglichkeiten für die gleiche Information. Das CASE-Werkzeug *Rational Rose* ist sogar in der Lage, die beiden Ansichten auf Knopfdruck ineinander umzuwandeln. Kollaborationsdiagramme betonen die Beziehungen und Verbindungen zwischen den an der Interaktion beteiligten Elementen, während die zeitliche Abfolge der Nachrichten schwieriger ersichtlich ist. Ein Kollaborationsdiagramm ist in Abbildung 2-12 dargestellt. (vgl. [uml1.3], S.3-105 ff.)

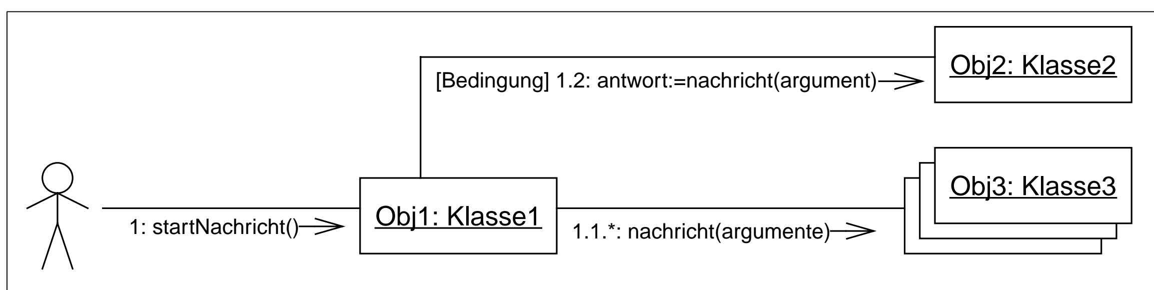


Abbildung 2-12: Kollaborationsdiagramm

Zustandsdiagramme (Statechart Diagrams)

Das Verhalten einzelner Modellelemente, wie z. B. eines Objekts, aber auch einer Interaktion, kann mit Zustandsdiagrammen, wie dem in Abbildung 2-13 gezeigten, dargestellt werden. Sie zeigen erweiterte Zustandsautomaten mit zusammengesetzten und/ oder nebenläufigen Zuständen (State). Zustände können einige Zeit andau-

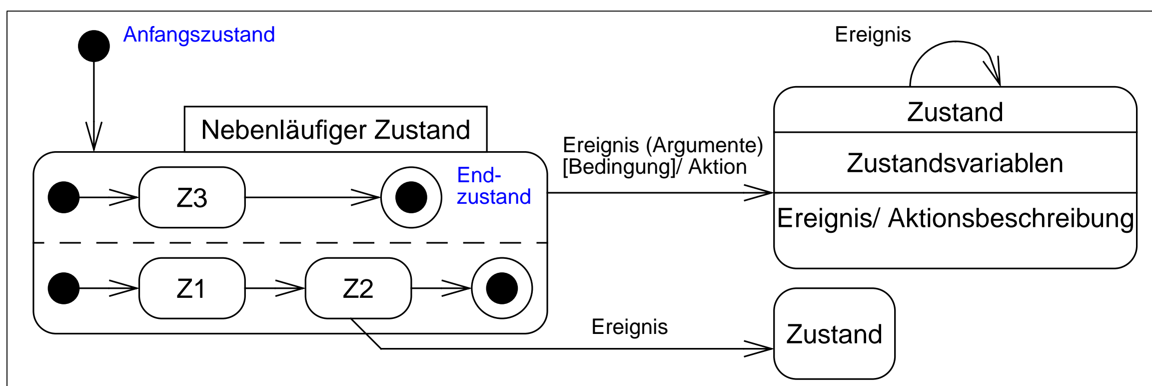


Abbildung 2-13: Zustandsdiagramm

ern oder sofort wieder wechseln. Atomare Tätigkeiten (`Action`) und längere Tätigkeiten (`Activity`) können visualisiert werden. Insgesamt sind die dargestellten Modellelemente sehr zahlreich, was zu teilweise verwirrenden Darstellungen führt. (vgl. [uml1.3], S.3-125 ff.)

Aktivitätsdiagramme (Activity Diagrams)

Sehr viel einfacher verständlich sind Aktivitätsdiagramme, z. B. das in Abbildung 2-14 dargestellte. Sie zeigen spezielle Zustandsautomaten, die sogenann-

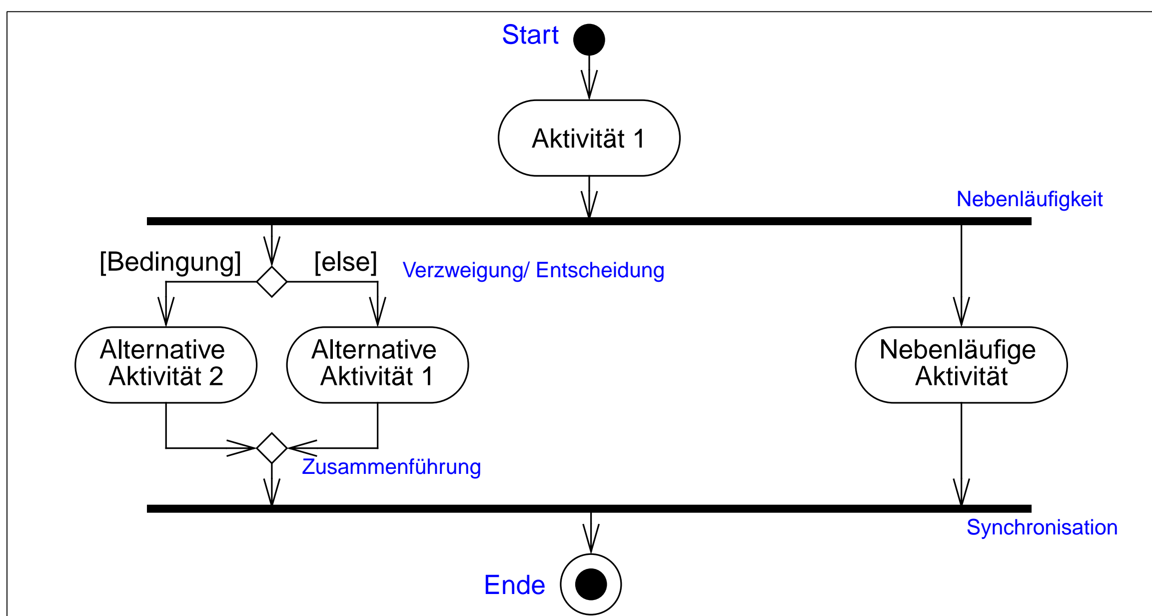


Abbildung 2-14: Aktivitätsdiagramm

ten *activity graphs*. Die in diesen enthaltenen Zustände repräsentieren die Ausführung von im allgemeinen kurzen, bzw. atomaren Aktionen. Ein Übergang findet statt,

sobald die Aktion ausgeführt wurde. Dadurch eignen sich Aktivitätsdiagramme besonders für die Veranschaulichung von Kontrollflüssen, z. B. bei der Geschäftsprozeßmodellierung, oder der Realisierung von Operationen. Sie werden aber auch eingesetzt, um Anwendungsfälle zu spezifizieren (siehe [Oes99b]). (vgl. [uml1.3], S.3-145 ff.)

Die verbleibenden zwei Diagrammarten zeigen die tatsächliche, konkrete Implementierung eines Systems. Sie werden daher auch als Implementierungsdiagramme bezeichnet.

Komponentendiagramme (Component Diagrams)

Komponentendiagramme visualisieren Komponenten (Component) mit ihren Schnittstellen und Abhängigkeiten. Ihre Notation zeigt Abbildung 2-15. Unter einer Komponente versteht die UML einen konkreten Teil einer Systemimplementation, z. B. den ausführbaren Programmtext einer Klasse, ein Programm oder eine Datei. (vgl. [uml1.3], S. 2-29, vgl. [uml1.3], S.3-159 f.)

Abbildung 2-15: Komponentendiagramm

Einsatzdiagramme (Deployment Diagrams)

Um die Struktur eines Systems zur Laufzeit zu zeigen, werden Einsatzdiagramme – wie das in Abbildung 2-16 gezeigte – verwendet. Sie visualisieren Knoten (Node) und Instanzen von Komponenten, Kommunikationsbeziehungen zwischen Knoten und Abhängigkeiten zwischen Komponenten. Die UML definiert einen Knoten als ein physisch existentes Objekt, meist mit eigenem Speicher und oftmals einer gewissen Rechenkapazität, auf dem Komponenten ausgeführt bzw. abgelegt werden können. (vgl. [uml1.3], S. 2-41) Neben Computern können darunter, z. B. bei der Modellierung von Geschäftsprozessen, auch Mitarbeiter oder Organisationseinheiten eines Unternehmens verstanden werden. Besonders geeignet sind Einsatzdiagramme beim Entwurf von verteilt ablaufenden Applikationen, wie mobilen Softwareagentensystemen. (vgl. [uml1.3], S.3-161 f.)

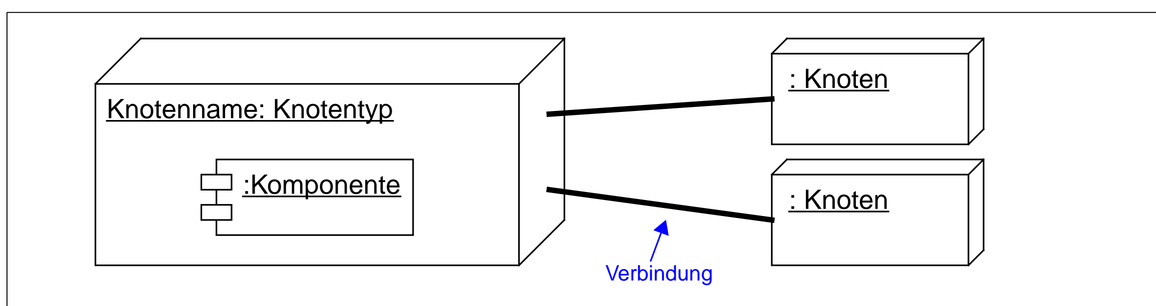


Abbildung 2-16: Einsatzdiagramm

2.2.5 Erweiterungsmöglichkeiten

Sollten die von der UML vorgegebenen Sprachmittel für eine Modellierung nicht ausreichen, existieren grundsätzlich zwei Möglichkeiten diese zu erweitern:

1. Eine Erweiterung des UML-Metamodells, z. B. indem neue Unterklassen von existierenden Modellelementen hinzugefügt werden, erlaubt eine einfache und mächtige Anpassung. Da dadurch allerdings ein zum Standard inkompatibler Dialekt entsteht, vorhandene CASE-Werkzeuge die Erweiterungen also nicht verarbeiten können, sollte diese Möglichkeit im allgemeinen verworfen werden.
2. Die UML definiert verschiedene Mechanismen zur definierten und standardkonformen Erweiterung des Sprachumfangs. Alle Modellelemente können mit beliebigen zusätzlichen Werten (TaggedValue) attributiert werden, z. B. den Namen der Autoren. Modellelemente können mit zusätzlichen Einschränkungen (Constraint) versehen werden, die ihre Semantik anpassen. Hierzu wurde sogar eine eigene Sprache in die UML mit aufgenommen, die sogenannte *Object Constraint Language* (OCL). Stereotypen gruppieren benutzerdefinierte Werte und Einschränkungen. Außerdem definieren sie neue Notationen für die erweiterten Klassen. Jedem Modellelement kann ein Stereotyp zugewiesen werden. Damit ist es oft möglich, neue Sprachelemente von vorhandenen abzuleiten.

Eine gute Einführung in die Arbeit mit UML geben [FowSco99] und [Boo99]. Als Nachschlagewerk zu empfehlen ist [Rum99]. Für genauere Informationen, insbesondere über das UML-Metamodell, empfiehlt sich [uml1.3], besonders die Kapitel 2 und 3.

Rahmenkonzeption

Kapitel 3

Das Ziel dieser Arbeit kann grob in drei Unterpunkte aufgeteilt werden:

1. Die in Form eines RFGs vorliegenden architektonischen Informationen sollen in eine UML-Darstellung transformiert werden. Hierzu ist es nötig, semantisch möglichst äquivalente UML-Sprachmittel zu finden.
2. Das erzeugte Modell soll in einem UML-Werkzeug dargestellt werden und der Benutzer soll gewisse Änderungen vornehmen können. Dies erfordert eine Integration der *Bauhaus*-Werkzeuge, insbesondere des RFGs, mit einem existierenden UML-Werkzeug.
3. Um die erfolgten Änderungen zu sichern, soll zudem eine Rücktransformation der UML-Daten in einen RFG möglich sein. Dazu ist zu klären, wie diese aussehen kann, was zum größten Teil von der ursprünglichen Transformation abhängig ist. Zudem muß festgelegt werden, welche UML-Sprachelemente, bzw. welche Benutzeränderungen, nicht zurück in den RFG transformiert werden können.

Es wäre möglich, diese Teile getrennt zu behandeln. Nach Einarbeitung in das Thema erschien es jedoch sinnvoller, ein anderes Vorgehen zu wählen, bei dem eine Aufteilung der Arbeiten nach den Teilbereichen der Architekturerkennung vorgenommen wird. Für jeden Teilbereich werden jeweils alle oben genannten Punkte behandelt. Dies hat den Vorteil, daß der Leser selektiv nur die Kapitel, die für ihn interessante Teilbereiche behandeln, lesen kann. Im Rest dieses Kapitels wird hierfür ein Rahmenkonzept erarbeitet und vorgeschlagen, das die in der obigen Aufzählung genannten Tätigkeiten für eine (initial leere) Architekturbeschreibung ermöglicht.

Hierzu werden die prinzipiellen Anforderungen an den verwendeten RFG sowie die zu tätigenden Transformationen untersucht, bevor die prinzipielle Modellierung entworfen wird. Da eine detaillierte Modellierung von der Integration mit dem UML-Werkzeug abhängt, werden danach zunächst die vorliegenden Integrationsmöglichkeiten diskutiert und eine der Möglichkeiten ausgewählt. Schließlich wird eine detaillierte Transformation einer als RFG vorliegenden Architekturbeschreibung nach UML, die erlaubten Manipulationen durch einen Benutzer sowie eine geeignete Rücktransformation entworfen.

In den weiteren Kapiteln wird dieser Entwurf jeweils um die in den verschiedenen Teilprojekten von Bauhaus ermittelten Informationen erweitert, so daß zuletzt eine Gesamtlösung vorliegt, die als Abschluß der Arbeit prototypisch implementiert wird.

3.1 Anforderungen

Es kann zwischen Anforderungen an den als Datenquelle verwendeten RFG und der zu erstellenden Modellierung unterschieden werden.

3.1.1 Anforderungen an den Resource Flow Graphen

Um als Ausgangspunkt der Transformationen dienen zu können, muß ein RFG bzw. die in ihm vorliegende Architekturbeschreibung gewissen Anforderungen gerecht werden.

Im Rahmen dieser Arbeit wird davon ausgegangen, daß die Erkennung der zu modellierenden Architektur (siehe auch die dieser Arbeit zugrunde gelegte Definition einer Architektur im Kapitel „Grundlegende Informationen“ auf Seite 5) zu einem hinreichenden Teil abgeschlossen wurde und die Ergebnisse im RFG gespeichert wurden. Die für die Transformation in UML benötigten Informationen sollten also bereits im RFG enthalten sein. Die Erkennung neuer Informationen ist nicht primäres Ziel dieser Arbeit. Die im Basis-RFG enthaltenen Informationen werden noch nicht als vollständige Architekturbeschreibung angesehen, obwohl sie einen sehr wichtiger Teil dieser Informationen darstellen. Ihre Modellierung erfolgt im Kapitel „Modellierung atomarer Komponenten“ auf Seite 41.

Eine weitere Vorbedingung für eine semantisch und syntaktisch korrekte Modellierung einer Architekturbeschreibung ist, daß die verarbeitete Architekturbeschreibung im RFG selbst semantisch und syntaktisch korrekt war. Während die meisten syntaktischen Fehler in der Architekturbeschreibung bei der Modellierung einfach entdeckt werden können, sind semantische Fehler nur sehr schwer, bzw. im allgemeinen gar nicht, automatisch erkennbar. Fehlen z. B. in einem RFG einige Unterprogramme, ist die Architekturbeschreibung fehlerhaft. Im Rahmen dieser Arbeit werden keine Verfahren untersucht, mit denen semantische Fehler entdeckt und korrigiert werden können (beispielsweise mittels des Vergleichs mit der IML-Darstellung). Enthält ein RFG keine korrekte Architekturbeschreibung in Form geeigneter Knoten und Kanten, sind die Ergebnisse einer Transformation dieses RFGs in UML prinzipiell undefiniert.

Dies gilt insbesondere für RFGs, in denen mehrere Architekturen vermischt vorliegen. Neben der Erkennung der Ist-Architektur verfolgt das Projekt Bauhaus das Ziel, die Soll-Architektur eines SW-Systems zu ermitteln. Während die Ist-Architektur eines SW-Systems angibt, wie das System zur Zeit tatsächlich aufgebaut ist, zeigt die Soll-Architektur, wie ein System gegebenenfalls zu einem späteren Zeitpunkt aufgebaut sein soll, bzw. wie es eigentlich gedacht war. Beide Architekturen können sich in Details und grundlegenden Punkten unterscheiden. Als Beispiel sei ein abstrakter Datentyp genannt, auf den konzeptionell nur über definierte Zugriffsoperationen zugegriffen werden darf. In der Soll-Architektur wäre es daher sinnvoll, Teilkomponenten dieses Typs als privat zu kennzeichnen. In der tatsächlich vorliegenden Architektur kann diese Kapselung allerdings verletzt sein, indem ein externes Unterprogramm direkt den Wert einiger Teilkomponenten setzt. In der Ist-Architektur sind diese Teilkomponenten daher als öffentlich zu deklarieren.

Sowohl die Ist- als auch die Soll-Architektur eines SW-Systems wird als RFG modelliert, teilweise sind beide im selben RFG enthalten. Angestrebt ist eine klare Trennung dieser Architekturen, z. B. mit Hilfe von Views (siehe Kapitel „Definition und Aufbau“ auf Seite 5). Derzeit ist dies jedoch noch nicht der Fall, d. h. es kann vorkommen, daß in einem RFG Teile der Ist- und der Soll-Architektur vermischt enthalten sind. Dies stellt keine korrekte Architekturbeschreibung dar und muß daher auch nicht korrekt verarbeitet werden. In Fällen, in denen eine einfache Korrektur möglich ist, sollte diese erfolgen. In allen anderen Fällen werden falsche Ergebnisse die Folge sein. Eine Korrektur ist nicht Aufgabe dieser Arbeit.

Der Grund für die fehlende Trennung der verschiedenen Architekturen liegt unter anderem darin, daß die Ist-Architektur bisher in verschiedene Views verteilt vorliegt. So sind die grundlegenden Informationen des Basis-RFGs, die ebenfalls zur Architekturbeschreibung zu zählen sind, z. B. Member-Knoten oder Call-Kanten, in der Base-View enthalten, während höhere Konstrukte, wie atomare Komponenten, die im Kapitel „Modellierung atomarer Komponenten“ auf Seite 41 besprochen werden, Teil der User-View sind. Eine entsprechende Festlegung für die Soll-Architektur existiert noch nicht. Der dieser Arbeit zugrundegelegte Ansatz zur Trennung basiert darauf, daß alle zur Beschreibung einer Architektur verwendeten Knoten und Kanten in einer gesonderten View vorliegen. Diese View wird im Rahmen dieser Arbeit als Architektur-View bezeichnet. Unter welchem tatsächlichen Namen sie im RFG abgelegt wird, ist prinzipiell nicht von Bedeutung, solange den Werkzeugen der Name der zu behandelnden View bekannt ist. Dies kann z. B. über eine statische Festlegung oder die Übergabe des Namens als Parameter gewährleistet werden. Die hier gewählte Variante wird im Kapitel „Namengebung der Architektur-View“ auf Seite 88 behandelt.

Die Architektur-View resultiert aus einer Verschmelzung von Base- und User-View eines RFGs. Sie kann als Ausgangsbasis aller weiteren Betrachtungen angesehen werden, d. h. alle in ihr enthaltenen Knoten und Kanten werden transformiert. Alle in anderen Views enthaltenen RFG-Teile werden dagegen ignoriert. Auf diese Weise ist es sehr einfach möglich, den in dieser Arbeit vorgeschlagenen Werkzeugen mitzuteilen, welche von eventuell mehreren in einem RFG vorliegenden Architekturbeschreibungen transformiert werden soll. Ob es sich bei dieser Architektur um die Soll- oder Ist-Architektur eines SW-Systems handelt, ist nicht von Bedeutung. Erweiternd ist es später eventuell möglich, mehrere Views und damit mehrere Architekturen abzubilden und sie als Ganzes oder einzelne ihrer Teile miteinander (z. B. mit Hilfe von Kanten in einer weiteren View) in Beziehung zu setzen, wie dies im Kapitel „Prinzipielle Modellierung einer Architekturbeschreibung“ auf Seite 30 angedacht ist.

Im weiteren Verlauf dieser Arbeit wird davon ausgegangen, daß der untersuchte RFG in genau einer Architektur-View genau eine korrekte, konsistente, hinreichend vollständig erkannte Architekturbeschreibung eines SW-Systems enthält.

3.1.2 Anforderungen an die Modellierung

Neben den genannten Anforderungen an den RFG sind weitere Anforderungen an die durchzuführende Modellierung selbst zu stellen.

- **Korrektheit der Modellierung**: die Semantik der im RFG enthaltenen Informationen muß soweit wie möglich erhalten bleiben, d. h. die Architektur muß korrekt modelliert werden. Dazu zählt auch, daß eine Architektur sich nicht allein dadurch ändert, daß ihre Darstellung im RFG in die UML transformiert und – ohne daß der Benutzer Änderungen vorgenommen hätte – in einen RFG zurücktransformiert wurde. Es dürfen demzufolge in einem solchen Fall z. B. bei der Rücktransformation keine neuen Knoten in den RFG eingefügt werden, die die Semantik der enthaltenen Architektur abändern würden, wenn der Benutzer dies nicht veranlaßt hat. Bedingung für eine korrekte Modellierung ist – wie bereits erwähnt – eine korrekte Datenbasis in Form einer Architektur-View in einem RFG. Während der Durchführung dieser Diplomarbeit wurde immer klarer, daß eine semantisch vollkommen äquivalente Modellierung nicht machbar sein wird, da sich der RFG und die UML in ihrer Ausrichtung und den verwendeten Mitteln zu stark unterscheiden.
- **Korrekte und standardkonforme Verwendung der UML**: es ist auf die korrekte Verwendung der UML zu achten, da die resultierende UML-Modellierung in UML-Werkzeugen verwendbar sein soll. Dafür ist eine richtige und standardkonforme Anwendung der UML notwendige Voraussetzung.
- **Korrekte Abbildung von Benutzermanipulationen**: Änderungen in der UML-Ansicht einer Architekturbeschreibung müssen zu semantisch passenden Änderungen in der Architektur-View im RFG führen. Beispielsweise muß die Umbenennung eines Elements in UML zur Umbenennung des entsprechenden Knotens in der View führen. Diese Anforderung kann – zumindest bisher – nicht für alle prinzipiell möglichen Manipulationen durch den Benutzer im UML-Werkzeug erfüllt werden. Stattdessen muß eine Festlegung auf zulässige Benutzerinteraktionen erfolgen.
- **Verständlichkeit der Modellierung**: die Semantik einer Modellierung soll sich dem Betrachter so schnell und einfach wie möglich erschließen, idealerweise auch dann, wenn er nicht mit UML oder OOA&D vertraut ist. Auf der anderen Seite sollte ein in der UML erfahrener Betrachter soweit möglich nicht durch ungewöhnliche bzw. dem Geist der UML widersprechende Konstrukte verwirrt werden.
- **Wahrung des Bezugs zum Quelltext**: da bei Wartungsarbeiten bereits eine Implementation vorliegt, ist darauf zu achten, daß es dem Benutzer jederzeit möglich sein muß, von den Beschreibungen auf abstrakterer Ebene wieder den Bezug zur konkreten Realisierung herzustellen.
- **Verarbeitung großer Systeme**: die modellierten SW-Systeme sind oft relativ umfangreich, da erst bei großen Systemen die Vorteile einer halb-automatischen Architekturerkennung voll zum Tragen kommen. Es muß möglich sein, Systeme zu verarbeiten, die aus mehreren tausend Knoten und einem mehrfachen an Kanten bestehen.

Nicht gefordert wird:

- die Erzeugung eines UML-Modells, aus dem mit Hilfe eines Quelltextgenerators der Quelltext eines ablauffähigen Programms erstellt werden kann,
- die Ermittlung *fehlender* Informationen, soweit dies nicht einfach möglich ist,
- eine *vollständige* Abbildung *aller* im RFG enthaltenen Informationen auf UML-Konstrukte sowie eine vollständige Rücktransformation eines UML-Modells in einen RFG. Bei beiden Abbildungen werden gewisse Informationsverluste nicht sinnvoll vermeidbar sein. Beispielsweise existiert derzeit keine sinnvolle Möglichkeit, UML-Verteilungsdiagramme auf einen RFG abzubilden.

Ein prinzipielles Problem dieser Arbeit ist, daß ein RFG und die UML zum Teil sehr verschiedene Ausdrucksmittel vorsehen. Die Architektur eines SW-Systems wird durch die in ihm vorliegenden Elemente und deren Beziehungen und Interaktionen zueinander definiert. Die Elemente einer in einem RFG vorliegenden Architektur sind auf niedriger Ebene Konstrukte wie Typen, Unterprogramme und globale Variablen, auf höherer Ebene atomare Komponenten und Subsysteme. In UML werden dagegen Architekturen mit Konstrukten wie Klassen, Objekten, Vererbung und Polymorphie sowie Paketen modelliert. Eine direkte Abbildung zwischen den Darstellungen ist nicht möglich, ohne z. B. an der UML so große Änderungen der Semantik vornehmen zu müssen, daß die resultierende Modellierung nicht mehr als standardkonformes UML-Modell angesehen werden kann. Hier müssen Kompromisse eingegangen werden.

3.2 Grobentwurf

Die zu transformierende Architekturbeschreibung liegt in Form einer Architektur-View eines RFGs vor. Es ist daher nicht nötig, den kompletten RFG mit allen in ihm enthaltenen Views abzubilden. Nur die Architektur-View wird abgebildet. Die Information, in welchen Views einzelne Knoten und Kanten zudem enthalten sind, ist für die Modellierung der Architekturbeschreibung nicht relevant und wird deshalb verworfen.

Mehrfach werden in dieser Ausarbeitung Ausschnitte aus einem RFG gezeigt, um verschiedene Sachverhalte zu verdeutlichen, so z. B. Abbildung 2-1 auf Seite 6. Hierzu werden Objektdiagramme verwendet. Grundsätzlich stellt dies bereits eine mögliche Modellierung eines RFGs mit UML-Sprachmitteln dar. Allerdings entspricht sie eher einer unmittelbaren Darstellung der Graphenstruktur mit UML-Notation als einer sinnvollen Modellierung der in dieser Struktur enthaltenen architektonischen Informationen. Aus diesem Grund wurde diese Möglichkeit nicht weiter verfolgt.

3.2.1 Grundlegende Festlegungen

Vor Beginn der eigentlichen Untersuchungen müssen einige grundlegende Festlegungen getroffen werden.

Verwendete UML-Version

Zunächst sollte eine Festlegung auf eine spezifische UML-Version erfolgen, die als Ziel der Transformationen dient. Gewählt wurde die derzeit aktuelle Version 1.3. Alle im weiteren Verlauf der Arbeit untersuchten UML-Werkzeuge verwenden diese Version der UML zur Anzeige von Diagrammen oder werden dies in nächster Zukunft tun. Aufgrund der im Kapitel „Werkzeugunabhängige Integration mittels XML Metadata Interchange“ auf Seite 35 genannten Schwierigkeiten, mußte für die prototypische Realisierung der Integration mit einem UML-Werkzeug auf UML 1.1 zurückgegriffen werden. Sobald die verwendeten UML-Werkzeuge entsprechend erweitert wurden, kann auch hier die Version 1.3 der UML Verwendung finden. Daher erscheint es sinnvoll, bei der Untersuchung der möglichen Transformationen zwischen RFG und UML von Beginn an UML 1.3 einzusetzen und die prototypische Realisierung zunächst auf UML 1.1 aufzubauen, mit der Option auf UML 1.3 zu wechseln.

Die neueren Versionen 1.4 und 2.0 wurden noch nicht vollständig definiert und werden bisher von keinem Werkzeug unterstützt. Sie werden deshalb nicht weiter betrachtet.

Nicht äquivalent transformierbare Informationen

Um eine möglichst umfassende und semantisch äquivalente Transformation der in einem RFG enthaltenen Informationen zu erreichen, wird es nötig sein, Erweiterungen an der UML vorzunehmen. Da eine standardkonforme Verwendung der UML gefordert ist, werden diese Erweiterungen ausschließlich mit Hilfe der in der UML hierfür definierten Mechanismen, wie z. B. Stereotypen vorgenommen. Sollten diese Mechanismen nicht ausreichen, um eine semantisch äquivalente Modellierung der Information in UML zu erlauben, sind verschiedene Ansätze möglich:

1. Die betreffenden Informationen können ignoriert werden. Dies führt zum Verlust von Information, garantiert aber eine korrekte Modellierung, solange es nicht zu Interferenzen kommt.
2. Die die Semantik der Information am besten erhaltende Transformation kann gewählt werden. Hierdurch bleiben Informationen erhalten, ihre genaue Bedeutung kann aber unter Umständen inakzeptabel verfälscht werden.

Im folgenden wird die zweite Möglichkeit eingesetzt, soweit die entstehenden Abweichungen in der Semantik vertretbar sind. Ist dies nicht der Fall, wird die entsprechende Information ignoriert.

Gewählte UML-Perspektive

Im Kapitel „Grundlagen und Ziele der UML“ auf Seite 14 werden drei Perspektiven erläutert, für welche die UML eingesetzt werden kann. Dies sind die Konzeptions-, die Spezifikations- und die Implementierungsperspektive. Eine Modellierung in UML muß – je nachdem, für welche Perspektive sie erstellt wurde – anders interpretiert werden. Daher ist es nötig zu definieren, welche Perspektive oder Betrachtungsebene den in dieser Arbeit erläuterten Modellierungen zu Grunde liegt.

Grundsätzlich ist keine der angegebenen Perspektiven ohne Einschränkung geeignet. Die Implementierungsperspektive erscheint als passendste Alternative, da sie den direktesten Bezug zum Quelltext bietet. Sie stellt als einzige dar, wie ein System tatsächlich implementiert ist. Allerdings wird in der UML im allgemeinen davon ausgegangen, daß eine Implementierung in einer objektorientierten Programmiersprache erfolgt, was im Projekt *Bauhaus* bisher nicht der Fall ist. Wird die Spezifikationsperspektive gewählt, können im RFG vorliegende Informationen über die Realisierung, wie z. B. welche Unterprogramme auf eine gegebene Variable zugreifen, nicht modelliert werden, da in dieser Perspektive nur die Schnittstellen der Softwarekomponenten abgebildet werden. Die interne Architektur der Komponenten wird bewußt ignoriert. Die in der Analysephase verwendete Konzeptionsperspektive ist noch weniger geeignet. Sie zeigt die Architektur des Zielbereichs des SW-Systems, nicht die des SW-Systems selbst. Diese Informationen liegen im RFG nicht vor.

Aus diesen Gründen wird grundsätzlich die Implementierungsperspektive gewählt, allerdings mit dem Hinweis, daß die tatsächliche Implementierung des SW-Systems von dieser Darstellung abweichen kann. Üblicherweise ist das auch der Fall.

Visualisierung der UML-Modellierung

Eine wichtige Schwierigkeit entsteht dadurch, daß die UML vor allem für den *manuellen* Einsatz, z. B. zum Entwurf oder der Dokumentation eines Systems durch mehrere Benutzer konzipiert wurde. Es ist sehr effizient möglich, eine angedachte Architektur zu ‘skizzieren’. Für eine in allen Punkten exakte und vollständige Definition dieser Architektur ist die UML dagegen wenig geeignet. Zum einen resultiert dies aus der Tatsache, daß die Semantik der UML-Sprachmittel nicht formal sondern nur natürlichsprachlich definiert ist, zum anderen aus der Tatsache, daß mit der UML erstellte graphische Darstellungen relativ schnell sehr unübersichtlich werden, wenn Details gezeigt werden. Ein Klassendiagramm, das beispielsweise zwanzig Klassen mit all ihren Details zeigt, wird für einen Benutzer nur schwer überschaubar sein. Aus diesem Grund zeigen UML-Diagramme immer nur bestimmte Aspekte oder Teile eines Systems und zusammengehörige Systemteile werden in Paketen gruppiert. Wie diese Teile gewählt werden, ist Sache des Benutzers. Dies stellt die eigentliche Schwierigkeit bei der Arbeit mit der UML dar.

Mit den derzeit im RFG vorliegenden Informationen, ist eine vollständige *automatische* Auswahl der in Diagrammen gemeinsam zu visualisierenden Modellelemente und die Festlegung, welche Details dieser Elemente jeweils gezeigt werden nicht machbar. Für eine sinnvoll verwendbare Darstellung der Informationen ist eine solche Einteilung aber unabdingbar. Eine detaillierte Untersuchung geht über den Rahmen dieser Arbeit hinaus und könnte Gegenstand weiterer Arbeiten sein. Daher wird die Verantwortung für diese Festlegungen bisher dem Benutzer übertragen. Er muß manuell entscheiden, welche Modellelemente in welchen Diagrammen mit welchen Details dargestellt werden sollen. Teilweise kann ihn dabei das gewählte UML-Werkzeug unterstützen.

Die in dieser Arbeit diskutierte Modellierung mit UML beschränkt sich daher auf die zugrundeliegende UML-Modellierung. Hinweise, wie diese sinnvoll visualisiert werden kann, werden nur gegeben, wenn sich aus der Modellierung eine solche Visualisierung ergibt, wie dies z. B. bei Subsystemen der Fall ist (siehe Kapitel „Modellierung von Subsystemen“ auf Seite 67). Um dem Benutzer eine wiederholte

Eingabe dieser Informationen zu ersparen, sollten sie an geeigneter Stelle, beispielsweise im RFG in Form spezieller Views oder Knoten, gespeichert werden. Dies gilt auch für Layoutinformationen, wie die Positionierung einzelner Elemente in einem Diagramm. Zudem wäre es denkbar, diese Informationen zum Teil für die weitere Architekturerkennung einzusetzen. So könnte das Wissen, welche Elemente vom Benutzer gemeinsam visualisiert wurden, Hinweise auf eventuelle Zusammenhänge zwischen diesen Elementen liefern. Derzeit ist eine Speicherung dieser Informationen im RFG leider noch nicht möglich (siehe „Werkzeugunabhängige Integration mittels XML Metadata Interchange“ auf Seite 35), weshalb dieser Problembereich nicht näher untersucht wird. Er bleibt weiteren Arbeiten vorbehalten.

3.2.2 Prinzipielle Modellierung einer Architekturbeschreibung

Für die prinzipielle Modellierung einer Architekturbeschreibung werden UML-Modelle verwendet.

UML-Modelle

Wie im Kapitel „Modelle“ auf Seite 16 beschrieben, stellt ein Modell in UML eine semantisch vollständige Abstraktion eines SW-Systems dar. In gewissem Sinne gilt dies auch für die Architektur eines SW-Systems, bei der je nach Betrachtungsebene verschiedene Abstraktionsebenen denkbar sind. Daher sind UML-Modelle gut geeignet, um bei der Modellierung einer Architekturbeschreibung als umschließendes Rahmenelement eingesetzt zu werden. Die Elemente, die eine Architektur festlegen, sind im UML-Modell enthalten. Erweiternd wäre es auch denkbar, eine Architekturbeschreibung auf verschiedene Modelle mit verschiedenen Abstraktionsebenen abzubilden, vorteilhafter wäre es aber, dafür verschiedene Architekturbeschreibungen in jeweils getrennten Views anzubieten. Diese könnten in getrennte UML-Modelle transformiert werden, zwischen denen Abhängigkeiten definiert werden, wie dies Abbildung 3-1 zeigt. Analog hierzu könnten beispielsweise Ist- und Soll-Architektur

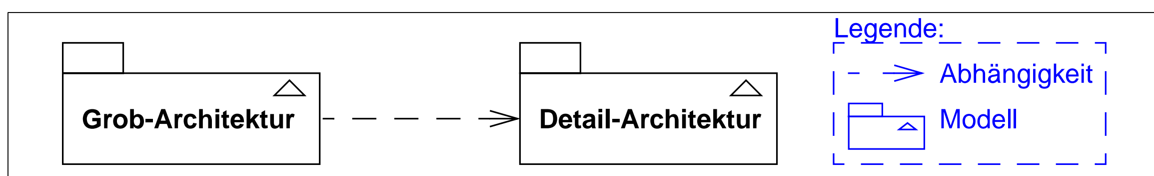


Abbildung 3-1: Modellierung mehrerer Architekturen eines SW-Systems

eines SW-Systems über mehrere Modelle abgebildet werden. Im weiteren Verlauf dieser Arbeit wird dieser Fall nicht näher behandelt, sondern ausschließlich eine einzelne Architekturbeschreibung als Modell abgebildet. Enthielt die Architekturbeschreibung keine Informationen, war die entsprechende View also leer, ergibt sich ein leeres UML-Modell.

UML-Diagramme

Die zweite Möglichkeit, in der UML eine bestimmte Sicht auf ein SW-System zu definieren, stellen Diagramme dar. Diese werden aber nur für die graphische Präsentation von Modellteilen genutzt und sind nicht in der Lage, die in ihnen angezeigten

Elemente zu definieren, d. h. ein Diagramm *enthält* keine Elemente, es *visualisiert* sie nur. Für die Modellierung einer Architekturbeschreibung sind sie daher primär nicht geeignet, sehr wohl jedoch für deren Präsentation in einem UML-Werkzeug.

Das Ziel der Arbeit ist demzufolge die Modellierung einer korrekt ermittelten, in Form einer speziellen Architektur-View eines RFGs vorliegenden Architektur als UML-Modell und die Darstellung dieses Modells mit UML-Diagrammen in einem geeigneten UML-Werkzeug. Die Ermittlung neuer Informationen über das modellierte SW-System ist nicht primäres Ziel der Arbeit. Allerdings sollten Änderungen, die der Benutzer am UML-Modell vornimmt, nicht verloren gehen, wenn der RFG relativ einfach entsprechend erweitert werden kann. Engen Einfluß auf die konkrete Gestaltung dieser Teilbereiche hat die gewählte Integration mit dem UML-Werkzeug.

3.2.3 Integration mit einem UML-Werkzeug

Um eine mit UML modellierte Architektur darstellen zu können, muß eine geeignete Visualisierungskomponente eingesetzt werden. Im Rahmen dieser Arbeit sollte hierfür ein existierendes UML-Werkzeug verwendet werden, das mit *Bauhaus* integriert werden muß. Prinzipiell können nach der Integrationsdichte zwei Möglichkeiten unterschieden werden.

Bei der **engen Integration** werden UML-Werkzeug und RFG direkt aneinander gekoppelt. Jede Änderung im RFG wirkt sich sofort auf das dargestellte UML-Modell aus. Jede Änderung des Modells bewirkt eine unmittelbare Aktualisierung des RFGs. Dadurch sind beide Sichten stets konsistent zueinander. Insbesondere ist es möglich, Manipulationen im UML-Werkzeug nur dann zuzulassen, wenn eine Darstellung im RFG möglich ist, bzw. eventuell zusätzlich benötigte Informationen sofort abzufragen. Dadurch ist es sehr einfach machbar, nicht zulässige Änderungen durch den Benutzer zu verhindern und eine korrekte Rücktransformation in den RFG zu gewährleisten. Dies stellt hohe Anforderungen an das verwendete UML-Werkzeug, das z. B. Aktionen des Benutzers sofort an die zu erstellende Erweiterung melden muß, so daß entsprechend reagiert und eine Interaktion gegebenenfalls abgewiesen werden kann. Abbildung 3-2 zeigt ein vereinfachtes Beispiel. Ein Benutzer öffnet in einem UML-

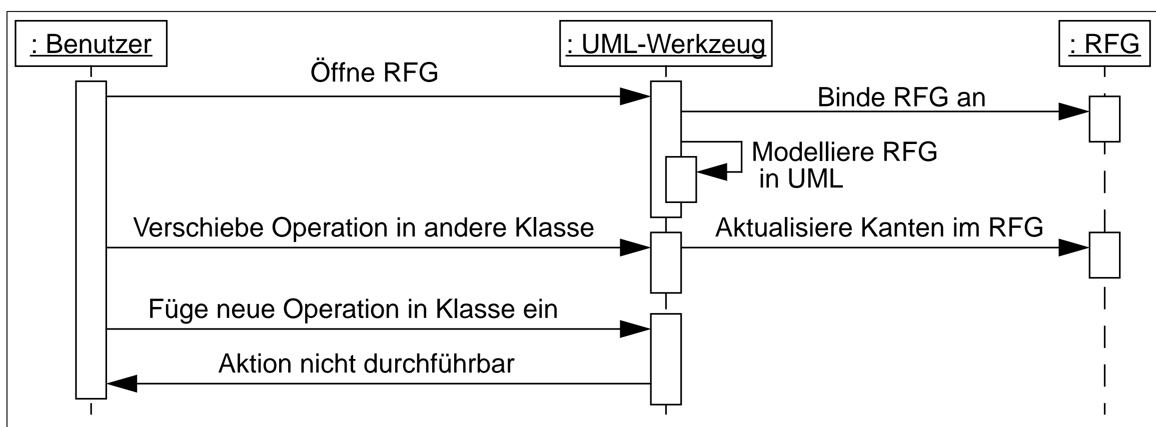


Abbildung 3-2: Anwendungsfall bei enger Integration

Werkzeug einen RFG. Er betrachtet die daraufhin erstellte UML-Modellierung und gibt die Anweisung, eine Operation in eine andere Klasse zu verschieben. Dies führt

zu einer sofortigen Aktualisierung des RFGs. Danach versucht der Benutzer, eine neue Operation einzufügen. Diese Aktion ist nicht zulässig und wird vom UML-Werkzeug sofort zurückgewiesen.

Dagegen bietet die **lose Integration** dem Benutzer Filter für Im- und Export eines RFGs in das UML-Werkzeug. Nach dem Import arbeitet der Benutzer wie gewohnt mit dem Werkzeug, ohne daß zu diesem Zeitpunkt eine Anbindung an den RFG besteht. Nach dem Ende der Arbeiten exportiert das Werkzeug auf Anweisung des Benutzers das manipulierte UML-Modell typischerweise in einen neuen RFG. Teile des Modells, die nicht im RFG abgelegt werden können, werden verworfen. Da keine Möglichkeit besteht, den Benutzer im UML-Werkzeug an der Durchführung nicht zulässiger Änderungen zu hindern, stellt das Zurückschreiben der Informationen in den RFG ein schwieriges Problem dar. Die Integration beschränkt sich prinzipiell auf die Nutzung gemeinsamer Dateien. Dies sollte mit den meisten UML-Werkzeugen möglich sein. Den Ablauf des oben beschriebenen Anwendungsbeispiels bei loser Integration zeigt Abbildung 3-3.

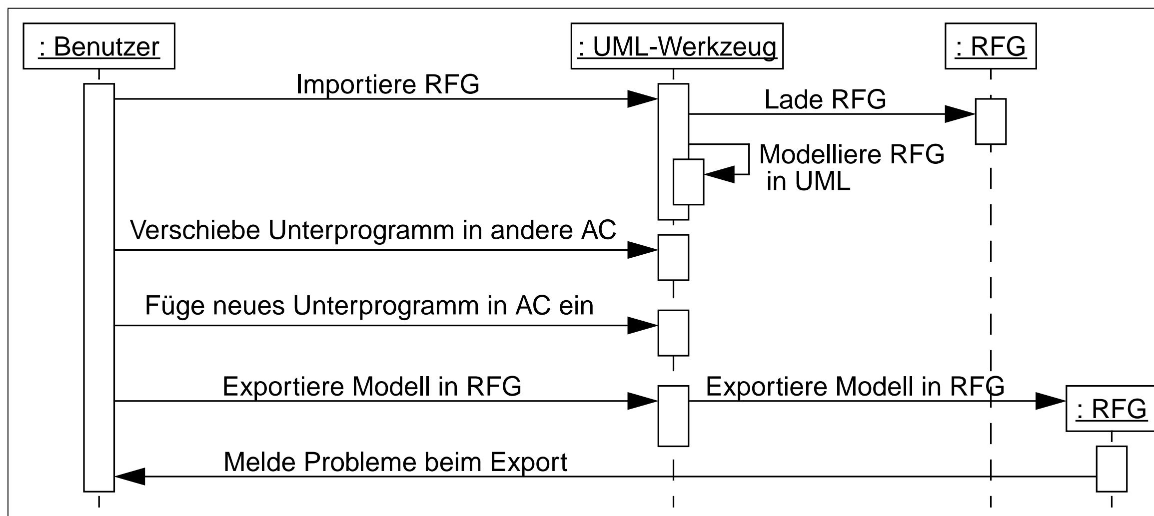


Abbildung 3-3: Anwendungsfall bei loser Integration

Die enge Integration ist, insbesondere falls Manipulationen am dargestellten UML-Modell relativ häufig vorkommen, der losen vorzuziehen. Sie bietet deutlich mehr Komfort und ermöglicht eine sehr viel einfachere Absicherung der Integrität der Daten. Fehler können sofort entdeckt und gegebenenfalls verhindert werden, statt erst beim Export behandelt zu werden. Allerdings stellt die enge Integration hohe Anforderungen an das UML-Werkzeug. Es muß daher erst geklärt werden, ob ein entsprechendes UML-Werkzeug gefunden werden kann.

Inzwischen existieren eine Vielzahl von Werkzeugen, die UML-Unterstützung bieten. Eine aktuelle Übersicht der bekanntesten ist auf der Homepage von *Mario Jeckle* im Internet zu finden ([umltools]). Von den verfügbaren Werkzeugen wurden *Rational Rose*, *Together Control Center* und *Innovator* näher betrachtet. Diese drei Werkzeuge bieten eine relativ umfassende Unterstützung der UML und sind am Markt breit akzeptiert, wodurch sie sich prinzipiell für eine Integration mit *Bauhaus* anbieten. Außerdem sind alle unter anderem für *Unix*-Plattformen verfügbar, was eine weitere Vorbedingung darstellt.

Integration mit Rational Rose

Rational Rose gehört zu den bekanntesten und am meisten verbreiteten UML-Werkzeugen am Markt. Da die ursprünglichen Initiatoren der UML – *Grady Booch*, *Ivar Jacobson* und *James Rumbaught* (die sog. ‘drei Amigos’) – alle bei *Rational* arbeiten, wird *Rose* oft als das offizielle UML-Werkzeug angesehen und gilt als Industriestandard. Es ist derzeit in der Version *Rational Rose 2000 Enterprise* für *Windows*- und *Unix*-Plattformen (z. B. *Solaris*) verfügbar.

Alle im Standard erwähnten UML-Diagrammarten werden unterstützt, wobei zwischen der Darstellung mit UML, OMT und der *Booch*-Methode gewechselt werden kann. Bedienung und Oberfläche des Werkzeugs sind einfach und klar. Zwischen UML-Modell und dessen Darstellung wird deutlich unterschieden. Es existiert eine Vielzahl von Erweiterungen für *Rational Rose*, die unter anderem die Erzeugung von Quelltext aus dem UML-Modell für die verschiedensten Programmiersprachen ermöglichen.

Allerdings unterstützt *Rose* nicht die komplette UML 1.3. So fehlen z. B. Multiplizitäten für Attribute, generische Kollaborationen und Abhängigkeiten für Attribute oder Operationen. Die *Unix*-Version von Rose scheint teilweise eine eher halbherzige Portierung der *Windows*-Version zu sein. Teile der Windowsumgebung werden nachmodelliert, z. B. bei der Darstellung der Oberfläche. Auch die von *Windows* bekannte Datei `win.ini` wird angelegt, ebenso ein `system`-Ordner. Die mitgelieferten Handbücher der *Unix*-Version beschreiben ausschließlich die *Windows*-Version.

Die Schnittstelle zur Erweiterung von *Rational Rose* besteht aus verschiedenen Teilen. Zum einen ist es möglich, Skripte in einer *Visual Basic for Applications* nachempfundenen Skriptsprache zu erstellen. Diese können in *Rose* über eigene Menüeinträge aufgerufen werden. Dazu kommt die sogenannte *Rose Automation* über *Object Linking and Embedding* (OLE), einen Windowsstandard für eingebettete Dokumente. Da OLE derzeit unter *Unix* nicht zur Verfügung steht, kann *Rose Automation* hier nicht verwendet werden. Die dritte Möglichkeit stellt die *Rose Type Library* dar, eine Bibliothek, über die aus verschiedenen Programmiersprachen auf *Rose* zugegriffen werden kann. Allerdings liegt praktisch keine Dokumentation über deren Verwendung unter *Unix* vor. Lediglich für die Verwendung mit *Visual Basic* und *Visual C* werden Beispiele aufgeführt. Das dort genannte Vorgehen funktionierte unter *Unix* nicht. Eine detailliertere Unterstützung für die Erstellung von Erweiterungen für *Rose* könnte die Teilnahme am Partnerprogramm von *Rational* bringen. Die angebotenen Möglichkeiten zur Integration erlauben unter *Unix* lediglich eine lose Integration, keine enge.

Integration mit Together Control Center

Ein weiteres recht bekanntes UML-Werkzeug ist *Together Control Center* von *Togethersoft*. Es ist speziell für die Arbeit mit den Programmiersprachen *Java* und *C++* konzipiert und selbst vollständig in *Java* implementiert. Daher ist es konzeptionell plattformunabhängig und kann in den verschiedensten Umgebungen eingesetzt werden, darunter *Windows* und verschiedene *Unix*-Varianten. Derzeit liegt die Version 4.0 vor.

Together Control Center unterstützt alle im Standard aufgeführten Diagrammarten und ist in der Lage, UML-Modelle und -Diagramme von *Rational Rose* zu im- und exportieren. Die große Stärke von *Together Control Center* liegt in der Fähigkeit, jederzeit zwischen dem *Java* bzw. *C++* Quelltext eines Systems und seiner UML-Modellierung zu wechseln. Aus beliebigem bestehendem Quelltext werden automatisch UML-Modellelemente abgeleitet. Änderungen wirken sich sofort in Modell und Quelltext aus, egal, wo sie vorgenommen wurden. Um dies erreichen zu können, werden teilweise Konstrukte der UML in spezieller Art und Weise interpretiert. So wird z. B. jede Operation, deren Name mit 'set' beginnt und einen einzigen Parameter besitzt, als spezielle Zugriffsoperation auf ein sogenanntes *Property* interpretiert. Dies ist nötig, um Systeme erstellen zu können, die mit dem Komponentenframework von *Java* – *JavaBeans* – arbeiten.

Auch *Together Control Center* unterstützt UML 1.3 nicht vollständig. Es fehlen beispielsweise generische Kollaborationen, Subsysteme, Utilities, Abhängigkeiten von Attributen und Operationen. Assoziationsklassen sind nur eingeschränkt möglich. Subjektiv ergibt sich der Eindruck, daß größere Teile der UML fehlen als in *Rational Rose*, ein exakter Vergleich fand jedoch nicht statt. Da *Together Control Center* vollständig in *Java* geschrieben ist, treten teilweise Performanceprobleme auf. Insbesondere unter *Solaris* war das Programm extrem langsam, was wahrscheinlich auf die Graphikausgabe zurückzuführen ist. Dies könnte zu Problemen bei der Modellierung großer Systeme führen.

Die Implementierung des UML-Werkzeugs in *Java* führt gleichzeitig dazu, daß es sehr einfach und umfangreich erweitert werden kann. Über die Programmierschnittstelle (API) *Together Open API* kann jedes *Java*-Programm direkt auf die Implementierung des Werkzeugs zugreifen. Neben *Java* können Erweiterungen auch in den Skriptsprachen *TCL* und *JPython* geschrieben werden. Um die oben erwähnte sofortige Anpassung der jeweils anderen Darstellung bei Änderungen zu ermöglichen, ist in *Together Control Center* eine API integriert, die bei Aktionen des Benutzers beliebige Objekte benachrichtigen kann. Damit ist eine enge Integration prinzipiell machbar. Nachfragen bei *Togethersoft* ergaben jedoch, daß diese API nur intern zugänglich ist, bzw. offiziell nicht unterstützt wird. Aus diesem Grund wurde diese Möglichkeit fallengelassen.

Insgesamt folgt *Together Control Center* einem anderen Ansatz als *Rational Rose*. Es konzentriert sich auf die Unterstützung weniger Programmiersprachen (*C++*, *Java*) und versucht, für diese eine enge Integration von Quelltext und UML-Modellierung zu erreichen. Da die hierfür eingesetzte API allerdings für externe Stellen nicht offiziell verfügbar ist, ist dennoch derzeit nur eine lose Integration mit *Bauhaus* möglich. Teilweise leidet das Programm unter schlechter Performance und fehlender Unterstützung für UML-Konstrukte.

Integration mit Innovator

Seit inzwischen ca. 13 Jahren wird *Innovator*, das CASE-Werkzeug des Systemhauses *MID* entwickelt. Es ist unter den verschiedensten Plattformen verfügbar, darunter *Windows*, diverse *Unix*-Varianten (z. B. *AIX*, *Solaris*, *Linux*), *OS/2* und *AlphaVMS*. Aktuell ist die Version 6.2. Mit Version 7.0 ist in Kürze zu rechnen.

Neben UML unterstützt *Innovator* eine Vielzahl anderer Notationen und Vorgehensweisen für den nicht objektorientierten Entwurf, z. B. *Structured Analysis*, die im Rahmen dieser Arbeit nicht von Bedeutung sind. Zentrales Element von *Innovator* ist das Repository, in dem alle Daten abgelegt werden. Von hier aus kann über verschiedene Methoden auf sie zugegriffen werden. So ist es beispielsweise möglich, Projekte durchzuführen, die teilweise Objektorientierung, teilweise Datenorientierung verwenden. Aus mit *Rational Rose* erstellten UML-Modellen können Klassen-, Anwendungsfall- und Sequenzdiagramme übernommen werden.

Innovator unterstützt in der derzeitigen Version 6.2 noch nicht UML 1.3, sondern erst UML 1.1. Erst Innovator 7.0 soll hier Abhilfe schaffen. Ebenfalls erst in Version 7.0 enthalten sind Kollaborations-, Komponenten-, Verteilungs- und Objektdiagramme. Ebenso wie die anderen untersuchten Werkzeuge ist *Innovator* nicht in der Lage, Abhängigkeiten von Attributen oder Operationen zu modellieren. Modellelementen in Klassendiagrammen können zwar Notizen zugewiesen werden, nicht aber solchen in Sequenzdiagrammen. Ein Test der neuen Version war noch nicht möglich.

Um die Funktionalität des CASE-Werkzeugs zu erweitern existieren zwei Wege. Zum einen ist es möglich, über *TCL*-Skripte auf das bereits angesprochene Repository und die in ihm vorliegenden Daten zuzugreifen. Zum anderen kann beliebigen externen Programmen der Zugriff auf das Repository gestattet werden. Da beide Möglichkeiten nicht über die Benutzungsschnittstelle des Werkzeugs sondern die Datenhaltungskomponente verlaufen und eine unmittelbare Benachrichtigung über Benutzeraktionen anscheinend nicht vorgesehen ist, erlaubt dies lediglich eine lose Integration mit *Bauhaus*.

Besonders bei Systemen, die noch mit nicht objektorientierten Methoden erstellt wurden und deren Dokumentation teilweise noch vorhanden ist, erweist sich *Innovator* als sehr mächtig. Die UML-Unterstützung ist derzeit noch nicht vollständig. Eine lose Integration mit *Bauhaus* wäre denkbar, eine enge Integration scheint allerdings nicht machbar.

Fazit der Werkzeugbetrachtungen

Obwohl zunächst eine enge Integration zwischen Bauhaus und einem UML-Werkzeug favorisiert wurde, zeigte die Untersuchung der verschiedenen Werkzeuge, daß keines diese zuläßt. Ein klarer Favorit für die Verwendung in dieser Arbeit konnte nicht ermittelt werden. Alle Werkzeuge setzen eigene Schwerpunkte und werden daher auch für verschiedene Anwendergruppen interessant sein. Eine vollständige Unterstützung der UML bietet keines der Werkzeuge, wobei *Rational Rose* subjektiv am meisten abdeckt. Aus diesem Grund wurde in Absprache mit Betreuer und Prüfer dieser Arbeit eine werkzeugunabhängige Lösung angestrebt, die es dem Benutzer erlaubt, zur Darstellung einer Architektur das UML-Werkzeug seiner Wahl einzusetzen. Zu diesem Zweck existiert seit kurzem ein Standard der OMG.

Werkzeugunabhängige Integration mittels XML Metadata Interchange

XML Metadata Interchange (XMI) ist ein Standard der OMG zum streambasierten Austausch von Metadaten und Metamodellen ([xmi]). Darunter fallen unter anderem auch UML-Modelle. Mit XMI ist es möglich, UML-Modelle werkzeugunabhängig zu definieren und zu speichern, sowie in verschiedenen CASE-Werkzeugen zu betrachten und zu manipulieren. Aktuell ist Version 1.0, XMI 1.1 wird derzeit entwickelt.

Die ausgetauschten Metadaten und -modelle müssen einem weiteren Standard der OMG entsprechen, der *Meta-Object Facility* (MOF) ([mof]). Diese legt fest, wie auf einheitliche Weise Metamodelle definiert werden und bietet Werkzeugen, z. B. Metadaten-Repositories, eine standardisierte API zum Zugriff auf die Metadaten. Da XMI auf MOF basiert, können mit ihm erzeugte Modellbeschreibungen auch in MOF-konformen Repositories abgelegt werden. Um das UML-Metamodell MOF-konform zu machen, sind einige Änderungen an ihm nötig, die in Kapitel 6 des aktuellen UML-Standards beschrieben sind. (siehe [uml1.3], Kapitel 6)

Neben MOF basiert XMI auf der *Extensible Markup Language* (XML) ([xml]), einem Standard des *World Wide Web Consortium* (W3C). XML erlaubt die einfache Definition neuer Datenformate für den Austausch strukturierter Daten. Hierzu können einfache Grammatiken in Form einer *Document Type Definition* (DTD) festgelegt werden, die die Struktur der Daten beschreiben.

XMI legt fest, wie aus einem MOF-konformen Metamodell automatisch eine entsprechende DTD erzeugt werden kann, mit der Metadaten als XML-Datei übertragen werden können. Obwohl XMI noch sehr neu ist, wird es bereits von verschiedenen CASE-Werkzeugen unterstützt, für andere wurde eine Unterstützung angekündigt. Für *Rational Rose* kann eine XMI-Erweiterung aus dem Internet bezogen werden. *Together Control Center* unterstützt XMI direkt. Ab Version 7.0 soll *Innovator* zumindest in der Lage sein, UML-Modelle als XMI-Datei zu exportieren. Meist basiert die XMI-Unterstützung auf der ursprünglich im XMI-Standard enthaltenen DTD für UML 1.1, weshalb mit diesen Werkzeugen nur der Austausch von in UML 1.1 erstellten Modellen möglich ist. Lediglich Rational Rose bietet XMI inzwischen auch für UML 1.3. Eine fehlerfreie XMI-Unterstützung ist nach den im Verlauf dieser Diplomarbeit gesammelten Erkenntnissen noch in keinem der untersuchten Werkzeuge enthalten. Hierzu sollen einige Beispiele gegeben werden:

- *Rational Rose* exportiert Klassen prinzipiell als finale Klassen, angezeigt durch den Eintrag `isLeaf = true`. Finale Klassen können nicht als Ausgangspunkt einer Vererbung dienen, d. h. andere Klassen können nicht von ihr erben. Obwohl dies im Rahmen dieser Arbeit keine Probleme verursacht, da Vererbung nicht verwendet wird, ist es allgemein nicht korrekt. Standardmäßig sollte dieser Eintrag den Wert `false` enthalten. (vgl. [uml1.3], S. 2-35) Zudem exportiert *Rose* keine Parameter, obwohl es sie importiert und auch anzeigt.
- *Together Control Center* exportiert Attribute falsch. Eine genauere Beschreibung dieses Fehlers ist im Kapitel „Korrigieren der XMI-Ausgabe von Together Control Center“ auf Seite 97 zu finden.

Da über XMI nur Informationen übertragen werden können, die im UML-Metamodell enthalten sind, ist es nicht möglich, Informationen über die Darstellung des Modells zu übermitteln. Dieses Problem wird derzeit im Rahmen der Arbeiten zu UML 2.0 behandelt, indem das Metamodell um diese Informationen erweitert werden soll. (vgl. [umlnews]) XMI-Dateien erreichen sehr schnell eine enorme Größe, was teilweise daran liegt, daß XMI auf XML basiert und daher textbasiert ist und teilweise von den in XMI verwendeten sehr langen Bezeichnern herrührt. Schon mittlere Modelle sind mehrere MB groß, pro Modellelement kann mit etwa 500 Byte gerechnet werden, wobei als Modellelement neben Klassen, Paketen, Attributen, Operationen und Asso-

ziationen auch Parameter und zusätzliche Werte gelten. XMI 1.1 soll deutlich kleinere Dateien erzeugen und dadurch dieses Problem beheben bzw. zumindest abschwächen.

Vorgeschlagene Integration

Trotz der zuvor genannten Schwierigkeiten wurde im Rahmen dieser Arbeit XMI als Schnittstelle zwischen Bauhaus und UML-Werkzeugen eingesetzt. Dadurch entfällt die Notwendigkeit, sich auf ein bestimmtes UML-Werkzeug festzulegen. Die Qualität der XMI-Unterstützung sollte mittelfristig besser werden. Lösungen für fehlende Layoutinformationen und die Größe der Dateien werden bereits diskutiert. Insbesondere wird es mit XMI möglich, aus einer eventuell unabhängig von *Bauhaus* erstellten UML-Modellierung des untersuchten Softwaresystems einen RFG zu erstellen, den *Bauhaus*-Werkzeuge für die weitere Architekturerkennung verwenden können. Ein Schema für die vorgeschlagene Integration zwischen RFG und UML zeigt Abbildung 3-4.

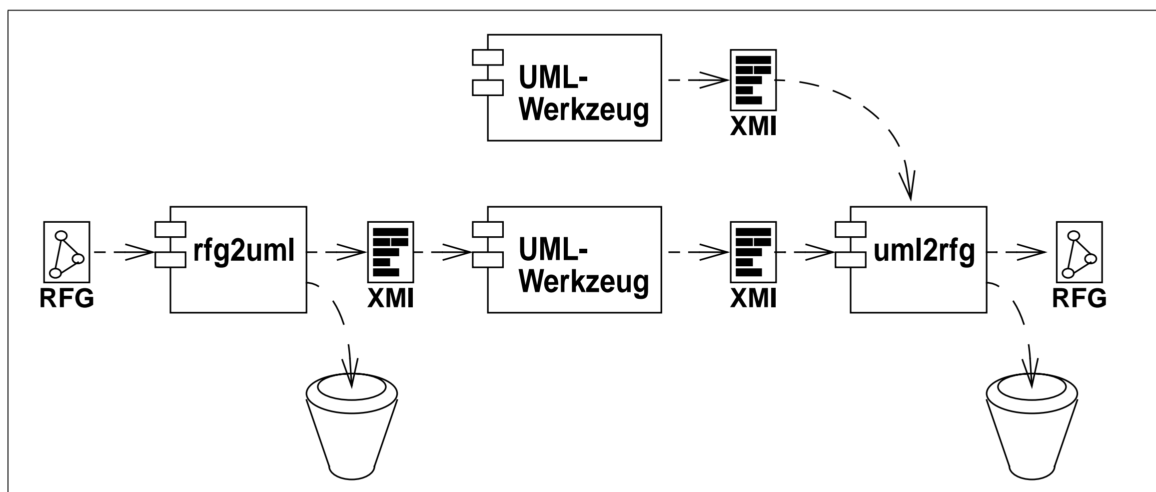


Abbildung 3-4: Prinzipielle Integration

Die im Rahmen dieser Arbeit zu erstellenden Werkzeuge sind:

- ***rfg2uml***, das einen RFG entgegennimmt, eine enthaltene Architektur-View in ein UML-Modell konvertiert und dieses in eine XMI-Datei schreibt und
- ***uml2rfg***, das eine durch ein beliebiges UML-Werkzeug erzeugte XMI-Datei einliest, das enthaltene UML-Modell in eine neue Architektur-View eines RFGs umwandelt und abspeichert.

Sowohl bei der Konversion des RFGs in UML als auch bei der Rücktransformation können Informationsverluste auftreten, was nach den im Kapitel „Anforderungen an die Modellierung“ auf Seite 26 beschriebenen Anforderungen zulässig ist.

Bevor eine Architekturbeschreibung mit *rfg2uml* transformiert werden kann, muß in den RFG eine Architektur-View eingefügt werden, die alle abzubildenden Knoten und Kanten enthält. Bisher kann dies einfach dadurch erfolgen, daß Base- und User-View des RFGs verschmolzen werden. Ein entsprechendes Werkzeug wurde im Rahmen dieser Arbeit realisiert (siehe Kapitel „Erstellen der Architektur-View“ auf Seite 96).

Bei der Rücktransformation in einen RFG mit *uml2rfg* kann eine neue Architektur-View in einem gegebenenfalls bereits existierenden RFG angelegt werden. Eine andere Möglichkeit ist es, prinzipiell einen neuen RFG zu erstellen, was hier aus Gründen der Einfachheit angenommen wird. In beiden Fällen muß noch der ursprüngliche Quell-RFG mit dem neuen RFG abgeglichen werden, um inkonsistente RFGs zu vermeiden. Insbesondere muß es möglich sein, die neue Architektur-View mit den Views zu integrieren, aus denen die ursprünglich transformierte Architektur-View entstanden ist, derzeit Base- und User-View. Hierzu wird im Rahmen dieser Arbeit ein weiteres Werkzeug ***mergeRFGs*** vorgeschlagen. Ein sehr ähnliches Werkzeug wird unter der Bezeichnung *rfgdiff* im Projekt *Bauhaus* bereits erstellt. Es vergleicht zwei RFGs, bzw. einzelne Views aus ihnen und gleicht Änderungen so ab, daß ein neuer, konsistenter RFG erzeugt wird, wie dies Abbildung 3-5 zeigt. Dieser Vor-

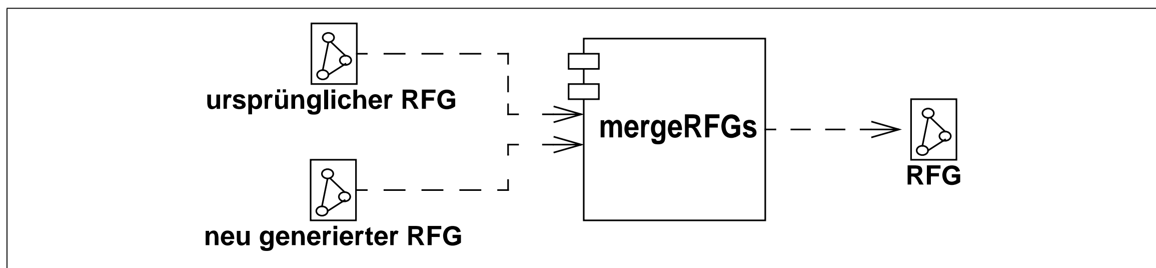


Abbildung 3-5: Integration der Änderungen mit dem ursprünglichen RFG

gang ist sehr kompliziert, so daß die Erstellung dieses allgemeinen Werkzeugs über den Rahmen dieser Arbeit weit hinausgeht. Stattdessen werden im folgenden Regeln festgelegt, wie definierte Änderungen übertragen werden können. Alle anderen Änderungen können nicht übertragen werden.

Es ergeben sich für jeden zu modellierenden Teil der Architekturbeschreibung fünf Untersuchungsbereiche:

1. welche UML-Sprachmittel sind prinzipiell für eine Modellierung geeignet,
2. welche Transformationsvorschriften existieren für die Modellierung in UML,
3. welche Änderungen darf der Benutzer an dieser Modellierung vornehmen, wenn eine Rückübertragung der Informationen in den RFG möglich sein soll,
4. welche Vorschriften existieren für die eigentliche Rücktransformation und
5. wie sind die durch zulässige Änderungen neu hinzukommenden Informationen mit dem ursprünglichen RFG zu integrieren.

Da die Punkte drei und fünf eng zusammenhängen, werden sie jeweils gemeinsam behandelt. Für die grundsätzliche Modellierung einer Architekturbeschreibung bzw. der entsprechenden View des RFGs wurde Punkt eins im Kapitel „Prinzipielle Modellierung einer Architekturbeschreibung“ auf Seite 30 bereits diskutiert. Da die für die Behandlung der weiteren Punkte benötigten Informationen in diesem Kapitel dargelegt wurden, können die Punkte zwei bis fünf im folgenden behandelt werden.

3.2.4 Genaue Modellierung einer Architekturbeschreibung

Eine Architekturbeschreibung wird als UML-Modell modelliert.

Transformation nach UML (rfg2uml)

Aus der die Architekturbeschreibung enthaltenden View des angegebenen RFGs wird ein UML-Modell erstellt. Der Name des UML-Modells ergibt sich aus dem Namen der View. Eine View *Current-Architecture* in einem RFG *emacs* wird beispielsweise in ein UML-Modell mit dem Namen *Current-Architecture* transformiert. Der Name des RFGs geht verloren. Die in der View enthaltenen Knoten und Kanten ergeben Elemente des UML-Modells. Die angegebene Transformation zeigt Listing 3-1 in Pseudocode.

```
MapViewToUML
  in rfg: RFG;
  in view: View;
  return umlModel: Model := new Model;
begin
  umlModel.setName(view.getName);
  umlModel.setVisibility(public);
  foreach Node n in rfg(view) do
    umlModel.add(MapNodeToUML(n, umlModel));
  end foreach;
end;
```

Listing 3-1: Transformation einer View nach UML

Rücktransformation in einen RFG (uml2rfg)

Bei der Rücktransformation in einen RFG wird ein UML-Modell in eine neue View in einem neuen RFG transformiert. Der Name der View ergibt sich aus dem Namen des Modells. Alle in dem Modell enthaltenen Modellelemente ergeben Knoten bzw. Kanten, die in der neuen View sichtbar sind. Dies verdeutlicht Listing 3-2.

```
MapModelToRFG
  in umlModel: Model;
  inout rfg: RFG;
begin
  view := rfg.createView;
  if not rfg.viewExists(umlModel.getName) then
    view.setName(umlModel.getName);
  else raise exception;
  foreach ModelElement elem in umlModel do
    MapModelElementToRFG(elem, view);
  end foreach;
end;
```

Listing 3-2: Rücktransformation eines UML-Modells in einen RFG

Erlaubte Änderungen und deren Integration mit dem ursprünglichen RFG

Es sind nur die wenigen folgenden Benutzermanipulationen erlaubt:

- Jedes Modellelement darf aus dem Modell entfernt werden. Es gehört dann nicht mehr zur erkannten Architektur. Elemente des RFGs, aus denen es resultierte, werden bei der Integration aus der entsprechenden View gelöscht.
- Bestimmte Modellelemente dürfen in das Modell hinzugefügt werden. Welche dies sind, wird in den späteren Kapiteln erläutert. Bei der Integration werden Knoten und Kanten, die aus dem neuen Modellelement resultieren, in die Architektur-View eingefügt.

Prinzipiell ist auch eine Änderung des Namens des Modells zulässig. Dies führt dazu, daß beim Zurückschreiben eine Architektur-View des neuen Namens angelegt wird. Bei der Integration des ursprünglichen und des neuen RFGs muß die Architektur-View im ursprünglichen RFG den neuen Namen erhalten. Probleme entstehen hierdurch, falls die verwendeten Werkzeuge auf die Architektur-View mit Hilfe eines statisch festgelegten Namens zugreifen. In diesem Fall ist diese Benutzermanipulation verboten.

Alle weiteren Manipulationen sind bisher nicht zulässig.

3.2.5 Weiteres Vorgehen

Das weitere Vorgehen bei der Ermittlung einer UML-Modellierung der Ist-Architektur eines SW-Systems gliedert sich in verschiedene Teile, analog zur möglichen Aufteilung der Architekturerkennung in *Bauhaus*: in die Erkennung atomarer Komponenten, die Subsystem-, Konnektoren- und die Protokollerkennung. Zunächst wird jeweils erläutert, welche architektonischen Informationen im Rahmen des gegebenen Teilprojekts ermittelt werden, bzw. welche Informationen für eine Transformation in Frage kommen. Anschließend wird untersucht, wie diese Informationen vorliegen, d. h. welche Knoten und Kanten sie im RFG repräsentieren und welche UML-Sprachmittel geeignet wären, diese abzubilden. Schließlich wird eine spezifische Modellierung der vorliegenden Informationen mit der UML vorgeschlagen und eine mögliche Rücktransformation des UML-Modells in einen RFG angegeben.

Modellierung atomarer Komponenten

Kapitel 4

Eine **atomare Komponente** (AC) ist eine nicht-hierarchische Komponente, die aus miteinander in Beziehung stehenden globalen Variablen und Konstanten, Unterprogrammen und/ oder benutzerdefinierten Typen besteht. (vgl. [Kos00], S. 27)

Unter einer **Komponente** wird in *Bauhaus* eine Gruppe von durch ein gemeinsames Ziel oder Konzept miteinander in Beziehung stehenden Elementen verstanden, die auf architektonischer Ebene von Bedeutung sind. (vgl. [Kos00], S. 27) Neben atomaren Komponenten fallen darunter auch Subsysteme, die im Kapitel „Modellierung von Subsystemen“ auf Seite 67 behandelt werden. Komponenten sind in *Bauhaus* nicht mit Modulen zu verwechseln. Komponenten stellen *logische* Gruppierungen von Elementen dar, Module *konkrete*. Im Idealfall sind diese deckungsgleich, was aber nicht der Fall sein muß.

ACs haben für die Architekturerkennung eine zentrale Bedeutung und bilden die Grundlage für die meisten anderen Verfahren. Aus diesem Grund wurde besonders viel Aufwand in eine möglichst gute Modellierung in UML investiert. ACs werden im Projekt *Bauhaus* mit Hilfe von Analysen und Benutzerinteraktion (halb-automatisch) erkannt. Der Benutzer wählt Analysen aus, die auf dem RFG ausgeführt werden, bewertet und korrigiert die Ergebnisse. Die Algorithmen und Verfahren, die zu einer Erkennung führen, sind im Rahmen dieser Arbeit nicht von Bedeutung, da von einer (hinreichend) abgeschlossenen Erkennung ausgegangen wird. Ausführliche Informationen zu diesem Themengebiet enthält [Kos00].

Es können verschiedene Arten von ACs unterschieden werden:

- Ein **abstrakter Datentyp** (ADT) gewährt Zugriff auf Variablen seines Typs nur über definierte Zugriffsoperationen. Dadurch wird von der tatsächlichen Implementierung des Typs abstrahiert und eine klare Schnittstelle zur Umwelt festlegt. (vgl. [Kos00], S.29)
- Ein **abstraktes Datenobjekt** (ADO) ist eine Gruppe von globalen Variablen und Konstanten, auf die ebenfalls nur über definierte Operationen zugegriffen werden kann. ADOs werden auch als abstrakte Objekte oder Abstract State Encapsulation (ASE) bezeichnet. Sie sind eng mit ADTs verwandt. Jedes ADO kann sehr einfach auf einen ADT abgebildet werden. Deshalb werden ADOs oft als spezielle ADTs angesehen. (vgl. [Kos00], S. 30 f.)

- Ein **Hybrid** (hybrid component, HC) ist eine Mischung aus ADT und ADO, beispielsweise ein ADT mit globalem Zustand oder ein ADO mit privaten Typen. ACs, bei denen eine klare Zuordnung zu ADT oder ADO nicht möglich oder nötig ist, werden in *Bauhaus* als Hybrid klassifiziert. (vgl. [Kos00], S. 31)
- Eine **funktionale Komponente** (related subprograms, RS) besteht aus einer Menge von Unterprogrammen, die als zusammengehörig angesehen werden, z. B. weil sie eine gemeinsame Aufgabe erfüllen bzw. einen gemeinsamen Dienst erbringen. Ein Beispiel wären Unterprogramme einer mathematischen Bibliothek. (vgl. [Kos00], S. 90)

Im Zusammenhang mit ACs wird auch die Modellierung des größten Teils der im Basis-RFG vorliegenden Informationen untersucht, da diese in enger Abhängigkeit zur Modellierung von ACs steht und umgekehrt. Der Basis-RFG enthält Informationen über die statische Struktur des SW-Systems, wie sie direkt aus dem Quelltext bzw. der IML-Darstellung abgeleitet wurde (siehe Kapitel „Basis-RFG“ auf Seite 8). Nicht betrachtet wird die Modellierung von Modulen, die parallel zu ACs anzusiedeln sind. Eine Möglichkeit ist im Kapitel „Erweiterungen der Modellierung“ auf Seite 102 enthalten.

4.1 Vorliegende Informationen im RFG

Eine AC liegt im RFG als Knotenmenge vor, bestehend aus genau einem Atomic_Component-Knoten und beliebig vielen weiteren Knoten, die Part_Of-Kanten zum Atomic_Component-Knoten haben. Teilknoten können z. B. Variable- oder Type-Knoten aus dem Basis-RFG sein, nicht jedoch weitere Atomic_Component-Knoten. Dies würde zu einem Subsystem führen. Subsysteme werden derzeit ebenfalls mit Atomic_Component-Knoten modelliert. Die Gründe hierfür werden im Kapitel „Vorliegende Informationen im RFG“ auf Seite 67 behandelt. Ein Beispiel für eine im RFG vorliegende AC zeigt Abbildung 4-1.

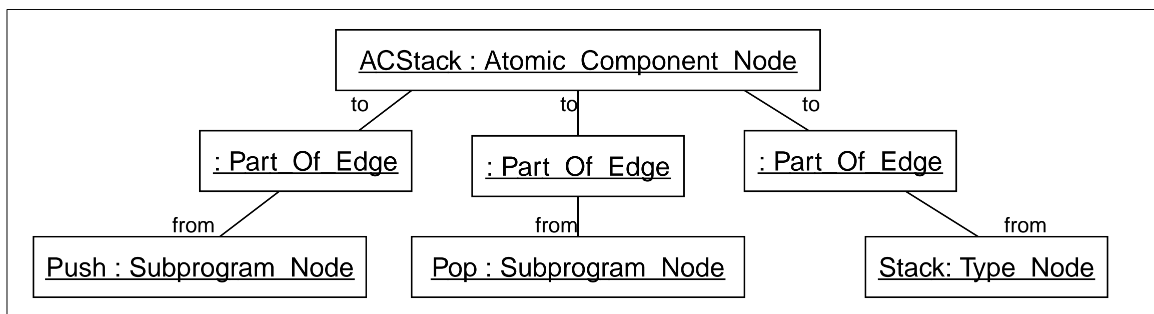


Abbildung 4-1: Beispiel für eine AC im RFG

Jeder Knoten kann in der User-View nur in höchstens einer AC enthalten sein. Einige Analysen erlauben zwar Knoten, die in mehr als einer AC enthalten sind, will der Benutzer solche ACs allerdings akzeptieren, d. h. in die User-View aufnehmen, müssen die entsprechenden Knoten eindeutig einer AC zugewiesen werden und aus allen anderen entfernt werden. In der Base-View sind keine ACs enthalten. Daher kann davon ausgegangen werden, daß auch in der betrachteten Architektur-View, die aus der Verschmelzung der User- und der Base-View entsteht, kein Teilknoten in

mehr als einer AC enthalten ist. ACs, die nicht in der Architektur-View enthalten sind, sind nicht Teil der erkannten Architektur und werden im Rahmen dieser Arbeit ignoriert.

Die Klassenhierarchie der gegenüber dem Basis-RFG neu hinzukommenden Knotentypen zeigt Abbildung 4-2.

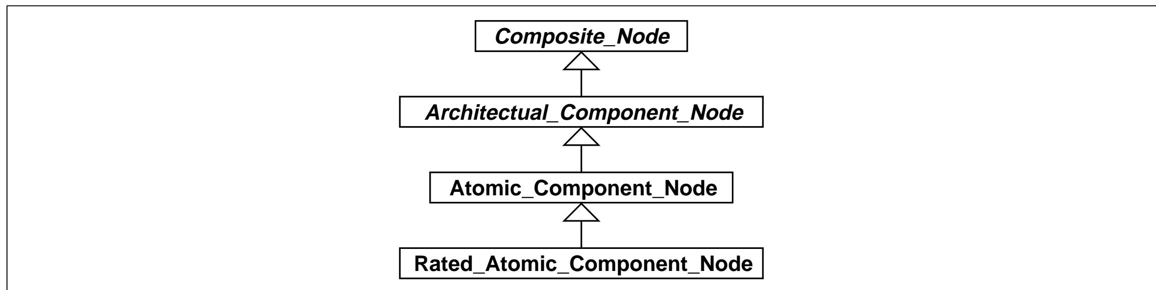


Abbildung 4-2: Klassenhierarchie der Knoten der AC-Erkennung

Eine kurze Beschreibung dieser Knotentypen enthält Tabelle 4-1. Atomic_Component-Knoten werden für die Modellierung aller zuvor besprochenen Arten von ACs eingesetzt. Von welcher Art eine AC ist, gibt ein Attribut des Typs `Component_Classification_Type` an. Der Wert `ADT_Component_Classification` zeigt einen ADT an, `ADO_Component_Classification` ein ADO. Eine HC wird mit dem Wert `HC_Component_Classification` gekennzeichnet, `RS_Component_Classification` weist auf einen RS hin. Falls die Art der AC nicht bekannt ist, kann das Attribut auf den Wert `Unknown_Component_Classification` gesetzt werden. Ebenso wie für verschiedene Knoten des Basis-RFGs kann auch für Atomic_Component-Knoten ihre Sichtbarkeit im umschließenden Kontext mit Hilfe eines Attributs angegeben werden.

Rated_Atomic_Component-Knoten werden von Analysen für die Erkennung von ACs benötigt. Da die in ihnen enthaltene Bewertung außerhalb dieser Analysen nicht von Bedeutung ist, werden diese Knoten im folgenden mit Atomic_Component-Knoten gleichgesetzt.

Bezeichnung	Beschreibung
`Atomic_Component`	atomare Komponente (AC)
`Rated_Atomic_Component`	bewertete AC, von Analysen verwendet, außerhalb dieser mit Atomic_Component gleichzusetzen

Tabelle 4-1: Knoten der AC-Erkennung

Die Vererbungshierarchie der im Rahmen der AC-Erkennung neu eingesetzten Kantentypen zeigt Abbildung 4-3.

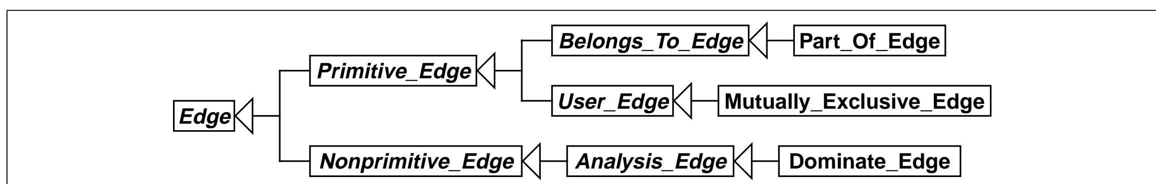


Abbildung 4-3: Klassenhierarchie der Kanten der AC-Erkennung

Sie werden in Tabelle 4-2 beschrieben. Part_Of-Kanten legen die Zugehörigkeit eines Knotens zu einer AC fest und sind daher für die Architekturbeschreibung sehr wichtig. Mutually_Exclusive- und Dominate-Kanten sind dagegen hauptsächlich für die AC-Erkennung von Bedeutung und würden eine Darstellung der Ergebnisse lediglich unübersichtlicher machen. Sie werden daher im weiteren Verlauf ignoriert.

Bezeichnung	Quelle s	Ziel d	Beschreibung
`Part_Of`	`Programming_Entity`	`Atomic_Component`	s ist Teil von d
`Mutually_ Exclusive`	`Programming_Entity`	`Programming_Entity`	s und d sind nicht Teil der gleichen AC
`Dominate`	`Subprogram`	`Subprogram`	s dominiert d

Tabelle 4-2: Kanten der AC-Erkennung

4.2 Semantisch äquivalente UML-Sprachmittel

ACs sind auf architektonischer Ebene die Grundbausteine der statischen Struktur eines SW-Systems. Sie gruppieren die Elemente des Basis-RFGs.

4.2.1 Modellierung von atomaren Komponenten

Zur Modellierung einer AC bieten sich in der UML die Sprachelemente Komponente (Metaklasse `Component` im UML-Metamodell), Paket (Metaklasse `Package`), Klasse (Metaklasse `Class`) und Utility an.

Modellierung mit UML-Komponenten

Vor allem der Name suggeriert eine semantische Nähe von atomaren Komponenten in *Bauhaus* und Komponenten in UML. Eine Komponente stellt in UML einen abgeschlossenen Teil einer Systemimplementation dar, d. h. die tatsächliche, *konkrete* Struktur einer Implementation. In *Bauhaus* ist eine Komponente eine *logische* Gruppierung, die von der tatsächlichen abweichen kann, z. B. wegen bestimmten Regeln der verwendeten Implementierungssprache. Eine UML-Komponente ist daher eher mit einem Modul in *Bauhaus* als einer AC vergleichbar.

Modellierung mit Paketen

Für eine rein logische Gruppierung von Modellelementen werden in UML Pakete verwendet. Sie stellen einen allgemeinen Gruppierungsmechanismus dar. Der Zweck einer solchen Gruppierung oder die Kriterien, nach der sie erfolgt, sind nicht näher definiert. (vgl. [uml1.3], S. 2-173 f.) Damit sind Pakete prinzipiell für eine Modellierung von ACs geeignet. Pakete können auf verschiedene Arten dargestellt werden. Abbildung 4-4 zeigt drei Möglichkeiten, ein in allen drei Fällen identisches Paket *Ein_Paket* zu visualisieren, das ein eingebettetes Paket *Ein_Teilpaket* enthält. Möglichkeit (a) zeigt dies, indem das Teilpaket graphisch innerhalb des Körpers des

umschließenden Pakets dargestellt wird. Bei (b) wird eine spezielle 'Enthält'-Verbindung zwischen den Paketen gezeichnet, während (c) das eingebettete Paket überhaupt nicht anzeigt, was ebenfalls zulässig ist.

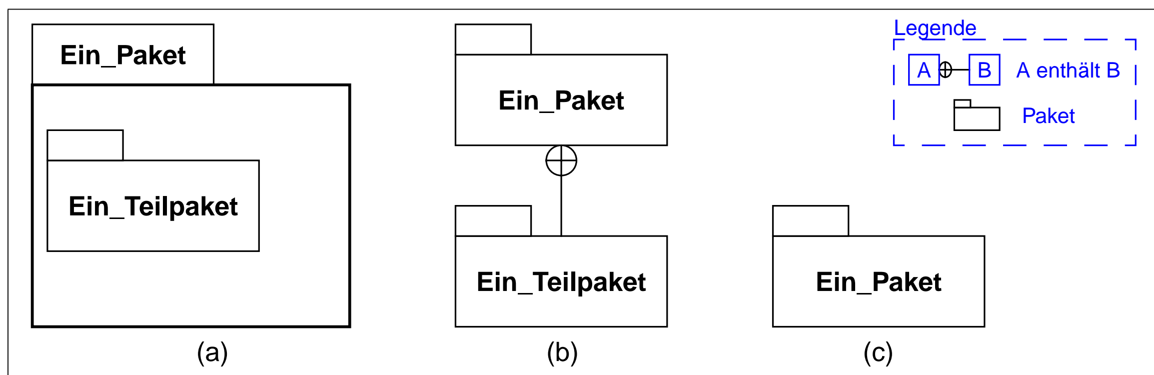


Abbildung 4-4: Alternative Darstellungen eines Pakets in UML

Probleme treten bei der Modellierung der in der AC enthaltenen Elemente auf. Üblicherweise werden bestimmte Modellelemente, die hierfür geeignet wären (z. B. Attribute) nicht direkt in Paketen gruppiert, obwohl der UML-Standard dies ermöglichen würde. Daher sind viele UML-Werkzeuge nicht in der Lage, eine solche Gruppierung darzustellen. Außerdem werden Pakete normalerweise nicht als atomar angesehen, was beim Betrachter zu Verwirrung führen könnte.

Modellierung mit Klassen

Eine Klasse wird in UML dazu verwendet, eine Menge von Methoden, Operationen, Attributen und weiteren eingebetteter Klassen festzulegen, die die Struktur und das Verhalten eines Objekts vollständig beschreibt. Klassen dienen also als eine Art Vorlagen für Objekte. Durch Instanziierung können zur Laufzeit entsprechende Objekte angelegt werden. (vgl. [uml1.3], S. 2-61, S.2-26 f.) Auch Klassen können auf verschiedene Arten dargestellt werden, die Abbildung 4-5 zeigt, (a) mit oder (b) ohne Details.

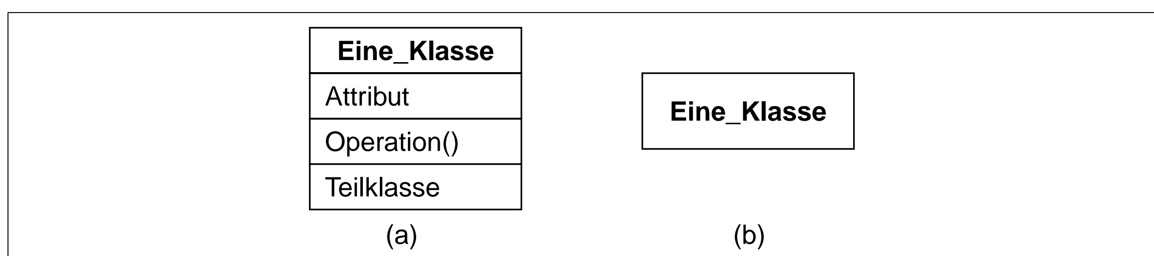


Abbildung 4-5: Alternative Darstellungen einer Klasse in UML

Klassen bieten eine logische, statische Sicht auf das System. Ihre Semantik ähnelt der atomarer Komponenten relativ gut, insbesondere bei ADTs, aber auch ADOs. Aus diesem Grund wurde diese Möglichkeit im Rahmen dieser Arbeit genauer untersucht, wobei sich insbesondere bei der Modellierung der in der AC enthaltenen Knoten Schwierigkeiten ergaben. So sollten Unterprogramme als Operationen abgebildet werden, was aus verschiedenen Gründen zunächst nur als Klassenoperationen möglich war (z. B. um den ansonsten definierten impliziten Parameter auf die Klasseninstanz zu vermeiden). Klassenoperationen sind Operationen, die der Klasse direkt zugeordnet sind, keiner Instanz.

Eine wichtige Abweichung zwischen einer AC und einer Klasse ist die Möglichkeit, von Klassen Instanzen zu bilden. Dies ist bei ACs so nur für ADTs möglich. Bei diesen wäre es denkbar, den Atomic_Component- und den zugehörigen Type-Knoten zu verschmelzen und gemeinsam als Klasse abzubilden. Probleme treten bei ADTs mit mehr als einem Type-Knoten auf. Bei diesen und anderen ACs, besonders einem RS, muß unter Umständen die Instanziierbarkeit entfernt werden und eine Klasse als reiner Behälter interpretiert werden. Dies ist mit einer abstrakten Klasse möglich. Die in ihr enthaltenen Elemente können nur verwendet werden, wenn sie der Klasse – nicht der Instanz – zugeordnet werden. D. h. es müssen Klassenoperationen und Klassenattribute verwendet werden. Insgesamt betrachtet ergab der Versuch, mit Klassen semantisch möglichst äquivalent ACs abzubilden, eine sehr komplizierte Modellierung, die für jede AC-Art andere Regeln vorsah. Eine wirklich semantisch nahe und weitgehend unumstrittene Modellierung wurde nur für ADTs mit einem Type-Knoten gefunden, wobei auch hier im Detail Strittigkeiten auftraten. Zudem war die resultierende Modellierung für Benutzer, die mit Objektorientierung und der UML keine Erfahrungen haben, schwer verständlich. Als interessante Alternative stellten sich im Verlauf der Untersuchungen Utilities heraus, die ursprünglich nur für die Modellierung von ACs der Art RS angedacht waren.

Modellierung mit Utilities

Eine Utility-Klasse oder kurz Utility ist eine in der UML definierte spezielle Klasse, die eine Gruppierung von globalen Variablen und Unterprogrammen darstellt. Von Utilities können keine Instanzen angelegt werden. (vgl. [uml1.3], S. 3-52) Ein Utility wird als Klasse mit dem Stereotyp `<<utility>>` modelliert und – wie Abbildung 4-6 zeigt – auch so dargestellt.

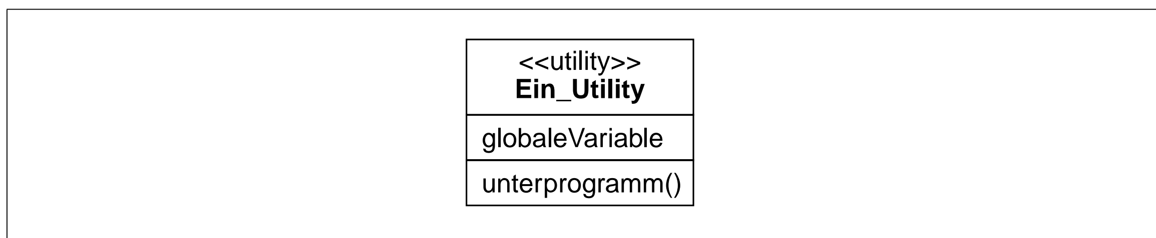


Abbildung 4-6: Darstellung einer Utility-Klasse in UML

Utilities werden normalerweise relativ selten eingesetzt, da die Verwendung von globalen Variablen beim Entwurf objektorientierter Systeme im allgemeinen vermieden werden sollte. Sie ermöglichen jedoch eine semantisch den ACs sehr nahe kommende Modellierung, bei gleichzeitiger großer Nähe zum Quelltext, bzw. der tatsächlichen Implementierung.

4.2.2 Modellierung der in ACs enthaltenen Elemente

Neben den für die Abbildung von ACs verwendeten UML-Sprachmitteln, müssen passende Möglichkeiten für die Modellierung der in ihnen enthaltenen Elemente gefunden werden. Grundsätzlich könnten alle Arten von Kanten mit Abhängigkeiten modelliert werden, alle Arten von Knoten als Klassen. Dies würde der im Kapitel „Prinzipielle Modellierung einer Architekturbeschreibung“ auf Seite 30 bereits disku-

tierten Modellierung des RFGs mit Objekten ähneln und wurde aus den selben Gründen verworfen. Eine solche Modellierung sollte nur in Fällen zur Anwendung kommen, in denen keine bessere Alternative existiert, z. B. bei Call-Kanten.

Zunächst werden nur Knoten behandelt, die tatsächlich Teil eines ACs sind. Sie werden prinzipiell dem die AC abbildenden UML-Element als Teilelement zugeordnet. Knoten, die nicht in einem AC enthalten sind, werden danach diskutiert.

Modellierung von Typen und ihrer Teilkomponenten

Da derzeit im RFG nur eingeschränkte Informationen über die Struktur eines Typs vorliegen, kann nur eine relativ grobe Abbildung auf UML erfolgen. Ein mögliches Sprachelement wären Datentypen (im Metamodell durch die Metaklasse `DataType` repräsentiert). Sie werden allerdings ausschließlich verwendet, um primitive Typen oder Aufzählungstypen zu modellieren. Für zusammengesetzte Typen werden in der UML dagegen Klassen eingesetzt. Die Teilkomponenten zusammengesetzter Typen – im RFG in Form von Member-Knoten enthalten – können über Attribute oder Assoziationen abgebildet werden, deren Typ bzw. Ziel durch Member_Of_Type-Kanten angegeben wird. Attribute und Assoziationen werden im Kapitel „Modellierung von Variablen und Konstanten“ auf Seite 48 genauer behandelt.

Sollten im RFG einmal genauere Informationen über die Art und Struktur eines Typs enthalten sein, wären differenziertere Abbildungen möglich. Dazu werden im folgenden einige Überlegungen angestellt:

- **Zusammengesetzte Typen** können wie oben angegeben abgebildet werden. Type-Knoten ergeben Klassen, Member-Knoten in diesen enthaltene Attribute bzw. Assoziationen. Abbildung 4-7 (a) zeigt ein Beispiel.
- **Vereinigungstypen** sind prinzipiell analog behandelbar. Allerdings ergeben in ihnen enthaltene Member-Knoten Kompositionsassoziationen, die semantisch weitgehend mit Attributen gleichzusetzen sind. Zwischen den Assoziationen besteht eine Xor-Einschränkung. Diese besagt, daß zu jedem Zeitpunkt immer nur genau eine der Assoziationen gültig ist. (vgl. [uml1.3], S. 2-20) Dies entspricht der Semantik eines Vereinigungstyps recht gut. Ein Beispiel ist in Abbildung 4-7 (b) abgebildet. Eine andere Möglichkeit wäre es, bestimmte Vereinigungstypen über Vererbungshierarchien zu modellieren, wie dies z. B. *Bloch* vorschlägt ([Blo00]). Es ist bisher aber noch nicht möglich, zu erkennen, für welche Vereinigungstypen im RFG dies sinnvoll wäre.

- **Aufzählungstypen** werden in UML als Datentypen mit dem Stereotyp `<<enumeration>>` modelliert. Zusätzlich werden die möglichen Werte aufgelistet, wie dies Abbildung 4-7 (c) zeigt.
- **Primitive Typen** werden ebenfalls mit Datentypen realisiert. Dabei wird der Wertebereich des Typs angegeben. Ein Beispiel ist Abbildung 4-7 (d). (vgl. [Boo99], S. 58 f.) Im UML-Standard selbst werden sie zudem mit dem Stereotyp `<<primitive>>` versehen.

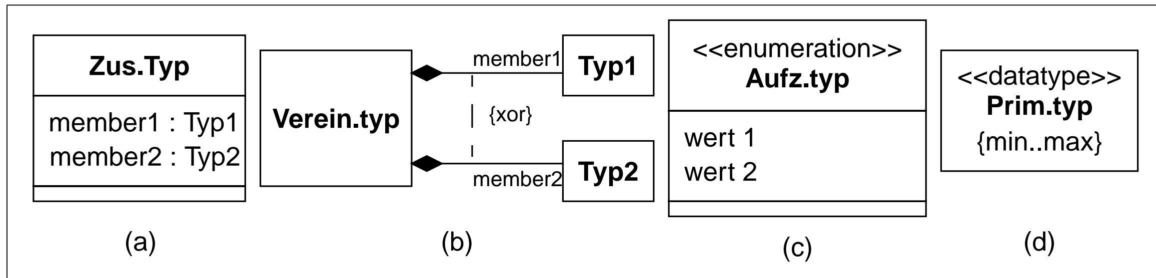


Abbildung 4-7: Verfeinerte Abbildung von Typen

Beziehungen zwischen Typen stellen die Kanten `Is_Part_Type_Of_Edge` und `Typedefed_By_Edge` dar. Sie können als Abhängigkeiten zwischen Typen abgebildet werden.

Ebenfalls untersucht wurde die Alternative, Is_Part_Of_Type-Kanten als Aggregationen zwischen den beteiligten Typen abzubilden. Eine Aggregation legt fest, daß ein Element Teilelement des anderen ist, was zunächst als eine passende Abbildung einer Is_Part_Of_Type-Kante erscheint. Genauer betrachtet besagt eine Is_Part_Of_Type-Kante jedoch nur, daß der 'Teiltyp' in der Deklaration des neuen Typs verwendet wird. Wie und weshalb dies erfolgt, ist nicht ersichtlich. Eine Aggregation in UML macht dagegen eine Aussage über die Verwendung eines Typs, bzw. *seiner Exemplare* auf semantisch höherer Ebene als der Typstruktur, weshalb diese Möglichkeit verworfen wurde.

Falls für eine Typedefed_By-Kante ermittelt werden kann, ob sie einen Alias oder die Deklaration eines neuen Typs darstellt, ist ebenfalls eine alternative Modellierung möglich. Im Falle eines Alias können beide Typen verschmolzen werden, wobei gegebenenfalls ein Hinweis hierauf gegeben werden sollte. Im Falle eines neuen Typs kann dieser die Struktur des ursprünglichen Typs übernehmen. Enthielt dieser z. B. Member-Knoten, können diese kopiert werden. Bisher kann nicht entschieden werden, welcher Fall jeweils vorliegt.

Modellierung von Variablen und Konstanten

UML enthält zwei Sprachelemente, die sich für die Modellierung von Variablen und Konstanten anbieten: Attribute und Assoziationen.

Attribute stellen einen benannten Teil des Zustands beispielsweise einer Klasse dar. Sie definieren, welche Werte dieser Teilzustand annehmen kann und welche Operationen anwendbar sind. Dazu erhalten sie einen Typ. Attribute entsprechen Variablen bzw. Konstanten relativ gut. (vgl. [uml1.3], S. 2-23 f.)

Assoziationen definieren dauerhafte Verbindungen zwischen Elementen, z. B. Klassen, über die diese aufeinander zugreifen können. Diese Verbindungen können zwischen zwei oder mehr Elementen bestehen. Unterarten der Assoziation sind die bereits angesprochene Aggregation und die Komposition. Eine Aggregation ist eine dauerhafte Verbindung zwischen zwei Elementen, von denen das eine Teil des anderen ist. Weitere semantische Festlegungen trifft UML hier nicht. Eine stärkere Form der Aggregation ist die Komposition. Sie legt zusätzlich fest, daß das Teilelement in höchstens einem Ganzen enthalten sein darf und seine Lebensdauer mit diesem verknüpft ist. Praktisch bedeutet dies, daß das umschließende Element für die Freigabe des Teilelements verantwortlich ist. Typischerweise wird es das Teilelement auch selbst erzeugen. (vgl. [uml1.3], S. 2-19 f.)

Obwohl Attribute und Assoziationen auf abstrakter Ebene durchaus unterschiedlich eingesetzt werden, ist eine Unterscheidung zwischen ihnen auf Implementierungsebene teilweise schwierig. Dies resultiert daraus, daß eine Assoziation normalerweise mit Hilfe von Attributen realisiert wird. Beispielsweise wird die in Abbildung 4-8 (a) gezeigte ungerichtete Assoziation zwischen den Klasses *Klasse1* und *Klasse2* bei der Implementierung je ein Attribut in jeder Klasse ergeben. Beide Attribute stellen Referenzen auf Instanzen der jeweils anderen Klasse dar, z. B. Zeiger, wie dies Abbildung 4-8 (b) zeigt. Es ist im allgemeinen aus der Implementierung später nicht mehr ersichtlich, ob es sich bei diesen Attributen auch auf abstrakter Ebene um Attribute oder um eine Assoziation handelte. Die Gruppierung der beiden Attribute

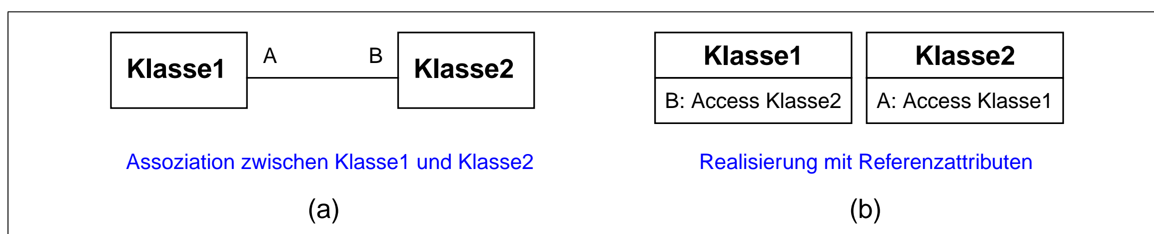


Abbildung 4-8: Realisierung einer ungerichteten Assoziation mit Attributen

und die Art der Assoziation (Aggregation, Komposition) geht im Normalfall verloren. Es ist theoretisch möglich, jedes Attribut als eigene Assoziation darzustellen. Im obigen Fall würden zwei gerichtete Assoziationen erstellt. *Together Control Center* geht genau so vor. Die entstehenden Darstellungen können sehr unübersichtlich sein. Im folgenden wird stellvertretend von einer Modellierung über Attribute ausgegangen. Die beschriebenen Sachverhalte können aber sinngemäß auf Assoziationen umgesetzt werden.

Der Typ eines Attributs kann aus einer Variable_Of_Type-Kante abgeleitet werden. Reference-Kanten, wie z. B. Variable_Set- oder Variable_Use-Kanten können über spezielle Abhängigkeiten modelliert werden.

Modellierung von Unterprogrammen

Für die Abbildung von Unterprogrammen bieten sich in UML unmittelbar Methoden (`Method`) an. Eine Methode spezifiziert einen Algorithmus oder eine Prozedur, der bzw. die die Ergebnisse einer Operation beeinflußt. Methoden implementieren Operationen. Sie benötigen daher zwingend zu ihnen passende Operationen. Eine Methode

ohne Operation ist in UML unzulässig. Operationen definieren Dienste, die bei einem Objekt angefordert werden können, um das Verhalten des Systems zu beeinflussen. (vgl. [uml1.3], S. 2-38, S. 2-41 f.)

Parameter_Of_Type-Kanten können als Parameter der Methode bzw. Operation abgebildet werden. Name und genauer Typ des Parameters gehen allerdings verloren. Außerdem werden mehrere Parameter des gleichen Typs auf die selbe Kante abgebildet. Die exakte Signatur der Methode kann daher nicht ohne Rückgriff auf den Quelltext bzw. die IML-Darstellung ermittelt werden. Der Rückgabetyp einer Methode wird mit Hilfe ausgehender Return_Type-Kanten ermittelt. Die weiteren mit Unterprogrammen in Beziehung stehenden Call-, Actual_Parameter-, Local_Var_Of_Type- und Function_Address-Kanten können als Abhängigkeiten abgebildet werden, was später noch genauer diskutiert wird.

4.2.3 Modellierung von Elementen, die keiner AC angehören

Einen Sonderfall bilden Knoten aus dem Basis-RFG, die in keiner AC enthalten sind. Sie stellen Systemelemente aus der Base-View dar, die nicht in der User-View enthalten waren, bzw. dort nicht Teil einer AC waren, d. h. noch nicht in die erkannte Architektur integriert wurden. Ursache dafür kann eine noch nicht vollständig abgeschlossene oder fehlerhafte AC-Erkennung sein. Eine Modellierung dieser Knoten in UML ist sehr schwierig.

Eine Möglichkeit besteht darin, eine spezielle AC in den RFG einzufügen, die alle eigentlich nicht zugeordneten Knoten für die Dauer der Modellierung in UML aufnimmt. Dadurch kann der Benutzer die UML-Darstellung auch nutzen, um eigene Zuordnungen vorzunehmen. Alternativ ist es möglich, für jeden freien Knoten eine eigene AC zu erzeugen. Dieses Vorgehen führt allerdings u. U. zu einer großen Anzahl tatsächlich nicht vorhandener ACs.

Besser ist es, solche Knoten zu ignorieren. Dies ist zulässig, da sie nicht Teil der zu modellierenden erkannten Architektur sind, obwohl sie in der Architektur-View enthalten sind. Diese Möglichkeit wurde in dieser Arbeit gewählt. Insbesondere im Falle einer noch nicht vollständig abgeschlossenen AC-Erkennung wird sie den Wünschen des Anwenders in den meisten Fällen am besten entsprechen.

Eine weitere Alternative besteht darin, beide Vorgehensweisen anzubieten und den Benutzer entscheiden zu lassen, welche verwendet werden soll.

4.3 Vorgeschlagene Modellierung

Aus den vorliegenden Möglichkeiten muß nun ein schlüssiges Konzept für die Modellierung von ACs und ihren Teilelementen in der UML erarbeitet werden.

4.3.1 Transformation nach UML

Eine grobe Übersicht über die Transformation nach UML ist in Tabelle 4-3 enthalten.

Knoten und Kanten im RFG	Resultierendes UML-Sprachelement
Atomic_Component-Knoten mit Part_Of-Kanten	Utility-Klasse mit enthaltenen Elementen
Type-Knoten mit Enclosing-Kanten zu Member-Knoten	Klasse mit enthaltenen Attributen
Variable- oder Constant-Knoten mit Type_Of-Kante	Attribut des angegebenen Typs
Subprogram-Knoten mit Parameter_Of_Type- und Return_Type-Kanten	Operation und Methode mit entsprechenden Parametern und Rückgabetyp
andere Kante, z. B. Reference-Kante	Abhängigkeit mit entsprechendem Stereotyp

Tabelle 4-3: Transformation ACs und Basis-RFG nach UML

Atomare Komponenten

Die derzeit beste Möglichkeit, ACs in UML abzubilden, sind Utilities. Sie kommen ACs semantisch sehr nahe, die Abbildung ist einfach und einheitlich durchzuführen. Um in der UML unerfahrenen Benutzern das Verständnis zu erleichtern, wird ein neuer Stereotyp `<<ac>>` definiert. Dieser erbt alle Eigenschaften von `<<utility>>`. Mit diesem Stereotyp versehene Klassen stellen daher spezielle Utilities dar, die zur Modellierung von ACs verwendet werden. In der UML erfahrenen Benutzern wird dies den unmittelbaren Bezug zu Utilities erschweren. Das grundsätzliche Verständnis der Modellierung leidet allerdings nur sehr wenig und ein kurzer Blick in die Dokumentation sollte ausreichen, den Zusammenhang klarzumachen.

Die genaue Art der AC (ADT, ADO, HC oder RS) kann über speziellere, von `<<ac>>` erbende Stereotypen modelliert werden. Abbildung 4-9 zeigt die entstehende Vererbungshierarchie zwischen den Stereotypen.

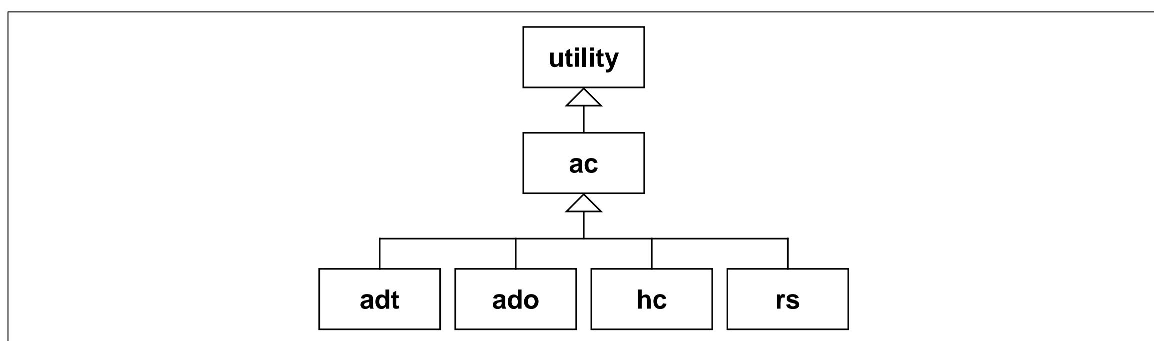


Abbildung 4-9: Vorgeschlagene Stereotyphierarchie

Ist die genaue Art unbekannt, wird `<<ac>>` verwendet. Es existieren zwei Möglichkeiten, Stereotypen in UML darzustellen:

1. *textuell* oder
2. *graphisch* in Form eines eigenen Symbols.

Abbildung 4-10 zeigt ein Beispiel für beide Formen.

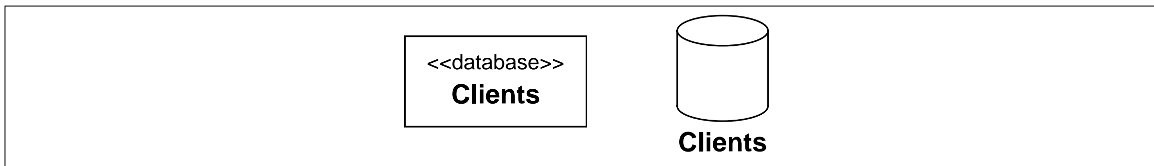


Abbildung 4-10: Verschiedene Darstellungsformen für Stereotypen am Beispiel einer Datenbank

Es liegt in der Verantwortung des verwendeten UML-Werkzeugs, welche Form gewählt wird und wie neue graphische Symbole definiert werden, d. h. die Angabe eines graphischen Symbols für einen Stereotyp kann derzeit nur werkzeugabhängig erfolgen. Zu erwähnen ist die Möglichkeit, die Art einer AC über farbliche Hervorhebungen mit Hilfe entsprechend gefärbter Stereotypsymbole zu visualisieren. Ob solche graphischen Symbole mit vorgegebener Farbe in der UML überhaupt zulässig sind, ist derzeit aber nicht definiert. Im Rahmen dieser Arbeit werden keine konkreten Vorschläge für eine graphische Darstellungsform gemacht, sondern Stereotypen prinzipiell textuell dargestellt.

Der Name der Utility-Klasse ergibt sich aus dem in der Node_Id enthaltenen Attribut `Object_Name`. Die weiteren Teile der Node_Id sowie weitere Attribute ergeben dem Utility zugeordnete zusätzliche Werte in Form von TaggedValues. Dieses Vorgehen wird analog auch bei allen anderen Knotentypen verfolgt. Alternativ wäre es möglich, die komplette Node_Id als Name der Klasse zu wählen. Dies würde aber zu teilweise sehr langen Namen führen, die eine unübersichtliche Darstellung zur Folge hätten. Ein Beispiel für die Modellierung einer – in diesem Fall leeren – AC zeigt Abbildung 4-11.

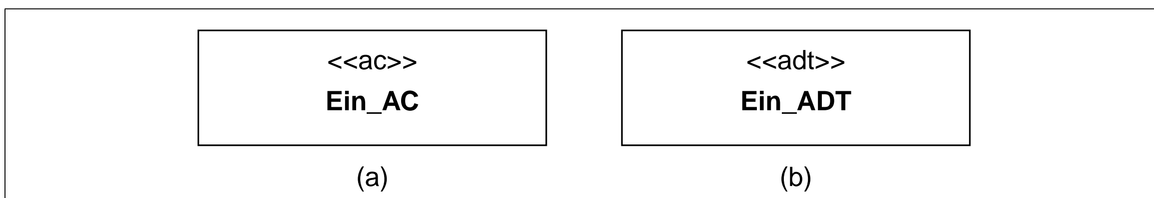


Abbildung 4-11: Modellierung eines Atomic_Component-Knotens in UML

Die Sichtbarkeit eines Atomic_Component-Knotens im übergeordneten Kontext, beispielsweise dem den Knoten enthaltenden Subsystem, kann auf die Sichtbarkeit der Utility-Klasse in ihrem Kontext abgebildet werden. Allerdings ist es prinzipiell schwierig, die Semantik der Sichtbarkeit einer Utility-Klasse zu definieren, da Utilities definitionsgemäß Gruppierungen global sichtbarer Elemente – wie z. B. globaler Variablen – sind und nicht selbst konkret im System vorliegen. Aus diesem Grund ist es auch schwierig, die Sichtbarkeitsinformationen der in der AC enthaltenen Knoten geeignet abzubilden. Eine Lösung könnte es sein, die Semantik der Utility-Klasse mittels der eingesetzten Stereotypen zu verfeinern, um private Utilities zu ermöglichen. Dies könnte allerdings zu Verwirrung bei UML-erfahrenen Benutzern führen.

Derzeit werden in Bauhaus noch keine Sichtbarkeitsinformationen ermittelt, d. h. alle Knoten haben prinzipiell eine unbekannte Sichtbarkeit, was als globale Sichtbarkeit interpretiert werden muß, da dieser Wert in UML nicht existiert. Aus diesem Grund werden im Rahmen dieser Arbeit die entsprechenden Attribute in ACs und allen direkt in ihnen enthaltenen Elementen, wie z. B. Typen, ignoriert und globale Sichtbarkeit (`public`) angenommen.

Zusätzlich ist es in der UML möglich, die Persistenz einer Klasse anzugeben. Darunter wird verstanden, ob der Wert einer Klasse zwischen Programmausführungen gespeichert wird und erhalten bleibt (angezeigt durch den Wert `persistent`) oder am Ende einer Programmausführung verloren geht (`transient`). Standardmäßig wird letzteres angenommen Der Aspekt der Persistenz ist im RFG bisher nicht enthalten, weshalb der Standardfall verwendet wird. (vgl. [uml1.3], S. 2-28)

Die derzeit vorgeschlagene Modellierung von ACs in UML wird in Listing 4-1 in Form von Pseudocode angegeben.

```
MapAtomic_Component_NodeToUML
  in anAc: Atomic_Component_Node;
  inout umlModel: Model;
  return c: Class := new Class;
begin
  switch anAc.getComponent_Classification
    case ADT: c.setStereotype(adt);
    case ADO: c.setStereotype(ado);
    case HC: c.setStereotype(hc);
    case RS: c.setStereotype(rs);
    default: c.setStereotype(ac);
  end switch;
  c.setName(anAc.getNode_Id.getObject_Name);
  c.addTaggedValue(anAc.getNode_Id.getFilename);
  c.addTaggedValue(anAc.getNode_Id.getProgram_Name);
  c.addTaggedValues(anAc.getOtherAttributes);
  c.setVisibility(public);
  c.setPersistence(transient);
  foreach Part_Of_Edge p in anAc.getIncomingEdges do
    c.add(MapProgramming_Entity_NodeToUML(p.getBegin,umlModel));
  end foreach;
end;
```

Listing 4-1: Transformation eines AC-Knotens nach UML

Typen und ihre Teilkomponenten

Wie bereits diskutiert, ist derzeit nur eine relativ grobe Abbildung möglich. **Type-Knoten** werden als Klassen abgebildet. Um diese besser von Utility-Klassen zu unterscheiden, die aus ACs resultieren, werden sie mit dem Stereotyp `<<typenode>>` versehen. Der eigentlich näherliegende Stereotyp `<<type>>` kann nicht verwendet werden, da er in UML eine vordefinierte andere Bedeutung besitzt (siehe [uml1.3], S. 3-44 ff.). Die Node_Id des Type-Knotens wird analog zum Vorgehen bei AC-Knoten abgebildet. Die neue Klasse wird in das Utility eingebettet, das aus der den Typ ent-

haltenden AC erzeugt wird. Dabei sind verschiedene Darstellungen möglich, die Abbildung 4-12 zeigt. Welche dieser Darstellungen gewählt wird, liegt in der Verantwortung des verwendeten UML-Werkzeugs.

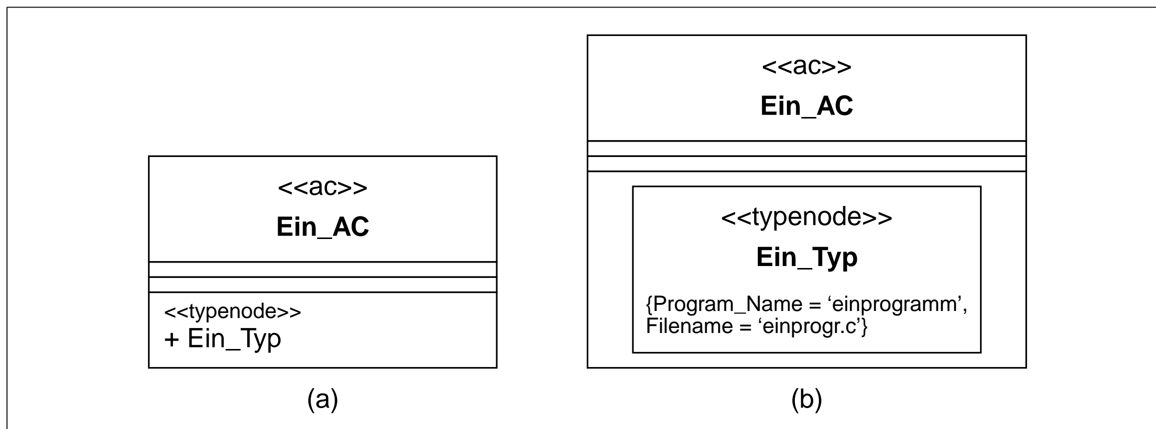


Abbildung 4-12: Verschiedene Darstellungen eingebetteter Typen

Listing 4-2 zeigt die Modellierung von Typknoten in Pseudocode.

```
MapType_NodeToUML
  in type: Type_Node;
  inout umlModel: Model;
  return tClass: Class := new Class;
begin
  tClass.setStereotype(type_node);
  tClass.setName(type.getNode_Id.getObject_Name);
  tClass.addTaggedValue(type.getNode_Id.getFilename);
  tClass.addTaggedValue(type.getNode_Id.getProgram_Name);
  tClass.addTaggedValues(type.getOther_Attributes);
  tClass.setVisibility(public);
  foreach Edge e in type.incomingEdges do
    MapEdgeForTypeNodeToUML(e, tClass, umlModel);
  end foreach;
end;
```

Listing 4-2: Transformation eines Type-Knotens nach UML

Is_Part_Of_Type- und **Typedefed_By-Kanten** werden als Usage-Abhängigkeiten modelliert. Usage-Abhängigkeiten stellen spezielle Abhängigkeiten dar, bei denen das Quellelement die Anwesenheit des Zielelements dauerhaft benötigt um zu funktionieren. Es handelt sich also um eine relativ konkrete Abhängigkeit, die zur Laufzeit des Systems vorliegt. Ein Beispiel wäre ein Unterprogramm, das ein anderes aufruft, um seine eigene Funktion zu erfüllen. Eine Is_Part_Of_Type-Kante ergibt eine Usage-Abhängigkeit mit Stereotyp `<<is-part-of-type>>`, eine Typedefed_By-Kante eine mit dem Stereotyp `<<typedefed-by>>` versehene Usage-Abhängigkeit.

Dies zeigt Listing 4-3. Alternativ wäre es möglich, Is_Part_Of_Type-Kanten zu igno-

```
MapEdgeForTypeToUML
  in edge: Edge;
  inout type: Class;
  inout umlModel: Model;
begin
  depend := new Usage;
  depend.setSupplier(type);
  depend.setClient(umlModel.getUMLfor(e.getBegin));
  switch typeof edge
    case Is_Part_Of_Type_Edge:
      depend.setStereotype(is-part-of-type);
      umlModel.add(depend);
    case Typedefed_By_Edge:
      depend.setStereotype(typedefed-by);
      umlModel.add(depend);
    case Enclosing_Edge:
      type.add(MapMember_NodeToUML(edge.getBegin, umlModel));
  end switch;
end;
```

Listing 4-3: Transformation von Is_Part_Of_Type- und Typedefed_By-Kanten nach UML

rieren, um eine übersichtlichere Darstellung zu erhalten, da die durch sie vermittelte Information für den Benutzer relativ unwichtig ist und über den Aufbau der Typen (dargestellt mit Member-Knoten) implizit bereits enthalten ist.

Die über **Enclosing-Kanten** mit dem Typknoten verbundenen **Member-Knoten** werden analog zu Variablen als Attribute modelliert, um eine einheitliche Abbildung auf UML zu erhalten. Die Gründe für die Wahl von Attributen sind im nächsten Teilkapitel enthalten. Node_Id und weitere Attribute des Member-Knotens ergeben – analog zu den bisherigen Knotentypen – den Namen des UML-Attributs und ihm zugeordnete TaggedValues. Die Sichtbarkeit eines Member-Knotens kann direkt als UML-Sichtbarkeit abgebildet werden, falls sie bekannt ist. Ist sie unbekannt, muß sie als global sichtbar definiert werden. Um eine eindeutige Zuordnung auch in der Gegenrichtung zu ermöglichen, wird dem Attribut in diesem Fall zudem die Beschränkung `{unknown_visibility}` zugewiesen. Die Modellierung der Sichtbarkeit zeigt Tabelle 4-4.

Knotensichtbarkeit	**Sichtbarkeit in UML**
`Public_Visibility`	`public`
`Restricted_Visibility`	`protected`
`Private_Visibility`	`private`
`Unknown_Visibility`	`public` + `{unknown_visibility}`

Tabelle 4-4: Zuordnung von Knotensichtbarkeiten zu Sichtbarkeiten in der UML

In der konkreten Implementation eines Systems kommt es vor, daß eigentlich bestehende Abstraktionsbarrieren durchbrochen werden, z. B. indem ein Unterprogramm auf einen ADT direkt zugreift, obwohl es nicht im ADT enthalten ist. Dadurch kann der Fall auftreten, daß auf als privat deklarierte Member-Knoten von außerhalb zugegriffen wird, was zu einem ungültigen UML-Modell führen würde. Das dies nicht der Fall ist, muß von der Architekturerkennung abgesichert werden. Handelt es sich um eine aus bestimmten Gründen nötige Durchbrechung der Abstraktionsbarriere, sollte entweder der Member-Knoten als global sichtbar definiert werden, oder gegebenenfalls über eine neu zu definierende Kante im RFG angezeigt werden, daß der Quellknoten Zugriff auf einen privaten Knoten besitzt. Dies könnte in UML als Abhängigkeit mit dem Stereotyp `<<friend>>` modelliert werden. Entsprechend wäre es möglich, fehlerhafte Abstraktionsdurchbrüche im RFG als Kanten zu modellieren und diese in UML als Abhängigkeit mit Stereotyp `<<violates>>` abzubilden.

Es kann vorkommen, daß der RFG keine Information darüber enthält, welchen Typ eine durch einen Member-Knoten modellierte Teilkomponente hat. Beispielsweise ist dies bei Teilkomponenten der Fall, die einen primitiven Typ haben. Primitive Typen ergeben keine eigenen Knoten im RFG. In der UML ist es nicht zulässig, ein Attribut ohne festgelegten Typ anzugeben. Es muß daher bei der Transformation in die UML auch für Teilkomponenten, deren Typ nicht bekannt ist, ein Typ gewählt werden. Zu diesem Zweck wird in das erzeugte UML-Modell ein temporärer Datentyp `unknown` eingefügt. Ist der Typ einer Teilkomponente nicht bekannt, bekommt das entsprechende Attribut diesen Typ zugewiesen. Dies verhindert, daß ein UML-Werkzeug einen Standardtyp wählt (*Together Control Center* verwendet hierfür z. B. den primitiven Datentyp `int`).

Ein Beispiel für die Modellierung von drei Member-Knoten *x*, *y* und *next* in einem Typ *Ein_Typ* ist in Abbildung 4-13 dargestellt.

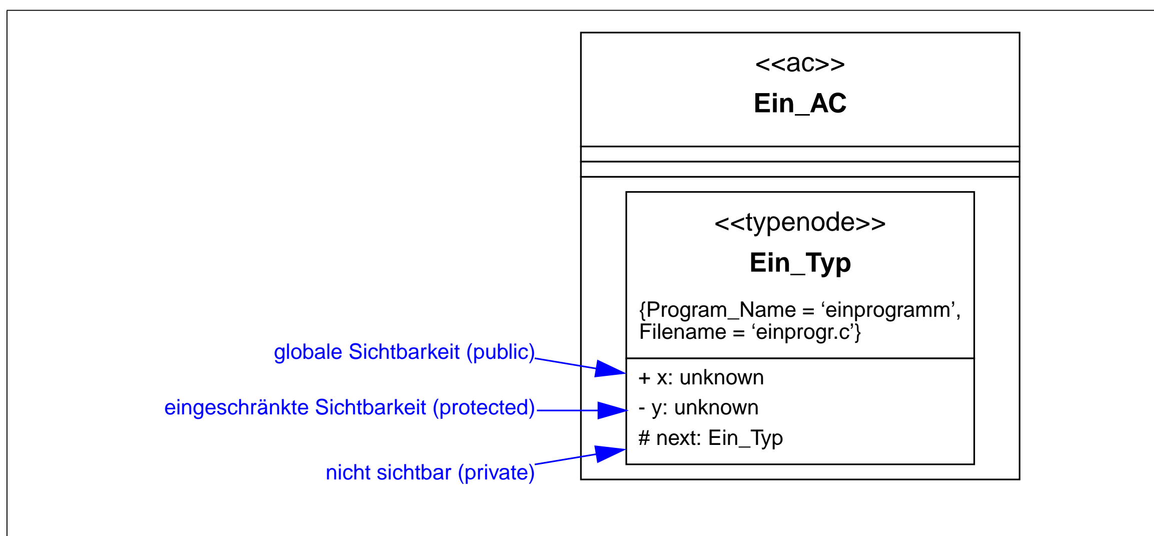


Abbildung 4-13: Modellierung eines Typs in UML

Die Multiplizität der aus den Member-Knoten abgeleiteten Attribute kann nicht gesetzt werden, da diese Information nicht im RFG enthalten ist.

Listing 4-4 zeigt die im Rahmen der Modellierung eines Member-Knotens durchgeführten Schritte.

```
MapMember_NodeToUML
  in memb: Member_Node;
  inout umlModel: Model;
  return mAttr: Attribute := new Attribute;
begin
  mAttr.name := memb.getNode_Id.getObject_Name;
  mAttr.addTaggedValue(memb.getNode_Id.getFilename);
  mAttr.addTaggedValue(memb.getNode_Id.getProgram_Name);
  mAttr.addTaggedValues(memb.getOtherAttributes);
  if memb.getVisiblity = unknown then
    mAttr.setVisibility(public);
    mAttr.addConstraint(unknown_visibility);
  else
    mAttr.setVisibility(memb.visibility);
  end if;
  if exists Member_Of_Type_Edge t in memb.getOutgoingEdges then
    mAttr.setType(umlModel.getUMLfor(t.end));
  else
    mAttr.setType(unknown);
  end if;
end;
```

Listing 4-4: Transformation eines Member-Knotens nach UML

Variablen und Konstanten

Bei der Transformation von Variable- und Constant-Knoten auf UML-Sprachkonstrukte muß zwischen der Modellierung als Attribut und der Modellierung als Assoziation entschieden werden. Wie bereits in Kapitel „Modellierung von Variablen und Konstanten“ auf Seite 48 diskutiert, ist es sehr schwierig, aus der Implementierung eines Systems darauf zu schließen, welche dieser Möglichkeiten im konkreten Fall semantisch besser geeignet ist. Es wäre möglich, den RFG um die entsprechende Information zu erweitern, allerdings sollte dies im Rahmen einer umfassenden Erweiterung des RFGs um das Konzept der Assoziation erfolgen. Ein einfaches neues Attribut in den entsprechenden Knoten reicht nicht aus, um es zu ermöglichen, daß mehrere Variablen zu einer Assoziation gruppiert werden. Im Rahmen dieser Arbeit wurde diese Möglichkeit nicht weiter verfolgt und stattdessen eine Modellierung über Attribute gewählt. Dafür sprechen Gründe, die weniger auf semantischer Ebene liegen, sondern vielmehr mit der Visualisierung zusammenhängen.

Für in der UML unerfahrene Benutzer ist eine Modellierung von Variablen und Konstanten über Attribute leichter verständlich, da Attribute in einer Form dargestellt werden, die der üblichen Notation von Variablen und Konstanten in einer Programmiersprache entspricht. Dazu kommt ein weiterer, aus den in Abbildung 4-12 bekannten Darstellungsalternativen für eingebettete Typen resultierender Vorteil von Attributen. Wie Abbildung 4-14 (a-2) zeigt, kann es vorkommen, daß Assoziatio-

nen zwischen Klassen und eingebetteten Typen nicht dargestellt werden, obwohl sie im UML-Modell vorliegen. In Abbildung 4-14 (b-1) und (b-2) ist dagegen die als Attribut modellierte Variable immer sofort erkennbar.

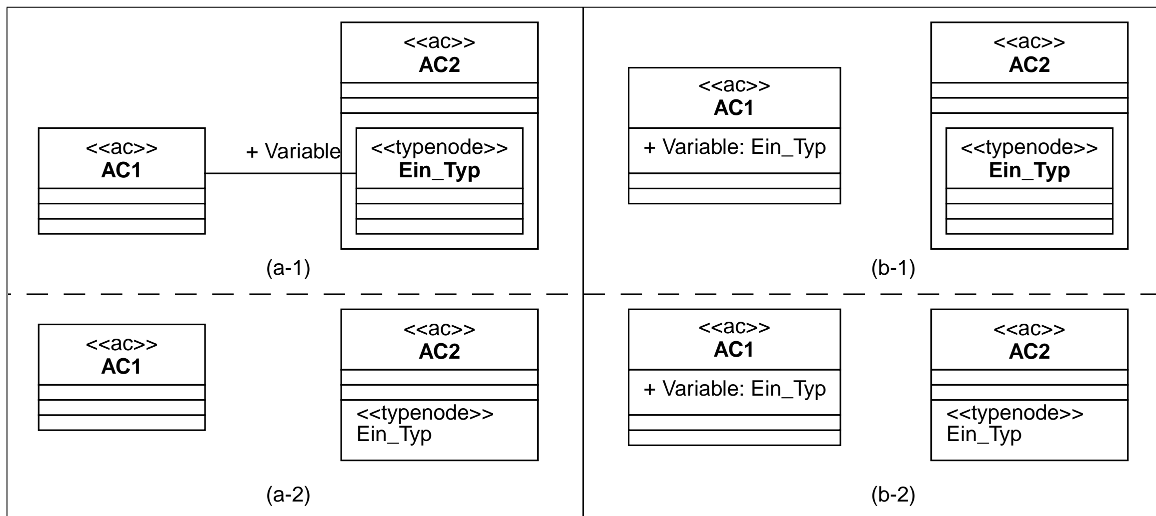


Abbildung 4-14: Verschiedene Darstellungen eingebetteter Typen

Konstanten werden zusätzlich als `{frozen}` gesetzt, eine Einschränkung, die besagt, daß das betreffende Attribut nach dem erstmaligen Setzen seines Werts nicht mehr geändert werden darf. (vgl. [uml1.3], S. 2-24)

Der genaue Ablauf der Transformation kann Listing 4-5 entnommen werden.

```
MapObject_NodeToUML
  in var: Object_Node;
  inout umlModel: Model;
  return attr: Attribute := new Attribute;
begin
  attr.setName(var.getNode_Id.getObject_Name);
  attr.addTaggedValue(var.getNode_Id.getFilename);
  attr.addTaggedValue(var.getNode_Id.getProgram_Name);
  attr.addTaggedValues(var.getOther_Attributes);
  attr.setVisibility(public);
  if exists Variable_Of_Type_Edge e in var.getOutgoingEdges then
    attr.setType(umlModel.getUMLfor(e.getEnd));
  else
    attr.setType(unknown);
  end if;
  if var typeof Constant_Node then
    attr.setChangeability(frozen);
  end if;
  foreach Enclosing_Edge e in var.getIncomingEdges do
    attr.add(MapMember_NodeToUML(e.begin, umlModel));
  end foreach;
end;
```

Listing 4-5: Transformation von Variable- und Constant-Knoten nach UML

Variable_Of_Type-Kanten werden analog zu den im vorherigen Teilkapitel besprochenen Member_Of_Type-Kanten als Typ des resultierenden Attributs abgebildet. Ist der Typ nicht bekannt, wird er auf den Datentyp `unknown` gesetzt.

Same_Expression-Kanten sind für die Darstellung einer Architektur nicht von Bedeutung, weshalb sie ignoriert werden. Alternativ wäre eine Modellierung als Abhängigkeit mit dem Stereotyp `<<same-expression>>` möglich.

Unterprogramme

Subprogram-Knoten werden als Paare von Methoden und ihnen zugehörige Operationen modelliert. Die UML ermöglicht für diese die Angabe verschiedener Informationen, die bisher im RFG nicht enthalten sind:

Bei einer Operation kann festgelegt werden, wie der gleichzeitige Aufruf der Operation durch nebenläufige Prozesse behandelt werden soll. Mögliche Werte sind

- `sequential`, was besagt, daß Prozesse sich beim Aufruf der Operation koordinieren müssen, da mehrere gleichzeitige Aufrufe zu undefiniertem Verhalten führen,
- `guarded`, wodurch die Operation die Koordination mehrerer Aufrufe selbst vornimmt, vergleichbar einem Monitor und
- `concurrent`, was eine Operation anzeigt, die zu jedem Zeitpunkt mehrfach ausgeführt werden kann.

Aufgrund fehlender Informationen wird bisher immer der Standardwert `sequential` verwendet. Die weiteren bei Operationen möglichen Angaben beziehen sich auf Aspekte, die erst bei Vererbung von Bedeutung sind, z. B. ob eine Operation in einer Kindklasse überschrieben werden darf. Sie werden daher auf ihre jeweiligen Standardwerte gesetzt. (vgl. [uml1.3], S. 2-42)

Methoden enthalten unter anderem Informationen darüber, in welcher Programmiersprache sie geschrieben wurden und können auch ihren eigentlichen Programmtext angeben. Soweit der Quelltext des Systems vorliegt, könnten diese Informationen übernommen werden. Bisher geschieht dies nicht.

Die Klassifikation eines Unterprogramms (Konstruktor, Destruktor, usw.) kann als entsprechende Beschränkung modelliert werden. Für Unterprogramme, die den Zustand des Systems nicht ändern, ist in UML bereits die Beschränkung `{query}` definiert (vgl. [uml1.3], S. 2-25). Alle hier verwendeten Beschränkungen zeigt Tabelle 4-5.

Unterprogrammart	UML-Beschränkung
Konstruktor	`{constructor}`
Destruktor	`{destructor}`
abänderndes Unterprogramm	`{modifier}`
nur lesendes Unterprogramm	`{query}`

Tabelle 4-5: Modellierung der Klassifikation eines Unterprogramms

Bei der Modellierung von Member-Knoten wurde deren Sichtbarkeit übernommen. Dies führt dazu, daß Operationen, die nicht in der aus dem umschließenden Type-Knoten abgeleiteten Klasse enthalten sind, keinen Zugriff auf private Member-Knoten, bzw. die resultierenden Attribute haben. In den meisten Fällen ist dies genau der erwünschte Effekt. Allerdings ergeben Unterprogramme, die in der selben AC enthalten sind wie der entsprechende Typ Operationen und Methoden der Utility, d. h. der aus dem AC-Knoten, nicht dem Typ-Knoten, resultierenden Klasse. Daher haben auch sie zunächst keinen Zugriff auf die Interna des Typs. Um dies zu beheben, müssen alle Operationen, die Zugriff erlangen sollen, eine 'friend'-Abhängigkeit zu der den entsprechenden Typ abbildenden Klasse erhalten. Aus Gründen der Übersichtlichkeit sollten diese Abhängigkeiten nicht dargestellt werden. Unter Umständen könnte ein entsprechendes Verhalten im Stereotyp `<<ac>>` enthalten sein.

Die Modellierung von Subprogram-Knoten zeigt Listing 4-6.

```
MapSubprogram_NodeToUML
   in sub: Subprogram_Node;
   inout umlModel: Model;
   return oper: Operation := new Operation
   return meth: Method := new Method;
begin
   MapCommonAttributesOfNode(oper); --e.g. Node_Id
   MapCommonAttributesOfNode(meth); --e.g. Node_Id
   oper.setConcurrency(sequential);
   meth.setSpecification(oper);
   switch sub.getSubprogramClassification
      case Is_Constructor: oper.addConstraint(constructor);
      case Is_Destructor: oper.addConstraint(destructor);
      case Is_Modifier: oper.addConstraint(modifier);
      case Is_Accessor: oper.setIsQuery(true);
   end switch;
   foreach Edge e in sub.getOutgoingEdges do
      MapEdgesForSubprogramToUML(e, oper, meth, umlModel);
   end foreach;
end;
```

Listing 4-6: Transformation eines Subprogram-Knotens nach UML

Parameter_Of_Type- und **Return_Type-Kanten** werden als Parameter der Operationen bzw. als deren Rückgabetyp modelliert. Für jeden Parameter wird ein im Rahmen dieser Operation eindeutiger Parametername erzeugt. Die UML würde es erlauben bei Parameter_Of_Type-Kanten zwischen

- schreibgeschützten Eingabeparametern (`in`),
- Eingabeparametern auf die ein schreibender Zugriff erlaubt ist (`inout`) sowie
- Ausgabeparametern (`out`) zu unterscheiden.

(vgl. [uml1.3], S. 2-43) Im RFG liegen dazu aber noch keine Informationen vor, weshalb immer `inout` gewählt wird. Return_Type-Kanten erhalten die Art `return`.

Call-Kanten werden als Usage-Abhängigkeit mit dem Stereotyp `<<call>>` modelliert. Dieser vordefinierte Stereotyp entspricht semantisch einer Call-Kante sehr gut. (vgl. [uml1.3], S. 2-46)

Actual_Parameter-Kanten bestehen zwischen einer globalen Variable und einem Unterprogramm. Sie können als Usage-Abhängigkeiten mit dem Stereotyp `<<parameter>>` modelliert werden. Analog hierzu können **Local_Var_Of_Type-Kanten** als Usage-Abhängigkeiten mit dem Stereotyp `<<local_variable>>` abgebildet werden. In beiden Fällen besteht zwar eine Verbindung zwischen den beteiligten Knoten, was als Assoziation dargestellt werden könnte, allerdings ist diese Verbindung nicht dauerhaft, sondern nur in einem gegebenen Kontext vorhanden. Beispielsweise besteht während der Ausführung eines Unterprogramms eine Verbindung zwischen diesem und einer über eine Actual_Parameter-Kante verbundenen Variable. Vor oder nach der Ausführung des Unterprogramms besteht diese Verbindung allerdings nicht. Solche Situationen werden in der UML mit sogenannten *transient links*, d. h. flüchtigen oder vergänglichen Verbindungen modelliert, die in Objektdiagrammen dargestellt werden. In Klassendiagrammen ergeben sie Usage-Abhängigkeiten. Die zuvor beschriebenen Modellierungen zeigt Listing 4-7.

```
MapEdgesForSubprogramToUML
  in edge: Edge;
  out oper : Operation;
  out meth : Method;
  inout umlModel: Model;
begin
  if typeof edge=Parameter_Of_Type_Edge or Return_Type_Edge then
    p := new Parameter;
    p.setName(Create_Unique_Name);
    p.setType(umlModel.getUMLfor(edge.getEnd));
    if typeof edge = Parameter_Of_Type_Edge then
      p.setKind(inout);
    else
      p.setKind(inout);
    end if;
    oper.addParameter(p);
    meth.addParameter(p);
  else
    dep := new Dependency;
    dep.setClient(oper);
    dep.setSupplier(umlModel.getUMLfor(edge.getEnd));
    switch typeof edge
      case Call_Edge: dep.setStereotype(call);
      case Actual_Parameter_Edge: dep.setStereotype(parameter);
      case Local_Var_Of_Type_Edge:
        dep.setStereotpye(local_Variable);
    end switch;
  end if;
end;
```

Listing 4-7: Transformation von Kanten zu bzw. von Unterprogrammen nach UML

Reference-Kanten

Reference-Kanten – wie z. B. Variable_Use-, Member_Set- oder Function_Address-Kanten – werden als Usage-Abhängigkeiten mit entsprechenden Stereotypen modelliert, wie dies Listing 4-8 zeigt. Set-Kanten erhalten den Stereotyp `<<set>>`, Address-

Kanten <<address>>. Eine Schwierigkeit stellen Use-Kanten dar. Der Stereotyp <<use>> wird innerhalb der UML zur Visualisierung von Usage-Abhängigkeiten eingesetzt. Daher wird hier – entsprechend des für Set-Kanten gewählten Stereotyps – der Stereotyp <<get>> verwendet, was zudem besser auf einen lesenden Zugriff hinweist als die Benennung im RFG.

```
MapReference_EdgeToUML
   in ref: Reference_Edge;
   return depend: Usage := new Usage;
begin
   depend.setClient(ref.begin);
   depend.setSupplier(ref.end);
   switch typeof ref
      case Variable_Use_Edge or Member_Use_Edge:
         depend.setStereotype(get);
      case Variable_Set_Edge or Member_Set_Edge:
         depend.setStereotype(set);
      case Variable_Address_Edge or Member_Address_Edge or
         Function_Address_Edge:
         depend.setStereotype(address);
   end switch;
end;
```

Listing 4-8: Transformation einer Reference-Kante nach UML

4.3.2 Rücktransformation in einen RFG

Nachdem die Transformation der im RFG enthaltenen Knoten und Kanten in UML relativ ausführlich behandelt wurde, wird die Besprechung der Rücktransformation kurz gehalten. Prinzipiell kann immer davon ausgegangen werden, daß UML-Sprachkonstrukte, die in der Form vorliegen, in der sie bei der Transformation erzeugt wurden, auf die dort besprochenen Knoten bzw. Kanten im RFG abgebildet werden. Beispielsweise wird eine Klasse mit dem Stereotyp <<ac>> einen Atomic_Component-Knoten ergeben. Eine Übersicht gibt Tabelle 4-6. Im folgenden soll nur noch auf einige Sonderfälle bzw. interessante Situationen bei der Rücktransformation eingegangen werden.

UML-Konstrukt	Resultierende Knoten und Kanten im RFG
Klasse mit enthaltenen Elementen	Atomic_Component- oder Type-Knoten mit Part_Of- bzw. Enclosing-Kanten zu Teilknoten
Attribut eines Typs	Variable-, Constant- oder Member-Knoten mit Variable_Of_Type- bzw. Member_Of_Type-Kanten
Operation mit Parametern und Rückgabetyp	Subprogram-Knoten mit Parameter_Of_Type- und Return_Type-Kanten
Abhängigkeit	strukturell ‘passende’ Kante

Tabelle 4-6: Rücktransformation von Klassen und ihrem Inhalt in einen RFG

Rücktransformation einer Klasse

Eine Klasse wird normalerweise in einen Atomic_Component- oder einen Type-Knoten transformiert. Ein Atomic_Component-Knoten wird erzeugt, wenn die Klasse

- einen passenden Stereotyp besitzt (<<ac>>, <<adt>>, <<ado>>,<<hc>>, <<rs>>)
- und nicht als Typ eines Attributs verwendet wird.

Ein Type-Knoten entsteht, wenn die Klasse

- den Stereotyp <<typenode>> besitzt und
- keine Operationen oder eingebetteten Klassen enthält.

In allen anderen Fällen kann nicht eindeutig zwischen AC und Typ unterschieden werden. Es wäre möglich, diesen Konflikt aufzulösen. Beispielsweise könnte bei einem zweiten Zuordnungsversuch ein fehlender oder falscher Stereotyp ignoriert werden. Ist auch dann noch keine Entscheidung möglich, wäre es denkbar, die Klasse in zwei Knoten abzubilden, einen Atomic_Component- sowie einen in der AC enthaltenen Type-Knoten und jedem Knoten die für ihn passenden Teile der Klasse zuzuweisen. Im Rahmen dieser Arbeit geschieht dies nicht. Stattdessen werden Klassen, bei denen keine eindeutige Zuordnung möglich ist, ignoriert. Die vorgeschlagene Rücktransformation zeigt Listing 4-9.

```
MapClassToRFG
  in c: Class;
  inout view: View;
  return node: Node;
begin
  if (umlModel.getNumberOfAttributesOfType(c) = 0) and
    (c.getStereotype = (ac or adt or ado or hc or rs))
  then
    node := new Atomic_Component_Node;
    node.setVisibility(unknown);
    node.setObject_Name(c.getName);
    node.setAttributes(c.getTaggedValues);
    switch c.getStereotype
      case adt: node.setComponentClassification(adt);
      case ado: node.setComponentClassification(ado);
      case hc: node.setComponentClassification(hc);
      case rs: node.setComponentClassification(rs);
      default: node.setComponentClassification(ac);
    end switch;
  elseif (c.getNumberOfOperations = 0) and
        (c.getNumberOfEmbeddedClasses = 0) and
        (c.getStereotype = typenode)
  then
    node := new Type_Node;
    node.setVisibility(unknown);
    node.setObject_Name(c.getName);
    node.setAttributes(c.getTaggedValues);
  end if;
end;
```

Listing 4-9: Rücktransformation einer Klasse in den RFG

Rücktransformation einer Operation

Eine Operation ergibt einen Subprogram-Knoten. Zugehörige Methoden ergeben bisher keine zusätzliche Information und werden ignoriert. Der Rückgabetyp einer Operation wird als Return_Type-Kante abgebildet. Parameter ergeben Parameter_Of_Type-Kanten. Falls der Benutzer eine Parameterbezeichnung abgeändert hat, geht diese Information derzeit verloren. Es wird vorgeschlagen, ein neues Attribut Parameter_Name in die Parameter_Of_Type-Kante einzufügen, die dies verhindert. Auch die Angabe, wie sich eine Operation bei nebenläufigen Ausführungen verhält, geht derzeit verloren, was mit einem neuen Attribut concurrency im Subprogram-Knoten behoben werden könnte.

Rücktransformation eines Attributs

Attribute können je nach Kontext und enthaltenen Werten auf drei verschiedene Knoten abgebildet werden:

- einen Variable-Knoten, falls das Attribut in einer Klasse enthalten war, die als AC abgebildet wurde und das Attribut nicht als {frozen} definiert war,
- einen Constant-Knoten, falls das Attribut in einer Klasse enthalten war, die als AC abgebildet wurde und das Attribut als {frozen} definiert war und
- einen Member-Knoten, falls das Attribut in einer Klasse enthalten war, die als Typ abgebildet wurde.

Der Typ des Attributs wird als Variable_Of_Type- bzw. Member_Of_Type-Kante modelliert, soweit das Attribut nicht den speziellen Typ unknown hatte. In diesem Fall wird der Typ des Attributs verworfen.

Hat der Benutzer angegeben, welche Kapazität ein Attribut besitzt, wäre es sinnvoll, diese Information in den RFG zu übernehmen. Bisher existiert dazu aber noch keine Möglichkeit. Vorgeschlagen wird, die Of_Type-Kante um ein neues Attribut Multiplicity zu erweitern, das diese Information aufnehmen kann.

Rücktransformation einer Abhängigkeit

Abhängigkeiten wurden eingesetzt, um verschiedene Kanten zu modellieren. Bei der Rücktransformation wird ermittelt, um welche Kante es sich handelte. Dazu wird der Stereotyp der Abhängigkeit sowie ihr Start- und Endelement betrachtet und aus diesen Informationen auf eine passende Kante geschlossen. Beispielsweise wird eine Abhängigkeit mit dem Stereotyp <<call>>, die von einer Operation zu einer weiteren Operation verläuft, eine Call-Kante zwischen den aus den Operationen resultierenden Unterprogrammen ergeben. Kann kein passender Kantentyp gefunden werden, muß davon ausgegangen werden, daß der Benutzer eine neue Abhängigkeit eingefügt hat bzw. eine bestehende verschoben hat. Es wäre möglich, solche Abhängigkeiten zu ignorieren. Alternativ kann ein neuer Kantentyp im RFG definiert werden, der für die Abbildung solcher Abhängigkeiten verwendet wird. Dieser Weg wurde hier verfolgt. Zu diesem Zweck wurde die neue Kante Depends_On_Edge definiert, die von Non_Primitive_Edge erbt, wie dies Abbildung 4-15 zeigt. Sie beschreibt eine nicht näher definierte Abhängigkeit zwischen zwei beliebigen Knoten. Über ein spezielles

Attribut kann die Bezeichnung bzw. die Art der Abhängigkeit angegeben werden.

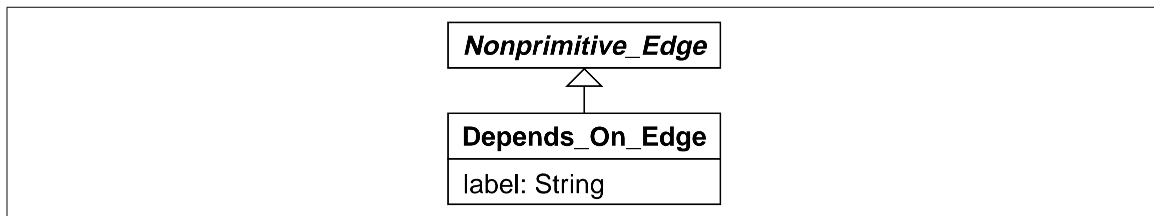


Abbildung 4-15: Klassenhierarchie der Kanten der AC-Erkennung

Kann eine Abhängigkeit im UML-Modell keiner anderen Kante zugeordnet werden, wird eine Depends_On-Kante mit der entsprechenden Bezeichnung, die sich aus dem Stereotyp der Abhängigkeit ergibt, eingefügt.

4.3.3 Erlaubte Änderungen und Integration mit dem RFG

Um eine einfache Integration zu ermöglichen, sind nur relativ wenige Benutzerinteraktionen zulässig:

- Der Benutzer darf eine AC bzw. eine Utility-Klasse umbenennen, soweit er sonst keine Änderungen an ihr vornimmt. Bei der Integration kann dies über die strukturelle Identität der unterschiedlich benannten ACs erkannt werden und die ursprüngliche AC umbenannt werden.
- Teilelemente einer Utility, d. h. Operationen, Attribute und aus Typen resultierende Klassen, dürfen in ein anderes Utility verschoben werden. Dies führt bei der Integration dazu, daß die alten Part_Of-Kanten gelöscht und durch solche zur neuen AC ersetzt werden.
- Es ist zulässig, Modellelemente, z. B. eine Operation oder eine Utility, zu löschen. Dadurch werden die diesem Element entsprechenden Knoten und Kanten zunächst aus der Architektur-View und bei der Integration mit dem ursprünglichen RFG aus der User-View entfernt. In der Base-View bleiben sie jedoch enthalten.
- Das Erstellen einer neuen Utility ist erlaubt, solange sie eine eindeutige Bezeichnung erhält und einen passenden Stereotyp trägt. Der resultierende Atomic_Component-Knoten wird der erkannten Architektur hinzugefügt.
- Um zusätzliche Zusammenhänge anzugeben, darf der Benutzer neue Abhängigkeiten mit beliebigem Stereotyp zwischen beliebigen Elementen einfügen. Bei der Integration werden allerdings nur neue Depends_On-Kanten übernommen, da alle anderen neuen Kanten in der Base-View abgelegt werden müßten, was nicht erlaubt ist.
- Die Klassifikation einer AC bzw. eines Unterprogramms darf geändert werden. Beispielsweise darf ein ADT zu einem ADO gemacht werden. Es ist aber unzulässig, eine unbekannte Klassifizierung festzulegen, z. B. eine AC mit einem Stereotyp `<<andere_ac>>`.
- Die Sichtbarkeit von Attributen einer aus einem Typ abgeleiteten Klasse darf geändert werden. Dabei ist zu beachten, daß eine globale Sichtbarkeit nur vorliegt, wenn das Attribut nicht die Einschränkung `{visibility_unkown}`

enthält. (siehe Tabelle 4-4, „Zuordnung von Knotensichtbarkeiten zu Sichtbarkeiten in der UML“, auf Seite 55) Nicht zulässig ist es zur Zeit, die Sichtbarkeit anderer Modellelemente zu ändern.

Alle weiteren denkbaren Manipulationen sind derzeit nicht zulässig. Im Laufe der Zeit sollten sie schrittweise freigegeben werden, sobald das Integrationswerkzeug mit ihnen umgehen kann.

Weitere Modellierungen

Kapitel 5

5.1 Modellierung von Subsystemen

Ausgehend von zuvor ermittelten ACs, werden bei der Subsystemerkennung ACs schrittweise zu größeren Teilen des Systems zusammengefaßt, den Subsystemen. Unter einem Subsystem wird in *Bauhaus* eine hierarchische Komponente verstanden, die aus miteinander in Beziehung stehenden ACs und/ oder anderen Subsystemen besteht. (vgl. [Kos00], S.27) Der Begriff der Komponente wird im Kapitel „Modellierung atomarer Komponenten“ auf Seite 41 erläutert.

Im Rahmen dieser Arbeit wird die Subsystemerkennung als abgeschlossen vorausgesetzt. Siehe [GirKos97] für ein Verfahren zur Subsystemerkennung.

5.1.1 Vorliegende Informationen im RFG

Ursprünglich lagen Subsysteme im RFG in Form sogenannter Cluster-Knoten vor. Dabei trat allerdings das Problem auf, daß ein bereits erkanntes Subsystem vom Benutzer durch Löschen einzelner Knoten strukturell zu einer AC gemacht werden kann, dies im RFG aber nicht reflektiert wird, d.h. die Komponente wird weiterhin als Subsystem behandelt. Um dieses Problem zu umgehen, werden Subsysteme derzeit über AC-Knoten modelliert. Jeder AC-Knoten, der weitere AC-Knoten enthält, wird als Subsystem angesehen. Wie Abbildung 5-1 zeigt, ermöglicht diese Definition

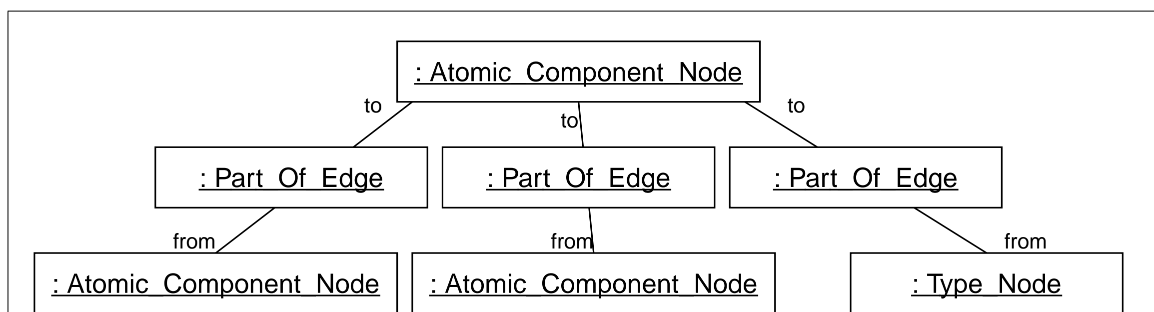


Abbildung 5-1: Beispiel für ein Subsystem im RFG

auch Subsysteme, in denen neben ACs Typen oder andere Elemente direkt enthalten sind.

5.1.2 Semantisch äquivalente UML-Sprachmittel

Charakteristisches Merkmal eines Subsystems in *Bauhaus* ist, daß es andere Elemente gruppiert, insbesondere wiederum Subsysteme oder ACs. Jedes Element kann in maximal einem Subsystem direkt enthalten sein. Da ACs auf spezielle Klassen abgebildet werden (siehe Kapitel „Modellierung atomarer Komponenten“ auf Seite 41), wird ein UML-Sprachmittel benötigt, das in der Lage ist, Klassen und weitere Exemplare von sich selbst zu enthalten.

Klassen

Klassen können neben Operationen und Attributen unter anderem auch andere Klassen enthalten. Sie stellen damit eine mögliche Modellierung für Subsysteme dar. Normalerweise werden Klassen in UML aber eingesetzt, um Systemteile auf einer relativ niedrigen Ebene zu modellieren. Für komplexe Hierarchien, wie sie Subsysteme im allgemeinen darstellen, werden sie nicht verwendet. Dazu sind in UML Pakete vorgesehen.

Pakete

Wie bereits berichtet, stellen Pakete in UML einen allgemeinen logischen Gruppierungsmechanismus dar, dessen Zweck nicht näher definiert ist. Insbesondere können Pakete Klassen und andere Pakete enthalten. Dies ist sogar der Normalfall. Sowohl von der Semantik als auch den Erwartungen der Benutzer her, entsprechen Pakete Subsystemen relativ gut, ohne daß Probleme bekannt wären.

Subsysteme in UML

Eine spezielle Form des Pakets, die im Metamodell als Kindklasse von `Package` enthalten ist, ist das Subsystem, hier zur Unterscheidung von Subsystemen in *Bauhaus* UML-Subsystem genannt. UML-Subsysteme stehen für abgeschlossene Systemteile. Sie sind daher konkretere Gruppierungen als Pakete, jedoch abstrakter als UML-Komponenten. Anders als Pakete sind UML-Subsysteme in der Lage, Schnittstellen zu anderen Systemteilen zu bieten und können Operationen enthalten, jedoch weder Attribute noch Methoden. Die enthaltenen Operationen müssen daher von im Subsystem enthaltenen anderen Elementen, z. B. Klassen realisiert werden. Die Elemente eines UML-Subsystems können optional in Spezifizierungs- und Realisierungselemente geteilt werden. (vgl. [uml1.3], S. 2-174 f.) Interessanterweise schlagen *Rumbaught* et al. in [Rum99], S.459 vor, ein System in eine Hierarchie von UML-Subsystemen aufzuteilen, die auf unterster Ebene Klassen enthalten. Dieses Vorgehen entspricht genau dem in *Bauhaus* gewählten.

UML-Subsysteme werden als Paket mit dem Stereotyp `<<subsystem>>` bzw. einem Gabelsymbol hinter ihrem Namen dargestellt. Falls gewünscht können sie in drei Abteilungen für Operationen, Spezifizierungs- und Realisierungselemente aufgeteilt werden. Abb 5-2 zeigt einige Darstellungsformen für UML-Subsysteme.

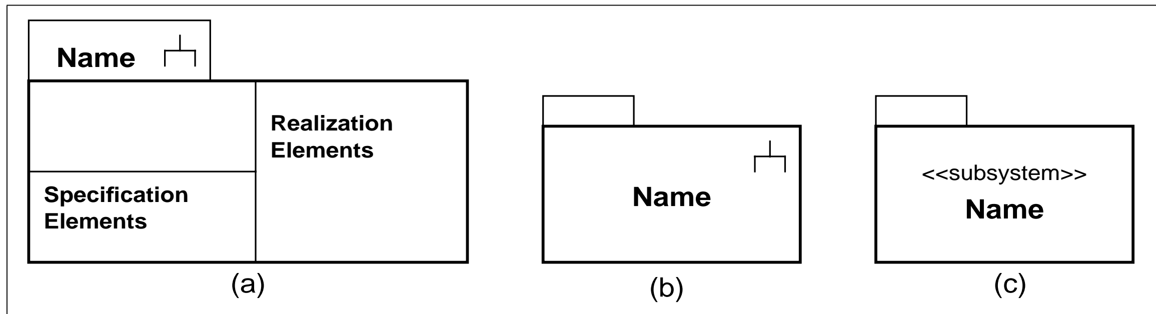


Abbildung 5-2: Verschiedene Darstellungen eines UML-Subsystems

Kollaborationen

Als weitere Möglichkeit für die Modellierung eines Subsystems könnte eine Kollaboration (`Collaboration`) angesehen werden. Diese definiert eine Beziehung zwischen Modellelementen. Sie stellt jedoch keine Gruppierung in dem Sinne dar, daß sie diese Elemente enthalten würde. Eher weist sie Modellelementen Rollen in einem gegebenen Kontext zu. Daher sind Kollaborationen für eine Abbildung von Subsystemen nicht geeignet.

5.1.3 Vorgeschlagene Modellierung

Transformation nach UML

Da sie semantisch passend sind und zu ähnlichen Zwecken eingesetzt werden, werden UML-Subsysteme für die Modellierung von Subsystemen im RFG verwendet. Pakete wären zwar ebenfalls eine sehr gute Möglichkeit, allerdings wäre es sinnvoll, diese mit einem passenden Stereotyp zu versehen, der dem Benutzer den Zusammenhang zum Subsystem verdeutlicht. Die Verwendung des naheliegenden Stereotyps ‘subsystem’ wäre nicht möglich, da das wiederum ein UML-Subsystem signalisieren würde. Es existiert kein Grund, in dieser Situation nicht direkt auf UML-Subsysteme zurückzugreifen.

Das Vorgehen entspricht dem bei der Modellierung von ACs gewählten weitgehend. Eine Abweichung entsteht bei der Modellierung der Klassifikationsinformation. Diese ist in allen Atomic_Component-Knoten enthalten, egal ob sie für ACs oder für Subsysteme stehen. Die Semantik dieser Klassifizierung bei einem Subsystem, beispielsweise die Information, daß es sich um ein ‘ADT’-Subsystem handelt, ist allerdings nicht definiert. Möglich wäre z. B. ein Subsystem, das einen ADT mit enthaltenen Teilkomponenten repräsentiert oder ein Subsystem, das nur ADTs enthält. Deshalb wird diese Information ignoriert. Sollte eine Festlegung dieser Semantik erfolgen, kann die Klassifizierungsinformation analog zum Vorgehen bei ACs mit Hilfe einer Hierarchie geeigneter Stereotypen modelliert werden. Die in Listing 4-1

auf Seite 53 angegebene Routine muß erweitert werden, wie dies Listing 5-1 zeigt. Neu hinzugekommene Anweisungen sind zur besseren Unterscheidung unterstrichen.

```
MapAtomic_Component_NodeToUML
  in anAc : Atomic_Component_Node;
  inout umlModel: Model;
  return c: Classifier;
begin
  if exists Part Of Edge part in anAc.getOutgoingEdges
    with part.end typeof Atomic Component Node
  then
    c := new Subsystem;
  else
    c := new Class;
    switch anAc.getComponent_Classification
      case ADT: c.setStereotype(adt);
      case ADO: c.setStereotype(ado);
      case HC: c.setStereotype(hc);
      case RS: c.setStereotype(rs);
      default: c.setStereotype(ac);
    end switch;
  end if;
  c.setName(anAc.getNode_Id.getObject_Name);
  c.addTaggedValue(anAc.getNode_Id.getFilename);
  c.addTaggedValue(anAc.getNode_Id.getProgram_Name);
  c.addTaggedValues(anAc.getOther_Attributes);
  c.setVisibility(public);
  c.setPersistence(transient);
  foreach Part_Of_Edge p in anAc.getIncomingEdges do
    c.add(MapNodeToUML(p.begin,umlModel));
  end foreach;
end;
```

Listing 5-1: Erweiterte Transformation eines Atomic_Component-Knotens nach UML

Die Modellierung eines einfachen Beispielsubsystems *Subsys* zeigt Abb 5-3. *Subsys* enthält zwei ACs, die HC *Eine_HC* und den ADT *Ein_ADT*.

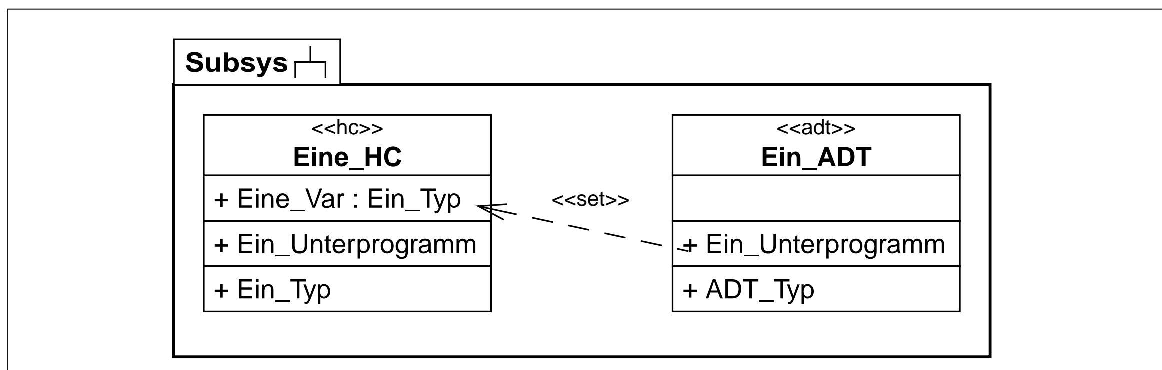


Abbildung 5-3: Beispiel einer Subsystemabbildung

Ein Sonderfall liegt vor, wenn ein Subsystem Knoten enthält, die nicht vom Typ `Atomic_Component_Node` sind, z. B. Subprogram-Knoten. Eine Modellierung solcher Subprogram-Knoten über Operationen des UML-Subsystems scheidet aus, da letztere nur virtuell existieren und von im Subsystem enthaltenen Elementen realisiert werden müssen. Analog zum Vorgehen bei Knoten, die in keiner AC enthalten sind (siehe Kapitel „Modellierung von Elementen, die keiner AC angehören" auf Seite 50), sind mehrere Reaktionen denkbar. Anders als dort, kann hier nicht argumentiert werden, die betreffenden Knoten seien noch nicht Teil der erkannten Architektur. Aus diesem Grund dürfen sie nicht ignoriert werden. Vielmehr wird virtuell ein neuer AC-Knoten in das Subsystem eingefügt, der die besagten Knoten enthält und eine Warnung gegeben. Um dem Benutzer zu verdeutlichen, daß der AC-Knoten eigentlich Teil des Subsystems ist, erhält er alle Attribute des Subsystems, z. B. auch dessen Name und Node_Id. Da der neue Knoten nicht wirklich im RFG liegt, sondern nur virtuell für die Modellierung in UML hinzugefügt wird, resultiert hieraus kein Problem mit der Eindeutigkeit der Node_Id.

Es entsteht ein Hierarchie von UML-Subsystemen. Diese kann in vielen miteinander verknüpften Diagrammen dargestellt werden, um große Systeme anzeigen zu können. Hierbei wird für jedes UML-Subsystem ein eigenes Diagramm erstellt. Dadurch kann der genaue Systemaufbau vom Überblick zu den Details verfeinert betrachtet werden.

Rücktransformation in einen RFG

Eine Rücktransformation ist sehr einfach möglich. Für jedes Subsystem wird ein Atomic_Component-Knoten erzeugt, wie dies Listing 5-2 zeigt. Zusätzlich hierzu wird

```
Map_Subsystem_To_RFG
  in subsys: Subsystem;
  inout view: View;
  return ac: Atomic_Component_Node := new Atomic_Component_Node;
begin
  ac.setNodeId(
    Object_Name => subsys.getName,
    Filename => subsys.getTaggedValue(Filename),
    Program_Name => subsys.getTaggedValue(Program_Name));
  ac.add_Attributes(subsys.getOtherTaggedValues);
  foreach ModelElement elem in subsys do
    part := new Part_Of_Edge;
    part.setBegin(Map_ModelElement_To_RFG(elem, view));
    part.setEnd(ac);
  end foreach;
end;
```

Listing 5-2: Rücktransformation eines Subsystems in UML in einen RFG

überprüft, ob in dem Subsystem ein Utility enthalten ist, dessen Name dem des Subsystems entspricht, bzw. das auf einen Knoten abgebildet werden würde, der die gleiche Node_Id wie das Subsystem hätte. In diesem Fall handelt es sich um einen bei der

Transformation virtuell in den RFG eingefügten Knoten, der nicht zurücktransformiert werden darf, sondern entfernt wird. Sein Inhalt wird dem Subsystem zugewiesen.

Erlaubte Änderungen und deren Integration mit dem ursprünglichen RFG

Es ist zulässig, ein Subsystem umzubenennen, wenn keine weiteren Änderungen an ihm erfolgen, außer, daß ein eventuell vorhandenes Utility gleichen Namens ebenfalls (gleich) umbenannt wird. Die Integration erfolgt analog zur für ACs beschriebenen. Ebenso wie bei ACs, darf der Benutzer ein Subsystem einfügen bzw. löschen.

Weitere erlaubte Benutzermanipulationen sind, ein Subsystem in ein anderes einzubetten, sowie ein eingebettetes Subsystem in ein anderes zu verschieben bzw. es aus dem Subsystem herauszuziehen. Dies führt zu einer Anpassung der entsprechenden Part_Of-Kanten.

In einem Subsystem enthaltene Utilities können aus diesem entfernt, bzw. in ein anderes verschoben werden. Sinnvoll wäre es, eine Möglichkeit vorzusehen, aus einer AC, d. h. einem Utility, ein Subsystem zu machen, und andersherum, wenn der Benutzer dies wünscht. Beispielsweise wäre es denkbar, daß der Benutzer bei der Darstellung der Architektur im UML-Werkzeug entdeckt, daß ein Utility irrtümlich einem Subsystem zugeordnet wurde und das Utility aus dem Subsystem entfernt. Enthielt das Subsystem nur diese Utility und z. B. eine aus einem virtuellen Knoten erzeugte, ist aus dem Subsystem dadurch eigentlich eine AC geworden, die als Utility abgebildet werden sollte. Hier existiert die Möglichkeit, das virtuelle Utility aus dem Subsystem herauszuziehen und das nun leere Subsystem zu löschen. Da das virtuelle Subsystem alle Attribute des Subsystems besaß, entsteht dadurch sofort wieder eine korrekte Modellierung. Schwieriger ist es, wenn das Subsystem leer ist, also kein geeignetes virtuelles Utility vorliegt. Dies wird erst nach einer Rücktransformation mit anschließender erneuter Transformation korrekt als Utility abgebildet. Soll aus einem Utility ein Subsystem gemacht werden, gibt es zwei Möglichkeiten. Zunächst kann der Benutzer manuell ein Subsystem erzeugen, das alle Eigenschaften des entsprechenden Utilities besitzt und das Utility mit diesem Subsystem ersetzen. Die zweite Möglichkeit besteht darin, in das Utility ein zweites Utility einzufügen. Dies ist bisher nicht zulässig. An dieser Stelle erscheint es aber sinnvoll, es zuzulassen. Bei der Rücktransformation wird dieses Utility zu einem normalen Atomic_Component-Knoten transformiert. Erst nach einer erneuten Transformation in die UML wird hieraus ein Subsystem erzeugt.

5.2 Modellierung von Konnektoren

Neben Komponenten sind bei der Erkennung einer Systemarchitektur Konnektoren wichtig. Sie repräsentieren Verbindungen zwischen Komponenten, über die diese miteinander kommunizieren, bzw. stellen Daten- und/ oder Kontrollflußkonstrukte mit einem bestimmten Protokoll dar, die den Zusammenhalt zwischen zwei oder mehr Komponenten herstellen. (vgl. [Eis00], S. 2) Konnektoren können durch einzelne Variablen oder Funktionen realisiert werden, aber auch durch eigenständige Komponenten, die beispielsweise als Filter oder Pipe fungieren. Es können verschiedene

Abstraktionsebenen unterschieden werden, Konnektoren auf hoher (z. B. Filter, Pipes), mittlerer (z. B. ADTs, ADOs) und niedriger Ebene (z. B. Zugriffe auf Variablen, Prozeduraufrufe).

In *Bauhaus* existieren seit einiger Zeit Bemühungen, Konnektoren zu erkennen. Dabei konzentriert man sich bisher auf Konnektoren auf mittlerer Abstraktionsebene. Konnektoren sind ADTs, ADOs oder andere ACs, die Listen, Stacks, Hashtabellen usw. realisieren. Neben der Konnektorkomponente selbst, werden Komponenten identifiziert, die über den Konnektor miteinander kommunizieren. Bisher werden nur einfache Konnektoren direkt erkannt, die genau zwei Komponenten verbinden. Komplexe Konnektoren werden u. U. als Menge einfacher Konnektoren interpretiert. (vgl. [Eis00], S.7)

Abb 5-4 zeigt ein Beispiel für einen einfachen Konnektor *Stack* und die über ihn kommunizierenden Komponenten *Producer* und *Consumer*. Die verwendete Notation enspricht der in [Eis00] verwendeten.

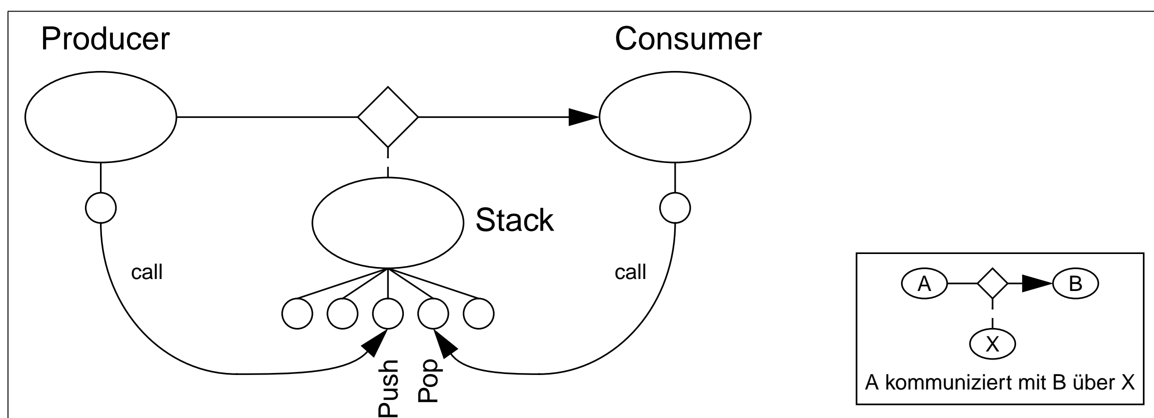


Abbildung 5-4: Beispiel für einen Konnektor

Die Konnektorenerkennung geht von einem RFG mit bereits erkannten ACs aus. Mit Hilfe eines allgemeinen Mustererkenners werden Komponenten ermittelt, die als Konnektor agieren, samt den beteiligten Datenquellen und -senken. Die Ergebnisse werden im RFG abgelegt. Weitere Informationen über Stand und Vorgehensweise der Konnektorenerkennung in *Bauhaus* können [Eis00] entnommen werden.

5.2.1 Vorliegende Informationen im RFG

Konnektoren werden im RFG je nach Art des Konnektors mit Hilfe der Knoten

- `ADO_Connector_Node` oder
- `ADT_Connector_Node`

und der Kanten

- `Connector_To_Edge`,
- `Connector_From_Edge` und
- `Connector_Using_Edge`

repräsentiert.

Die Klassenhierarchien dieser Knoten und Kanten zeigen Abbildung 5-5 und Abbildung 5-6.

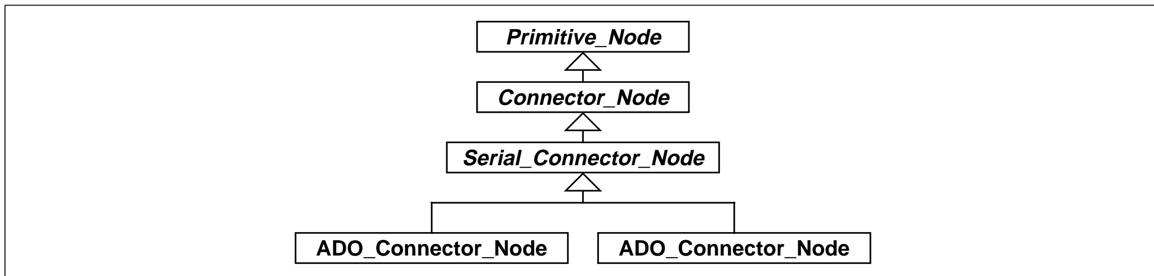


Abbildung 5-5: Klassenhierarchie der Knoten der Konnektorenerkennung

Jeder Konnektor wird über einen Knoten des entsprechenden Typs dargestellt. Welcher Knotentyp gewählt wird, ist von der Art der den Konnektor stellenden AC abhängig. So ergibt z. B. ein Konnektor, den ein ADO realisiert, einen ADO_Connector-Knoten. Die Kanten werden eingesetzt, um die beteiligten Akteure

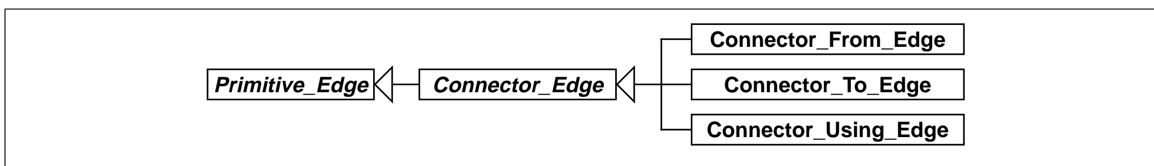


Abbildung 5-6: Klassenhierarchie der Kanten der Konnektorenerkennung

mit dem Konnektorknoten zu verbinden. Zwischen diesem und dem ihn repräsentierenden AC-Knoten wird eine Connector_Using-Kante eingefügt. Zur AC, die Daten in den Konnektor schreibt, führt eine Connector_From-Kante. Zur AC, die Daten entnimmt, führt eine Connector_To-Kante. Abb 5-7 zeigt, wie der aus Abb 5-4 bekannte ADO-Konnektor im RFG abgebildet ist.

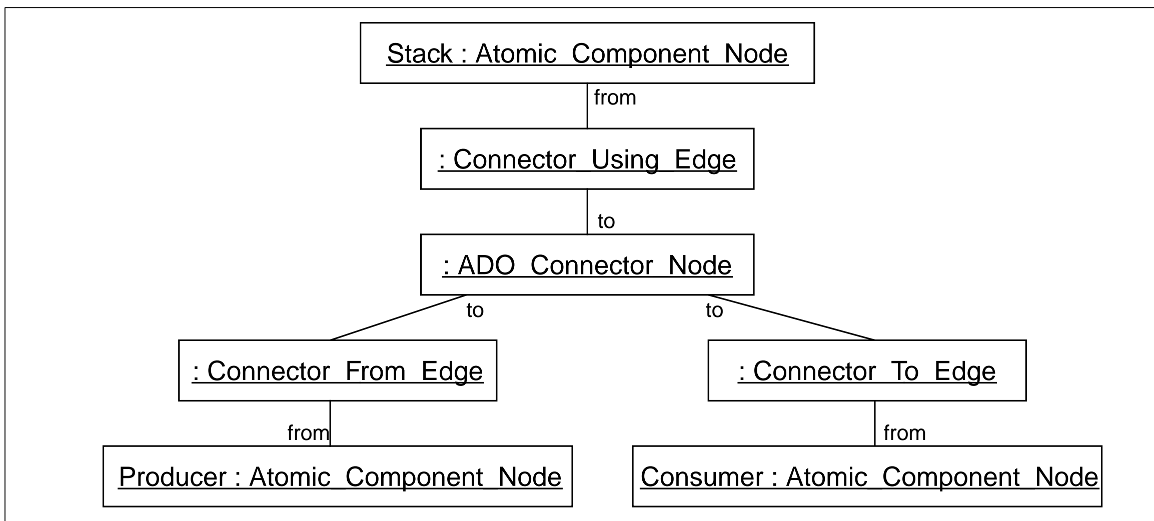


Abbildung 5-7: Beispiel für einen ADO-Konnektor im RFG

Derzeit sind Änderungen an dieser Modellierung angedacht. So soll es nur noch einen Knotentyp für alle Arten von Konnektoren geben. Ob dem Konnektor ein ADT oder ADO zugrundeliegt, kann vom über die Connector_Using-Kante mit dem Konnektor

verbundenen AC-Knoten direkt abgeleitet werden. Weitere Details liegen noch nicht vor. Bei der Modellierung von Konnektoren in UML sollte darauf geachtet werden, eine einfache Übertragung auf die neue Form im RFG zu ermöglichen.

Zum jetzigen Zeitpunkt ist es noch nicht möglich, die Richtung des Konnektors in jedem Fall festzulegen, d. h. zu erkennen, welche AC Daten einfügt, welche ausliest. Bessere Datenflußanalysen sollen hier Abhilfe schaffen. Im Rahmen dieser Arbeit wird davon ausgegangen, daß dieses Problem gelöst wurde. Ebenfalls derzeit nicht bekannt ist, in wie vielen verschiedenen Verbindungen die selbe Instanz eines Konnektors verwendet wird, ob es also z. B. ein oder drei Exemplare eines ADTs gibt, die Daten in einem gemeinsamen Stapel ablegen. (vgl. [Eis00], S.6)

5.2.2 Semantisch äquivalente UML-Sprachmittel

Aus den im RFG vorliegenden Informationen kann ermittelt werden, welche ACs mit Hilfe welcher Konnektoren miteinander kommunizieren und welche Rollen sie dabei einnehmen (Datenquelle oder -senke). In UML gibt es vor allem zwei Konstrukte, die für eine Abbildung dieser Informationen interessant sein könnten:

1. Assoziationsklassen und
2. (generische) Kollaborationen.

Assoziationsklassen

Eine Assoziationsklasse ist in UML definiert als eine Assoziation, die gleichzeitig selbst eine Klasse ist. Sie verbindet zwei oder mehr Klassen, hat eigene Attribute und Operationen und kann selbst wieder Assoziationen zu anderen Klassen haben. Dies erinnert an den hier verwendeten Konnektorbegriff. Allerdings darf es in der UML nicht vorkommen, daß zwei Objekte, die über eine bestimmte Instanz einer Assoziationsklasse verbunden sind, eine zweite Verbindung zueinander über eine weitere Instanz der gleichen Assoziationsklasse haben. Diese Bedingung ist für Konnektoren in *Bauhaus* zu restriktiv, da sie bisher nicht garantiert werden kann. (vgl. [Rum99], S. 157 ff.)

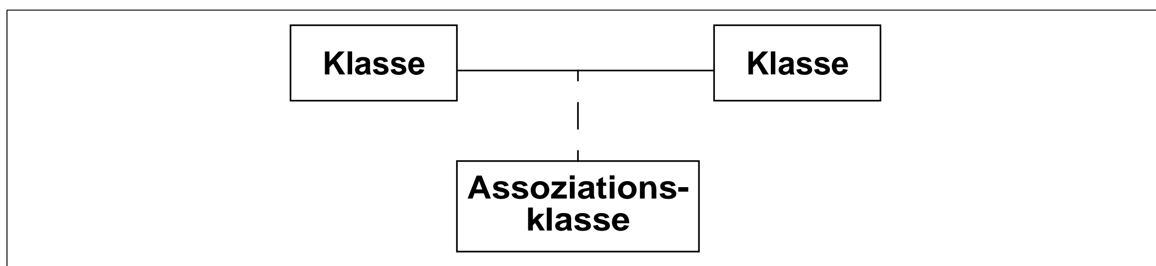


Abbildung 5-8: Darstellung einer Assoziationsklasse in UML

Kollaborationen (Collaboration)

Kollaborationen beschreiben, wie eine Operation, ein Anwendungsfall oder ein Aspekt eines Systems durch eine Menge von Modellelementen, z. B. Klassen und Assoziationen, die auf spezifische Weise verwendet werden, realisiert wird. Kollaborationen bestehen aus einem statischen und einem dynamischen Teil. Im statischen Teil weisen sie Modellelementen Rollen zu, die diese spielen, z. B. mittels Klassendiagrammen. Im dynamischen Teil legen sie die durchzuführenden Interaktionen fest, z.

B. in Sequenzdiagrammen. Von dieser Kollaborationsspezifikation können später Instanzen gebildet werden, die angeben, welche Instanzen der beschriebenen Modellelemente die Rollen tatsächlich spielen. (vgl. [uml1.3], S.2-106 ff.) Während ein Modell ein System als Ganzes beschreibt, zeigt eine Kollaboration demzufolge die statischen und dynamischen Aspekte eines bestimmten Modellausschnitts. Systemteile, die in einem bestimmten Kontext gemeinsam aktiv werden, werden gruppiert, wodurch es möglich wird, in diesem Zusammenhang unwichtige Elemente auszublenden.

Für die Modellierung von Konnektoren ist vor allem der statische Teil einer Kollaborationsspezifikation interessant. Der dynamische Teil könnte u. U. bei der Modellierung von Protokollen wichtig werden, indem er deren Modellierung enthält und damit ein Protokoll einem Konnektor zuordnet. Abb 5-9 zeigt die graphische Notation des statischen Teils einer Kollaborationsspezifikation in UML

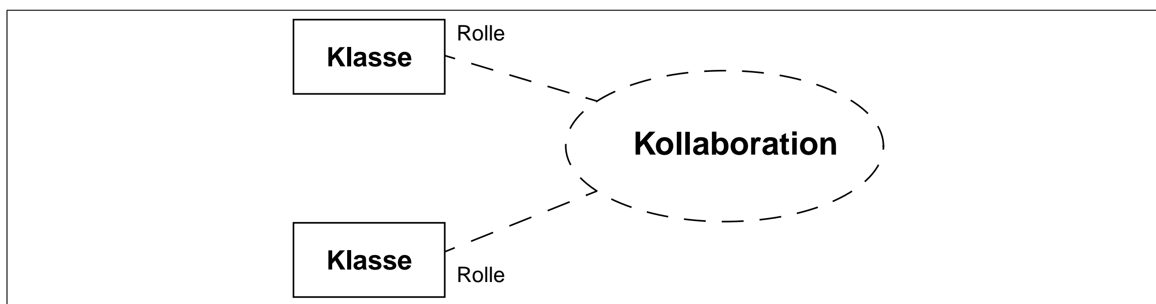


Abbildung 5-9: Darstellung einer Kollaboration in UML

Generische Kollaborationen

Um Kollaborationen, die prinzipiell gleichartig sind und sich nur in Details unterscheiden, nicht immer neu spezifizieren zu müssen, ist es in UML möglich, generische Kollaborationen anzulegen. Dies sind Vorlagen für Kollaborationen, die die beteiligten Modellelemente noch nicht festlegen, sondern nur mehr oder weniger exakte Bedingungen angeben, die Elemente erfüllen müssen, um eine bestimmte Rolle spielen zu können. Durch Angabe geeigneter Modellelemente kann aus einer generischen Kollaboration eine vollwertige Kollaboration erzeugt werden, von der wiederum Instanzen angelegt werden können. Dies zeigt Abbildung 5-10.

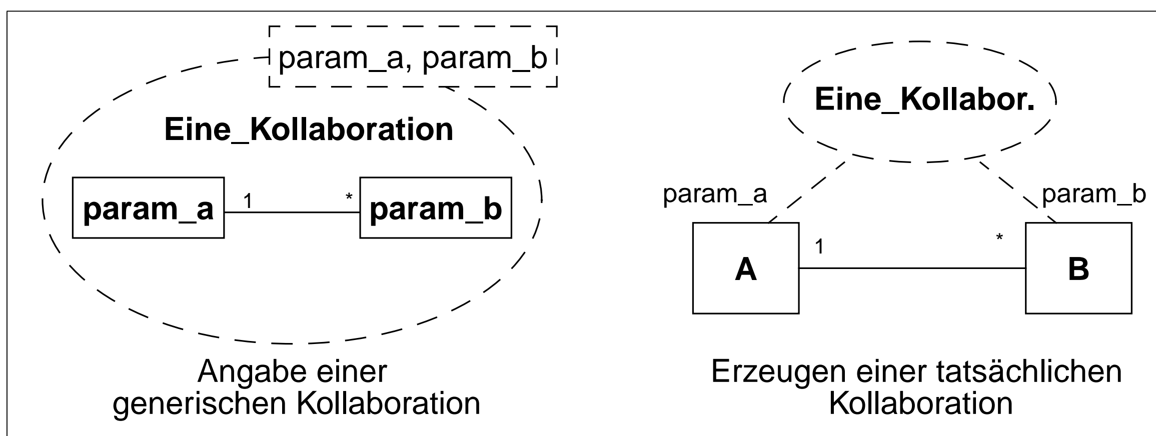


Abbildung 5-10: Darstellung einer generischen Kollaboration in UML

Mit generischen Kollaborationen werden in UML unter anderem Entwurfsmuster visualisiert. Damit bieten sie ein großes Potential für die Abbildung weiterer im RFG enthaltener Informationen, z. B. bestimmter typischer Muster in der Architektur. Insbesondere können sie zur Modellierung beliebig komplexer Konnektoren genutzt werden, so daß der Mechanismus mit der Konnektorerkennung wachsen kann.

Weitere Informationen über Kollaborationen sind unter anderem in [Rum99], S. 195 ff. zu finden.

5.2.3 Vorgeschlagene Modellierung

Transformation nach UML

Generische Kollaborationen sind sehr gut geeignet, um die Information, daß es sich bei bestimmten Komponenten im RFG um Konnektoren handelt und welche Komponenten über diese kommunizieren in UML abzubilden. Dabei wird der eigentliche Konnektor prinzipiell nicht anders als andere ACs abgebildet. In das UML-Modell wird pro Konnektorart eine eigene generische Kollaboration mit den noch offenen Rollen *From*, *To* und *Using* eingefügt. Wird ein entsprechender Konnektor gefunden, wird hieraus eine vollwertige Kollaboration abgeleitet, indem die beteiligten ACs der jeweils passenden Rolle zugewiesen werden. Dies ist problemlos möglich.

Abbildung 5-11 zeigt die für ADO-Konnektoren verwendete generische Kollaboration, die mit der für ADT-Konnektoren eingesetzten bis auf den Namen identisch ist. Falls später komplexere Konnektorarten hinzukommen, muß für diese jeweils eine eigene neue generische Kollaboration eingefügt werden.

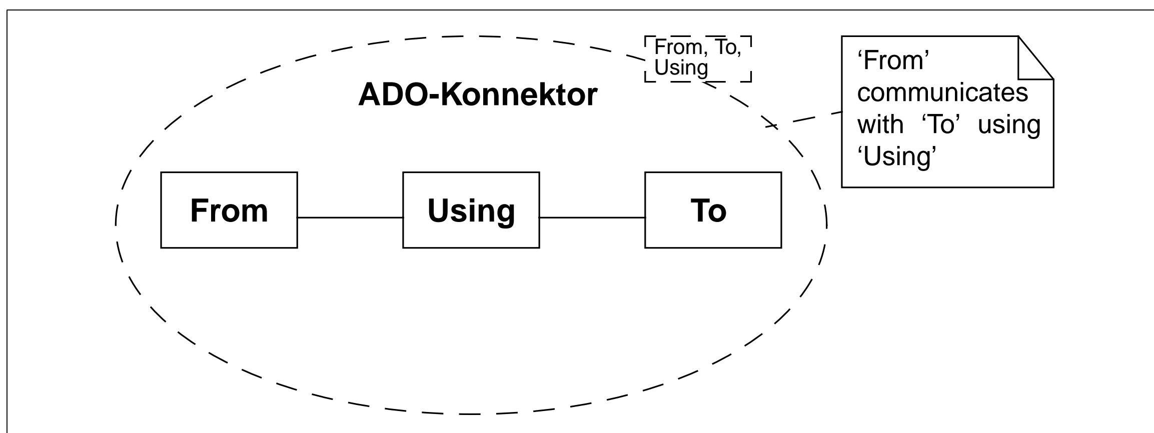


Abbildung 5-11: Generische Kollaboration für die Modellierung von ADO-Konnektoren

Das genaue Vorgehen bei der Modellierung von ADO-Konnektoren zeigt Listing 5-3. Auch hier gilt der entsprechende Vorgang analog für ADT-Konnektoren.

```
MapADO_ConnectorsToUML
  in rfg: RFG;
  inout umlModel: Model;
begin
  adoGenCollab := new GenericCollaborationForADO_Connector;
  umlModel.add(adoGenCollab);

  foreach ADO_Connector_Node connector in rfg do
    collab := new Collaboration(generic => adoGenCollab);
    collab.setName(adoCon.getNode_Id.getObject_Name);
    collab.setRole(from, connector.getIncomingEdgeOfType(
      Connector_From_Edge).getBegin;
    collab.setRole(to, connector.getIncomingEdgeOfType(
      Connector_To_Edge).getBegin;
    collab.setRole(using, connector.getIncomingEdgeOfType(
      Connector_Using_Edge).getBegin;
    umlModel.add(collab);
  end foreach;
end;
```

Listing 5-3: Transformation von ADO-Konnektoren nach UML

Rücktransformation in einen RFG

Auch das Zurückschreiben der Informationen in den RFG ist mit der gewählten Lösung problemlos möglich. Falls im UML-Modell eine Kollaboration gefunden wird, die eine passende generische Kollaboration realisiert, wird diese auf einen Konnektor abgebildet. Ob eine generische Kollaboration passend ist, kann über ihren Namen und ihre Struktur ermittelt werden. Kollaborationen, die keine solche generische Kollaboration realisieren, werden verworfen. Eine Erweiterung des RFGs um neue Knoten und Kanten würde es erlauben, auch diese zu übernehmen.

Das genaue Vorgehen zeigt Listing 5-4 stellvertretend für Kollaborationen, die ADO-Konnektoren ergeben.

```
MapCollaborationToRFG
  in umlModel: Model;
  inout rfg : RFG;
begin
  foreach Collaboration collab in umlModel do
    if exists (GenericCollaboration genColl in umlModel with
      (collab.realizes(genColl) and
      genColl.matches(GenericCollaborationForADO_Connector)))
    then
      connector := new ADO_Connector_Node;
      connector.setObject_Name(collab.Name);
      fromEdge := new Connector_From_Edge;
      fromEdge.setBegin(rfg.getMappingFor(collab.getFrom));
      fromEdge.setEnd(connector);
      toEdge := new Connector_To_Edge;
      toEdge.setBegin(rfg.getMappingFor(collab.To));
      toEdge.setEnd(connector);
      usingEdge := new Connector_Using_Edge;
      usingEdge.setBegin(rfg.getMappingFor(collab.Using));,
      usingEdge.setEnd(connector);
      rfg.add(connector, fromEdge, toEdge, usingEdge);
    end if;
  end foreach;
end;
```

Listing 5-4: Rücktransformation von Konnektor-Kollaborationen in einen RFG

Erlaubte Änderungen und deren Integration mit dem ursprünglichen RFG

Zulässige Änderungen sind

- die Umbenennung einer Kollaboration, soweit an dieser sonst keine Änderungen vorgenommen wurden,
- das Entfernen oder Hinzufügen einer Kollaboration und
- der Austausch eines an der Kollaboration beteiligten Utilities mit einem geeigneten anderen Utility.

Bei der Integration werden diese Änderungen in den ursprünglichen RFG einfach übernommen, was problemlos möglich ist.

5.3 Modellierung von Protokollen

Neben der statischen Struktur eines SW-Systems, sind für die Architekturerkennung dynamische Aspekte des Systems und seiner Elemente von Bedeutung. Diese zeigen, welche Interaktionen zwischen Elementen ablaufen und welchen Beschränkungen diese Interaktionen unterliegen, d. h. die tatsächliche und die zulässige Verwendung von Elementen, in *Bauhaus* insbesondere von ACs. Typischerweise wird es z. B. nötig sein, einen Stapel in geeigneter Weise zu initialisieren (`Init`), bevor Daten auf ihm

abgelegt werden können (`Push`). Daten werden nur dann vom Stapel genommen werden können (`Pop`), wenn dieser nicht leer ist. Teilweise könnte eine konkrete Realisierung andere Anforderungen stellen, immer aber ist es für eine korrekte Verwendung nötig, diese Anforderungen zu kennen.

Bei der Protokollerkennung wird der Ablauf der tatsächlichen Verwendungen von ACs ermittelt, z. B. welche Zugriffe auf Elemente einer AC innerhalb eines gegebenen Unterprogramms erfolgen. Wie die Erkennung dieser sogenannten *Traces* abläuft, wird in [Han00] erläutert. Aus *Traces* kann durch Zusammenfassen das Protokoll einer AC abgeleitet werden. Unter einem Protokoll wird hier die Spezifikation aller erlaubten Folgen von Aktionen auf ACs verstanden. Für detaillierte Betrachtungen über die hierfür nötigen Schritte wird auf [Hei00] verwiesen.

Abbildung 5-12 zeigt das bereits beschriebene Protokoll für einen Stapel, erweitert um eine Operation `Destroy`, die den Stapel freigibt. Die Darstellung erfolgt analog zur in [Hei00] gewählten Darstellung, die UML-Aktivitätsdiagrammen sehr nahe kommt.

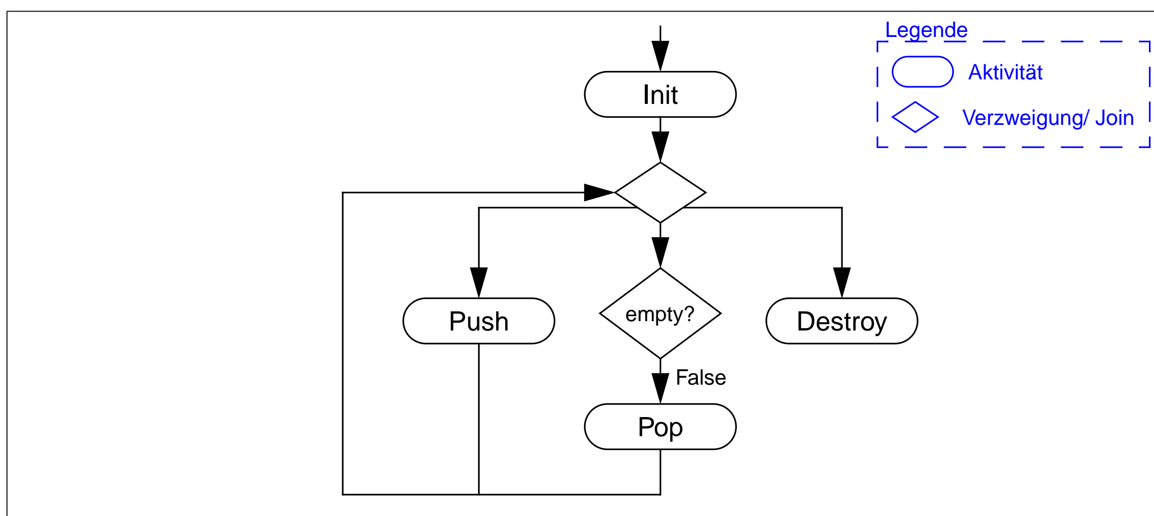


Abbildung 5-12: Beispielprotokoll

Neben der Information, welche Aktionsfolgen auf einer AC als Ganzes zulässig sind, werden – von den Unterprogrammen aus betrachtet – auch Bedingungen ermittelt, die *vor* (Preconditions) bzw. *nach* dem Aufruf des Unterprogramms erfüllt sein müssen (Postconditions). Allerdings werden nur solche Vor- und Nachbedingungen erkannt, die aus der Verwendung der AC abgeleitet werden können. Andere Bedingungen, beispielsweise solche, die die tatsächlichen Parameter eines Unterprogramms erfüllen müssen, sind nicht enthalten.

5.3.1 Derzeitige Form der vorliegenden Informationen im RFG

Zum Entstehungszeitpunkt dieser Arbeit war der Entwurf der im Rahmen der Protokollerkennung eingesetzten SW-Komponenten noch nicht abgeschlossen. Das gilt auch für die endgültige Form, in der die ermittelten Protokolle abgelegt werden. Fest steht derzeit, daß Protokolle im RFG abgelegt werden, und zu diesem Zweck neue Knoten- und Kantentypen eingeführt werden. Für diese existiert ein Entwurf, der allerdings weiteren Änderungen unterliegen wird. Die endgültige Namengebung der

Knoten und Kanten ist noch nicht bekannt. Jedes Protokoll wird vorerst in einem eigenen RFG abgelegt, ohne Verbindung zu seiner AC oder den die Aktivitäten ausführenden Unterprogrammen. Es ist somit zur Zeit nicht möglich, automatisch zu einer gegebenen AC dessen Protokoll zu ermitteln oder andersherum ein Protokoll einer AC zuzuordnen. Um dies zu ermöglichen, kann z. B. ein neuer Knoten `Protocol_Node` verwendet werden, der eine AC und die Startaktionen eines Protokolls über Kanten verbindet.

Genau betrachtet sind die Ergebnisse der Protokollerkennung derzeit noch *nicht Teil der erkannten Architektur* eines SW-Systems. Dennoch soll – soweit möglich – im folgenden untersucht werden, wie Protokolle in der UML modelliert werden können, sobald die genannten Punkte geklärt sind. Dazu wird den weiteren Betrachtungen der derzeit gültige Entwurf zugrundegelegt. Auf eine detaillierte Beschreibung der Transformationen muß jedoch verzichtet werden, da die vorliegenden Informationen hierfür noch nicht ausreichen. Ebenso wird auf die Diskussion der erlaubten Benutzermanipulationen sowie der Integration mit dem ursprünglichen RFG verzichtet.

Der RFG wird um insgesamt vier neue Knotentypen erweitert. Sie werden in Tabelle 5-1 aufgelistet. Der wichtigste neue Knotentyp ist der Action-Knoten. Dieser

Bezeichnung	**Beschreibung**
`Action`	Ausführung einer Aktion auf einer AC
`Activity`	zusammengesetzte Aktion, Teilgraph
`Unlabeled`	Alternative ohne Bedingung
`Labeled`	Alternative mit Bedingung

Tabelle 5-1: Knoten der Protokollerkennung

steht für einen einzelnen Zugriff auf ein Element einer AC, z. B. den Aufruf eines Unterprogramms. Mehrere Aktionen können zu einem Activity-Knoten gruppiert werden. Activity-Knoten sind eine angedachte Erweiterung des Konzepts und zur Zeit noch nicht realisiert. Kann in einer bestimmten Situation unter mehreren Aktionen eine Alternative frei gewählt werden, führt dies zu einer Verzweigung in Form eines Unlabeled-Knotens. Entscheidet eine gegebene Bedingung über die jeweils zu treffende Alternative, wird ein Labeled-Knoten eingefügt. Beispielsweise kann in dem aus Abbildung 5-12 bekannten Protokoll nach dem Aufruf des Unterprogramms `Init` frei gewählt werden, ob `Destroy` oder die zweite Verzweigung gewählt wird, was als Unlabeled-Knoten modelliert wird. Bei der zweiten Verzweigung kann `Pop` nur dann aufgerufen werden, wenn der Stapel nicht leer ist. Dies ist ein Beispiel für einen Labeled-Knoten.

Die Verbindungen zwischen den Knoten stellen die in Tabelle 5-2 gezeigten Kanten dar.

Bezeichnung	Quelle s	Ziel d	Beschreibung
`Unconditional`	`Action` | `Unlabeled`	`Action` | `Unlabeled` | `Labeled`	s wird vor d ausgeführt
`True`	`Labeled`	`Action`	d wird ausgeführt, wenn die Bedingung von s erfüllt ist
`False`	`Labeled`	`Action`	d wird ausgeführt, wenn die Bedingung von s nicht erfüllt ist

Tabelle 5-2: Kanten der Protokollerkennung

Unconditional-Kanten erzeugen Sequenzen von Aktionen, inklusive Verzweigungen. Eine True-Kante gibt an, welche Aktion ausgeführt wird, wenn die Bedingung eines Labeled-Knotens erfüllt ist, False-Kanten führen zu Aktionen, die ausgeführt werden, wenn dies nicht der Fall ist. Eine Modellierung des in Abbildung 5-12 gezeigten Protokolls mit Hilfe dieser Knoten und Kanten zeigt Abbildung 5-13.

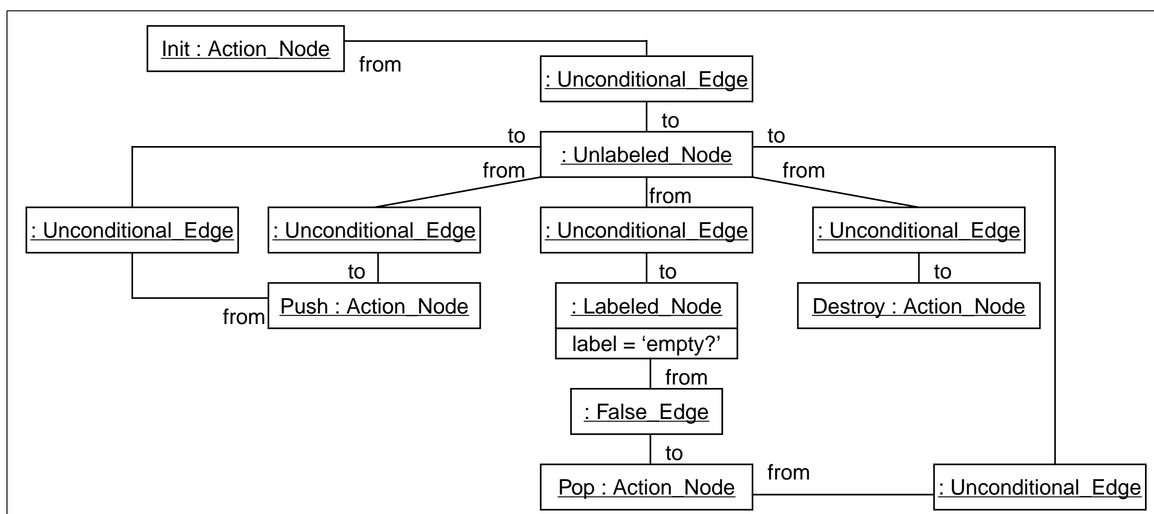


Abbildung 5-13: Beispiel für ein Protokoll im RFG

5.3.2 Semantisch äquivalente UML-Sprachmittel

Obwohl *Timo Heiber* in seiner Diplomarbeit bereits verschiedene Darstellungsformen diskutiert (siehe [Hei00], S. 9 ff.) und letztlich UML-Aktivitätsdiagramme präferiert, werden im folgenden neben dieser Möglichkeit kurz weitere UML-Sprachmittel auf ihre Verwendbarkeit hin untersucht.

Aktivitätsdiagramme und Aktivitätsgraphen

Die Visualisierung mit Aktivitätsdiagrammen (siehe Kapitel „Aktivitätsdiagramme (Activity Diagrams)" auf Seite 20) wird bereits in [Hei00] vorgeschlagen. Aktivitätsdiagramme zeigen Aktivitätsgraphen. Ein Aktivitätsgraph ist ein spezieller Zustandsautomat, der z. B. einen Berechnungsprozeß mittels dessen wesentlicher Aktionen und des Kontroll- und Daten-, bzw. Objektflusses zwischen diesen definiert.

Aktivitätsgraphen bieten Kurzformen für die Modellierung von Prozessen, die allgemeine Zustandsautomaten in UML nicht kennen. Dies führt zu einer einfacheren und leichter verständlichen Darstellung. (vgl. [uml1.3], S. 2-161) Für diese Betrachtung besonders interessant ist, daß Aktionen unter anderem Zugriffe auf Objekte und Operationsaufrufe sein können. Dies wird in der UML verwendet, um die Implementierung einer Operation anzugeben. Daher sind Aktivitätsgraphen eine gute Möglichkeit, Protokolle, aber auch Traces zu modellieren. Aktivitätsdiagramme und -graphen sind ein relativ neuer Teil der UML, weshalb noch mit Änderungen gerechnet werden muß. Schon beim Übergang von Version 1.1 zu 1.3 der UML wurden Anpassungen vorgenommen. Im Rahmen der laufenden Arbeiten zu UML 2.0 werden weitere Änderungen diskutiert. Es ist jedoch nicht damit zu rechnen, daß spätere Versionen die Verwendbarkeit von Aktivitätsgraphen für die Modellierung von Protokollen entscheidend beeinträchtigen werden.

Zustandsautomaten

Da Aktivitätsgraphen lediglich spezielle Zustandsautomaten darstellen, würden sich auch Zustandsautomaten prinzipiell für eine Modellierung anbieten. Allerdings tritt hier das Problem auf, daß im Rahmen der Protokollerkennung der tatsächliche Zustand einer AC nicht ermittelt wird, sondern nur, welche Aktionsfolgen an einem bestimmten Punkt im Lebenszyklus der AC zulässig sind. Dies würde zu einer in UML zwar zulässigen, aber unbefriedigenden Modellierung über anonyme Zustände führen und insgesamt in einer schlechter verständlicheren Darstellung resultieren, ohne daß es möglich wäre, mehr Information zu modellieren. Lediglich bei der Rücktransformation in den RFG könnten Vorteile entstehen, wenn der Benutzer den Zustand einer AC in der UML-Ansicht manuell hinzugefügt hat. Derzeit kann diese Information aber noch nicht im RFG gespeichert werden. Wirklich sinnvoll erscheint eine Modellierung von Protokollen mit Zustandsautomaten erst, wenn im RFG auch die Zustände einer AC abgelegt sind.

Interaktionsdiagramme

Eine weitere Möglichkeit, in der UML das dynamische Verhalten von Systemteilen zu zeigen, sind Interaktionsdiagramme (siehe die Kapitel „Sequenzdiagramme (Sequence Diagrams)“ auf Seite 19 und „Kollaborationsdiagramme (Collaboration Diagrams)“ auf Seite 19). Sie visualisieren die zeitliche Abfolge mehrerer Kommunikationsnachrichten zwischen Modellelementen, z. B. Objekten und betonen damit, wer mit wem kommuniziert und in welcher Abfolge. Die Quelle eines Zugriffs ist im Rahmen von Protokollen allerdings nicht von Bedeutung. Lediglich die zulässige Abfolge soll verdeutlicht werden. Es wäre möglich, eine Art ‘Pseudoquelle’ einzuführen, allerdings erscheint das nicht sinnvoll. Daher sind Interaktionen für die Modellierung von Protokollen nicht sehr gut geeignet. Sehr gut lassen sie sich dagegen verwenden, um Traces abzubilden, falls dies gewünscht werden sollte.

Vor- und Nachbedingungen

Eine gänzlich andere Möglichkeit als die bisher diskutierten ist, die durch die Protokollerkennung ermittelten Informationen zu verwenden, um Vor- und Nachbedingungen für Operationen anzugeben, wie dies bereits angesprochen wurde. Vor- und Nachbedingungen werden in der UML als Einschränkungen mit entsprechenden Stereotypen modelliert, wie dies Abbildung 5-14 zeigt.

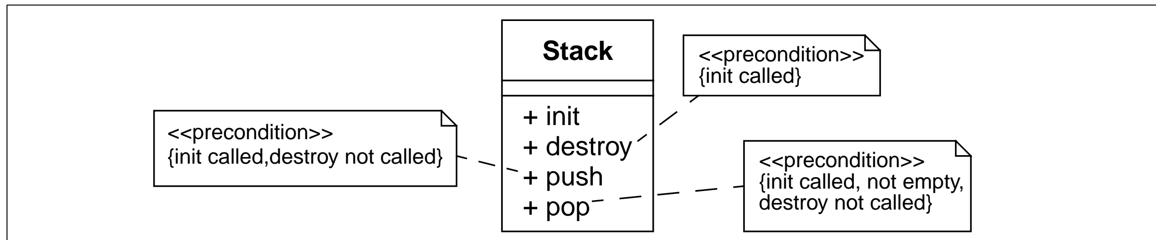


Abbildung 5-14: Vorbedingungen für Operationen in UML

Benötigt wird noch eine textuelle Darstellung der Protokollinformationen in einer geeigneten Form, die auch in der Lage ist, bei komplizierteren Protokollen verständliche Darstellungen zu erlauben. Die Darstellung wird recht schnell unübersichtlich werden, wenn alle Bedingungen präsentiert werden.

Besonders hier ist die Qualität der Ergebnisse der Protokollerkennung extrem wichtig. Sie muß eindeutig klären, daß bestimmte Aktionen notwendig sind und nicht nur zufällig im System immer in dieser Weise auftreten. Obwohl dies eine prinzipielle Anforderung an die Protokollerkennung ist, vermitteln besonders Vor- und Nachbedingungen, daß Aktionen wirklich notwendig sind, während andere Modellierungen, wie z. B. Aktivitätsdiagramme, auch als gemeinsame Muster interpretiert werden könnten.

5.3.3 Vorgeschlagene Modellierung

Von den oben diskutierten Möglichkeiten wird die Modellierung mit Aktivitätsdiagrammen vorgeschlagen, insbesondere, da die im RFG vorliegenden Informationen strukturell bereits auf eine solche Modellierung ausgerichtet sind, d. h. eine einfache Transformation möglich ist.

Um ein im RFG vorliegendes Protokoll mit UML zu modellieren, müssen kleine Anpassungen an der Struktur des Graphs vorgenommen werden. In UML existieren sogenannte *Pseudozustände* (`PseudoState`), die unter anderem bedingte Verzweigungen (`choice`) und Verzweigungen ohne Bedingung (`fork`) sowie Zusammenführungen (`join`) repräsentieren und streng genommen nicht vermischt werden dürfen. Daher müssen die im RFG vorliegenden Unlabeled- und Labeled-Knoten, die gleichzeitig als Verzweigungen und als Zusammenführungen dienen, gegebenenfalls in mehrere Pseudozustände aufgeteilt werden. Bedingungen werden nicht den Verzweigungen zugewiesen, sondern den ausgehenden Kanten und dort auch visualisiert. Außerdem muß sichergestellt sein, daß bei jeder bedingten Verzweigung eine Alternative gewählt werden kann, was unter Umständen einen `else`-Zweig nötig macht. Um den Aktivitätsgraphen in eine UML-konformere Form zu bringen, werden explizite Start-

und Endzustände hinzugefügt, auch wenn das nicht unbedingt nötig ist. Abbildung 5-15 zeigt die vorgeschlagene Modellierung des bereits mehrfach verwendeten Beispielprotokolls eines Stapels als Aktivitätsdiagramm.

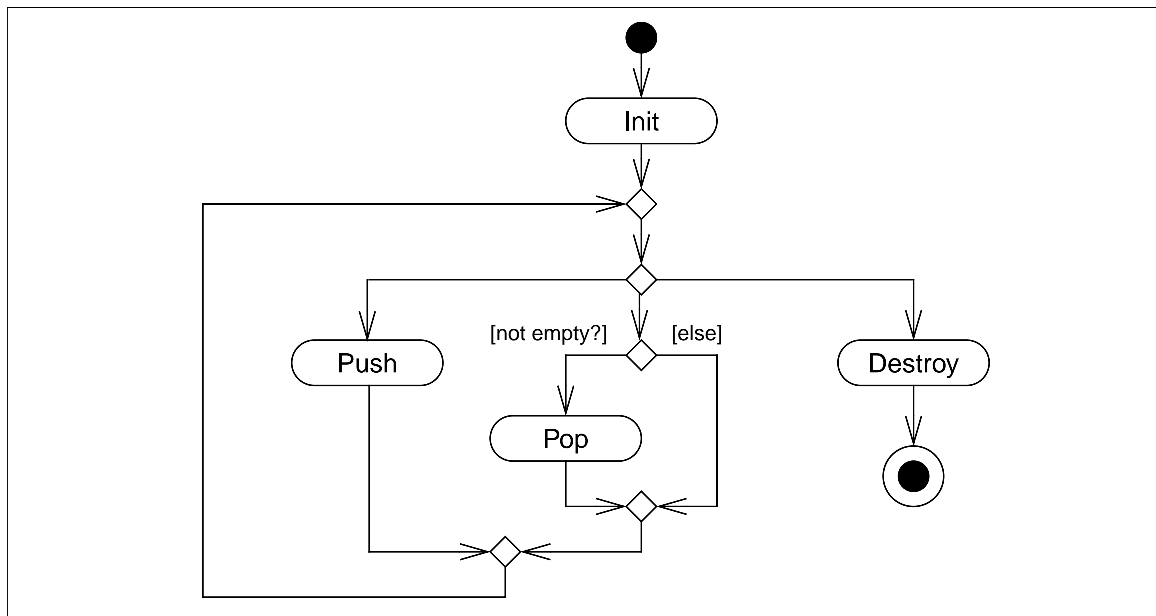


Abbildung 5-15: Modellierung eines Beispielprotokolls in UML

Es folgt eine kurze Aufzählung der jeweiligen Abbildung von Knoten und Kanten:

- **Action-Knoten** werden auf gleichnamige Aktionszustände (`ActionState`) abgebildet.
- **Activity-Knoten** ergeben SubactivityStates. (vgl. [uml1.3], S. 2-165)
- **Labeled-** und **Unlabeled-Knoten** ergeben Pseudozustände der jeweiligen Art (`choice` bzw. `fork`). Sind sie Ziel mehrerer Kanten, wird ein neuer Pseudozustand der Art `join` erzeugt und alle eingehenden Kanten auf diesen verbogen. Einziger Folgezustand dieses neuen Pseudozustands wird der ursprüngliche Pseudozustand.
- Für jede **Unconditional-Kante** wird eine Transition erzeugt, die die aus den Start- und Zielknoten der Kante abgeleiteten Zustände verbindet.
- **True-** und **False-Kanten** ergeben ebenfalls Transitionen. Zusätzlich zum Vorgehen bei Unconditional-Kanten wird bei ihnen die Bedingung (`guard`) der Transition gesetzt. Diese ergibt sich aus dem Label des mit der Kante verbundenen Labeled-Knotens und des jeweiligen Kantentyps, wie Abbildung 5-15 dies zeigt. Falls nötig, wird ein zusätzlicher `else`-Zweig eingefügt, um ein gültiges UML-Modell zu garantieren.

Prototypische Realisierung

Kapitel 6

Im zweiten Teil der Arbeit wurden die bisher vorgeschlagenen Werkzeuge und Modellierungen prototypisch realisiert. Ziel dieses Kapitels ist es nicht, detailliert Entwurf und Implementierung der Werkzeuge zu dokumentieren, beispielsweise welche programmiersprachlichen Konstrukte (z. B. Klassen) hierbei eingesetzt wurden. Dies erscheint bei einer prototypischen Realisierung nicht sinnvoll. Stattdessen sollen die grundsätzlichen Entscheidungen dargestellt und begründet werden, die zu der erfolgten Implementierung führten. Für eine genauere Dokumentation der Werkzeuge sei auf deren Quelltext verwiesen.

Zu realisierende Werkzeuge

Aus dem Kapitel „Vorgeschlagene Integration" auf Seite 37 sind die im Rahmen dieser Arbeit primär zu realisierenden Werkzeuge bekannt:

- *rfg2uml* transformiert eine in einem RFG vorliegende Architektur-View in ein UML-Modell in Form einer XMI-Datei,
- *uml2rfg* transformiert ein in einer XMI-Datei vorliegendes UML-Modell in eine Architektur-View in einem neuen RFG.
- *mergeRFGs* integriert die in einem RFG vorliegende Architektur-View mit einem zweiten RFG und den in diesem vorliegenden Views. Insbesondere kann der zweite RFG der ursprüngliche RFG sein. Dadurch können Änderungen in den ursprünglichen RFG übertragen werden.

Abbildung 6-1 zeigt beispielhaft das Zusammenspiel dieser Werkzeuge.

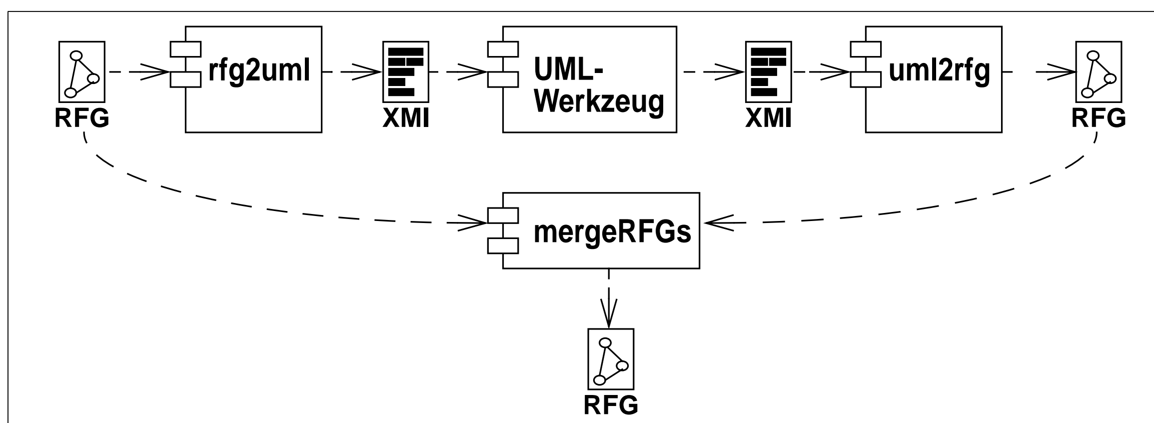


Abbildung 6-1: Zusammenspiel der beteiligten Werkzeuge

Realisierte Modellierungen

Nicht alle in dieser Arbeit vorgeschlagenen Modellierungen wurden realisiert, nur die in den Kapiteln „Modellierung atomarer Komponenten“ auf Seite 41 und „Modellierung von Subsystemen“ auf Seite 67 behandelten. Da die Speicherung von Protokollen (siehe Kapitel „Modellierung von Protokollen“ auf Seite 79) im RFG noch nicht vollständig implementiert ist, wurden Protokolle ausgelassen. Für Konnektoren wurde eine Modellierung mit Kollaborationen vorgeschlagen (siehe Kapitel „Modellierung von Konnektoren“ auf Seite 72). Leider war keines der vorliegenden UML-Werkzeuge in der Lage, Kollaborationen darzustellen, wodurch eine sinnvolle Realisierung nicht möglich war. Aus diesem Grund wurde auf die Realisierung der Modellierung von Konnektoren ebenfalls verzichtet. Auch die realisierten Modellierungen mußten teilweise angepaßt werden. Hierauf wird im Kapitel „Notwendige Anpassungen der Transformationen“ auf Seite 89 eingegangen.

Namengebung der Architektur-View

Um eine im Sinne dieser Arbeit zulässige Eingabe zu erhalten, wird ein RFG benötigt, der alle Knoten und Kanten der Architekturbeschreibung in einer gemeinsamen View enthält. Diese View wurde als Architektur-View bezeichnet. (siehe Kapitel „Anforderungen an den Resource Flow Graphen“ auf Seite 24) Den Werkzeugen muß mitgeteilt werden, welche View als Architektur-View angesehen werden soll. Bisher erfolgt dies der Einfachheit halber, indem der Name der Architektur-View statisch auf *Architecture_View* festgelegt wird. Die Festlegung auf diese Bezeichnung ist willkürlich. Alternativ könnten die Werkzeuge so erweitert werden, daß ihnen der Name der Architektur-View als Parameter übergeben werden kann.

Aus diesem Grund ist die im Kapitel „Erlaubte Änderungen und deren Integration mit dem ursprünglichen RFG“ auf Seite 40 angesprochene Änderung des Namens des UML-Modells unzulässig.

Weiteres Vorgehen

Im folgenden werden die zu realisierenden Transformationswerkzeuge *rfg2uml* und *uml2rfg* sowie das Integrationswerkzeug *mergeRFGs* besprochen. Neben diesen Werkzeugen wurden verschiedene Hilfswerkzeuge realisiert, so z. B. ein Werkzeug, das aus der Base- und der User-View eines RFGs die benötigte Architektur-View erstellt. Diese Werkzeuge werden im Anschluß an die primär zu erstellenden Werkzeuge im Kapitel „Realisierte Hilfswerkzeuge“ auf Seite 95 beschrieben.

6.1 Realisierung der Transformationswerkzeuge

Die Werkzeuge *rfg2uml* und *uml2rfg* werden zusammenfassend als Transformationswerkzeuge bezeichnet. Sie benötigen Schnittstellen

- zum verwendeten UML-Werkzeug, wofür XMI eingesetzt wird sowie
- zu den Bauhaus-Werkzeugen, bzw. dem RFG.

Eine geeignete Bibliothek für den lesenden und schreibenden Zugriff auf XMI-Dateien existiert derzeit noch nicht, so daß eigene und Lese- und Schreibroutinen entwickelt werden mußten. Da XMI auf XML basiert, sollte eine externe XML-Biblio-

thek verwendet werden, um den Realisierungsaufwand zu minimieren. Inzwischen liegt eine Vielzahl verschiedener Bibliotheken in verschiedenen Programmiersprachen vor, im XML-Umfeld auch XML-Parser genannt. Neben dem eigentlichen Parser enthalten diese Bibliotheken oftmals Routinen, die in der Lage sind, ein XML-Dokument auf syntaktische Korrektheit gegen die verwendete DTD zu prüfen (Validierung) und XML-Dateien in verschiedenen Zeichensatzkodierungen zu lesen und zu schreiben, neben ASCII z. B. in mehreren Versionen von Unicode. Solche XML-Bibliotheken sind relativ umfangreich und ihre Verwendung sehr zu empfehlen, soweit nicht auf die genannten Fähigkeiten verzichtet werden kann. Da *Rational Rose* XML-Dateien im Unicode-16 Format erzeugt, bei dem jedes Zeichen mit zwei Byte kodiert wird, war dies hier nicht der Fall.

Für den Zugriff auf einen RFG bietet sich die bereits angesprochene RFG-Bibliothek *GroupiusSE* an.

Es war nötig, die realisierten Transformationen teilweise abzuändern. Dies wird nachfolgend besprochen. Danach werden verschiedene Realisierungsmöglichkeiten diskutiert und die gewählte Variante ausgearbeitet. Dabei wird auch auf entstehende Probleme eingegangen.

6.1.1 Notwendige Anpassungen der Transformationen

An den realisierten Transformationen mußten einige Änderungen vorgenommen werden. Diese resultieren zum einen daraus, daß die meisten UML-Werkzeuge derzeit über XMI nur UML-Modelle austauschen können, die zur UML Version 1.1 konform sind, während die vorgestellten Transformationen auf UML 1.3 basierten. So heißt beispielsweise die im Kapitel „Modellierung von Typen und ihrer Teilkomponenten“ auf Seite 47 angesprochene Einschränkung `{xor}` in UML 1.1 `{or}`. Zum anderen mußten verschiedene Abweichungen der UML-Werkzeuge vom UML-Standard berücksichtigt werden.

- Subsysteme werden oftmals als Pakete mit dem Stereotyp `<<subsystem>>` modelliert. Dies ist nach dem UML-Standard zwar eine korrekte Darstellung, im UML-Modell sollte aber die eigens vorhandene Metaklasse `Subsystem` verwendet werden. Obwohl diese Modellierung daher dem UML-Standard nicht entspricht, wurde sie in dieser Arbeit verwendet um die betreffenden UML-Werkzeuge verwenden zu können.
- In der XMI-Ausgabe der betrachteten UML-Werkzeuge sind keine Methoden enthalten, auch wenn diese in der dem Werkzeug übergebenen XMI-Eingabe vorhanden waren. Daher wurden Subsystem-Knoten als Operationen ohne Methoden modelliert.
- Keines der betrachteten Werkzeuge ist aktuell in der Lage, Abhängigkeiten zwischen Attributen bzw. Operationen und anderen Modellelementen zu behandeln. Ein Beispiel für eine solche Abhängigkeit sind Call-Abhängigkeiten zwischen Operationen, die zur Modellierung von Call-Kanten verwendet werden. Solche Abhängigkeiten werden von den Werkzeugen ignoriert, wodurch ein großer Teil der modellierten Informationen verlorengeht. Um dies abzuschwächen, wurde ein Hilfswerkzeug realisiert, das aus Kanten im RFG, die als ignorierte Abhängigkeiten modelliert werden (z. B. Call-Kan-

ten), Depends_On-Kanten zwischen ACs ableitet. Siehe hierzu das Kapitel „Einfügen zusätzlicher Abhängigkeiten“ auf Seite 96. Depends_On-Kanten wurden im Kapitel „Rücktransformation einer Abhängigkeit“ auf Seite 64 eingeführt.

6.1.2 Realisierungsalternativen

Ziel der prototypischen Realisierung war es, mit möglichst geringem Aufwand die praktische Machbarkeit der vorgeschlagenen Modellierungen zu zeigen. Hierzu sollte soweit möglich auf existierende Programme bzw. Bibliotheken zurückgegriffen werden. Verschiedene Realisierungsalternativen wurden untersucht.

Realisierung in *Ada*

Die erste untersuchte Möglichkeit besteht darin, die Werkzeuge vollständig in *Ada* zu realisieren. Als Schnittstelle zum RFG wird die Bibliothek *GroupiusSE* eingesetzt. Für den lesenden und schreibenden Zugriff auf XML-Dateien findet eine XML-Bibliothek Verwendung. Abbildung 6-2 zeigt den resultierenden Aufbau der Werkzeuge.

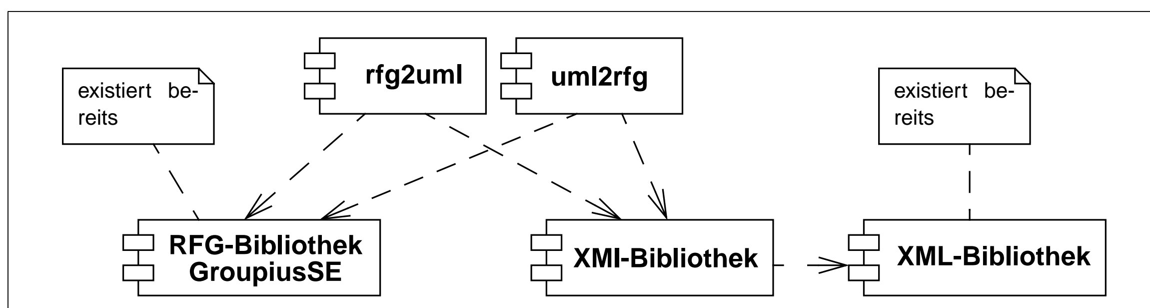


Abbildung 6-2: Aufbau der Transformationswerkzeuge in *Ada*

Benötigt wird hierfür eine geeignete XML-Bibliothek. Als Ausgangspunkt der Untersuchungen diente unter anderem die unter [xmltools] zu findende Aufzählung verschiedener XML-Werkzeuge und -Hilfsbibliotheken. Eine in *Ada* implementierte Bibliothek fand sich dabei nicht. Daher wurde die Untersuchung auf in *C++* implementierte XML-Bibliotheken ausgedehnt. Eine Integration zwischen *Ada* und *C++* ist relativ problemlos möglich. Verschiedene XML-Bibliotheken, z. B. *Apache Xerces for C++* ([xercesC++]) oder *libxml* (auch *gnome-xml* genannt, [libxml]), konnten trotz intensiver Bemühungen nicht erfolgreich übersetzt werden. Wahrscheinlich besteht eine Unverträglichkeit mit der verwendeten Compilerversion. Anders *expat* ([expat]), ein von *James Clark* geschriebener einfacher XML-Parser, der allerdings unter anderem keine Validierung bietet, so daß sein Einsatz keine große Erleichterung gewesen wäre. Derzeit wird im Rahmen eines Open Source Projekts *expat* 2.0 erstellt, das deutlich umfangreicher werden soll. Damit könnte *expat* mittelfristig für den Einsatz als XML-Bibliothek in *Bauhaus* geeignet sein. Zur Zeit ist allerdings keine der betrachteten Bibliotheken für eine Integration verwendbar.

Da keine geeignete XML-Bibliothek gefunden werden konnte, wurde diese Möglichkeit verworfen. Die Implementierung einer eigenen XML-Bibliothek stellt im Rahmen dieser Arbeit einen zu großen Aufwand dar.

Realisierung in *Ada* mit einer in *Java* implementierten XML-Bibliothek

Bei der Suche nach einer verwendbaren XML-Bibliothek zeigte sich, daß die reichhaltigste Auswahl an XML-Bibliotheken für *Java* besteht. Eine Kombination der in Ada geschriebenen Werkzeuge mit einer in *Java* implementierten XML-Bibliothek bietet sich daher an. Der Zugriff auf RFGs erfolgt bei dieser Realisierungsmöglichkeit weiterhin mit Hilfe der Bibliothek *GroupiusSE*. Aus diesem Grund ist es nötig, in *Ada* geschriebene Programmteile mit in *Java* realisierten zu integrieren. Hierfür wurden verschiedene Möglichkeiten evaluiert:

- Der *Ada*-Compiler *JGnat* ([jgnat]) ist in der Lage, aus *Ada*-Quelltexten Programmtext für die mit *Java* verwendete virtuelle Maschine zu erzeugen. Damit wäre es theoretisch möglich, *GroupiusSE* für diese Plattform zu übersetzen und eine Integration auf der Ebene der virtuellen Maschine zu erreichen. Leider scheiterte eine Übersetzung an Fehlern in *JGnat*.
- Die zweite Möglichkeit ist eine Integration über das sogenannte *Java Native Interface* (JNI). Dieses erlaubt es *Java*, auf in anderen Programmiersprachen realisierte Programme und Bibliotheken zuzugreifen. Im Standardumfang von *Java* sind entsprechende Werkzeuge für *C*-Programme enthalten. Der Aufwand für eine Integration mit *Ada* wäre allerdings sehr groß gewesen, da zunächst eine Anbindung zwischen *Ada* und *C* und daran anschließend zwischen *C* und *Java* benötigt worden wäre.
- Die für eine direkte Integration zwischen *Ada* und *Java* über JNI nötigen Komponenten sind in dem System *Ada to JNI* (AdaJNI) ([Fli00]) enthalten. AdaJNI verringert den für eine Integration nötigen Aufwand deutlich. Bisher arbeitet AdaJNI allerdings nur mit *Java* 1.1 zusammen, was zu Problemen mit aktuellen XML-Bibliotheken führte.

Keine der genannten Alternativen ist derzeit ohne erheblichen Aufwand realisierbar.

Realisierung mit XSLT

Eine gänzlich andere Realisierungsmöglichkeit bietet der Standard des W3C für die Transformation zwischen XML-Dokumenten: *Extensible Stylesheet Language Transformations* (XSLT) ([xslt]). XSLT liegt zur Zeit als Vorschlag vom 16. November 1999 vor. Von einem endgültigen Standard sind aber keine entscheidenden Abweichungen mehr zu erwarten. XSLT legt Syntax und Semantik einer XML-basierten Sprache fest, mit der aus einem XML-Dokument ein anderes XML-Dokument erstellt werden kann. Diese Fähigkeit wird insbesondere genutzt, um darstellbare XML-Dokumente zu erzeugen. Normale XML-Dokumente enthalten keine Informationen zu ihrer Darstellung. Mittels XSLT können sie in verschiedene XML-Dokumente mit enthaltenen Darstellungsinformationen transformiert werden, so daß aus einem Quelldokument verschiedene Darstellungen abgeleitet werden können.

Derzeit wird in einer Kooperation verschiedener Universitäten die *Graph Exchange Language* (GXL) ([gxl], [Hol00]) entwickelt. GXL ist ein standardisiertes Austauschformat für Graphen. Es soll z. B. eingesetzt werden, um die Ergebnisse verschiedener Reengineering-Werkzeuge auszutauschen. Auf dem Gebiet des Reengineering tätige Forschergruppen aus der ganzen Welt haben eine GXL-Unterstützung ihrer Werkzeuge angekündigt. Auch für *Bauhaus* soll eine GXL-Schnittstelle für den Austausch von RFGs entwickelt werden.

GXL basiert auf XML. Es ist damit möglich, eine GXL-Darstellung eines RFGs mit Hilfe von XSLT in eine XMI-Darstellung umzuwandeln, da beide auf XML basieren. Die hierzu nötigen Komponenten zeigt Abbildung 6-3.

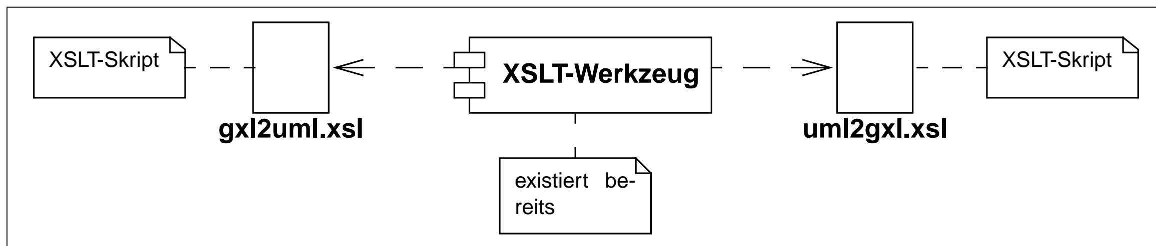


Abbildung 6-3: Aufbau der Transformationswerkzeuge mit XSLT

Die Erstellung der benötigten Skriptdateien ist sehr einfach möglich. Daher wurde diese Möglichkeit näher untersucht. Dabei zeigten sich große Performanceprobleme. Schon für RFGs mit einigen hundert Knoten ergaben sich Laufzeiten in der Größenordnung von Tagen. Dies ist wahrscheinlich vor allem darauf zurückzuführen, daß die GXL-Darstellung nicht hierarchisch ist, und die XSLT-Sprachmittel keine performante Transformation einer solchen Darstellung in eine hierarchische Darstellung wie XMI erlauben. XSLT ist vor allem für die Transformation von und in hierarchische Darstellungen ausgelegt. Da die genannten Laufzeiten inakzeptabel sind, wurde diese Möglichkeit nicht weiter verfolgt.

Realisierung in *Java*

Die letzte untersuchte Realisierungsalternative stellt die Realisierung in *Java* dar. Hierbei dient eine in *Java* implementierte XML-Bibliothek als Schnittstelle zu XMI. Als mögliche Bibliotheken wurden unter anderem untersucht: *Apache Xerces for Java* ([xercesJ]), *XP* von *James Clark* ([xp]), das auf *Xerces* basierende *XML4J* von *IBM* ([xml4j]) sowie *Project X* von *Sun* ([projectX]). Der Funktionsumfang dieser Bibliotheken ist recht ähnlich. Alle außer *XP* erlauben eine Validierung von Dokumenten und bieten verschiedene Zeichensatzkodierungen. Unterschiede gibt es in der Anzahl der unterstützten Kodierungen und den zur Verfügung stehenden Schnittstellen der Bibliotheken. Die wichtigsten Kodierungen und Schnittstellen bieten aber alle. Die größte Zukunftssicherheit bietet *Apache Xerces for Java*, da sich an dieser Entwicklung die verschiedensten Gruppierungen beteiligen, unter anderem auch die Unternehmen *IBM* und seit kurzem *Sun*, die den Quelltext der Bibliothek *Project X* einbringen und diese wohl nicht allein weiterführen werden. *XP* wird ebenfalls nicht mehr weiterentwickelt. Gewählt wurde daher die Bibliothek *Apache Xerces for Java*.

Obwohl es möglich wäre, eigene Ein-/ Ausgabe-Routinen für den Zugriff auf das native RFG-Format zu entwickeln, ist es einfacher, auf RFGs über die bereits erwähnte GXL-Schnittstelle zuzugreifen und für diese Ein-/ Ausgaberoutinen zu realisieren. Den resultierenden Aufbau der Transformationswerkzeuge zeigt Abbildung 6-4.

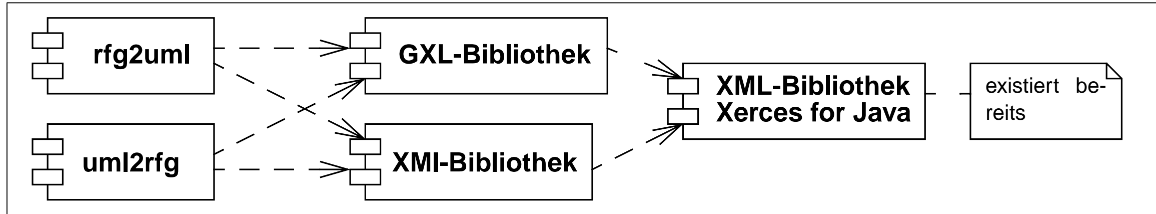


Abbildung 6-4: Aufbau der Transformationswerkzeuge in Java

6.1.3 Realisierung

Gewählt wurde die zuletzt besprochene Realisierung in Java, da diese Alternative mit dem geringsten Aufwand verwirklicht werden konnte und gleichzeitig in der Lage war, auch große RFGs in annehmbarer Zeit zu verarbeiten. Die benötigten Java-Klassen wurden in verschiedenen Paketen gruppiert, deren Abhängigkeiten Abbildung 6-5 zeigt. Im Paket *rfg* sind alle für eine Verarbeitung von RFGs benötigten Klassen enthalten. Das Paket *gxl-io* ermöglicht es, einen RFG aus einer GXL-Datei einzulesen, bzw. als GXL-Datei zu schreiben. Analog dazu enthält das Paket *uml* alle Klassen, die benötigt werden um UML-Modelle zu bearbeiten und *xmi-io* erlaubt es, aus einer XMI-Datei ein UML-Modell einzulesen, bzw. dieses in eine XMI-Datei zu schreiben. Die Transformationen sind im Paket *transformations* enthalten. Das Paket *tools* schließlich umfaßt die eigentlichen Werkzeuge.

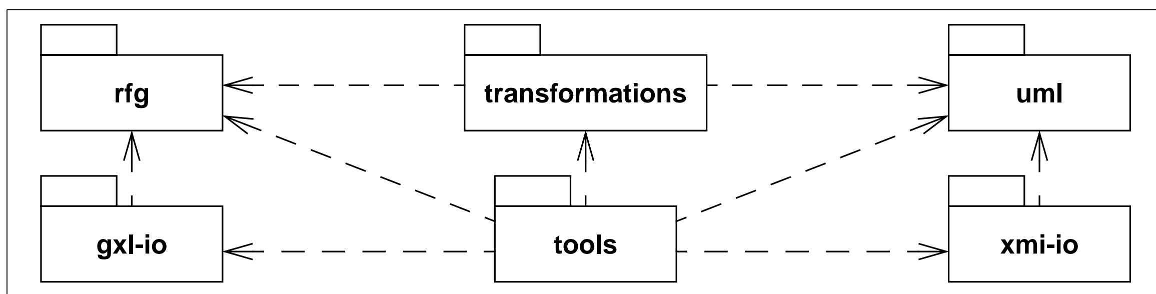


Abbildung 6-5: Zur Realisierung verwendete Java-Pakete

Da die Transformationswerkzeuge als Ein-, bzw. Ausgabe einen RFG im GXL-Format erhalten, bzw. ergeben, wurden sie umbenannt:

- ***gxl2uml*** ersetzt *rfg2uml*,
- ***uml2gxl*** ersetzt *uml2rfg*.

Das grobe Vorgehen des Werkzeugs *gxl2uml* zeigt Listing 6-1, *uml2gxl* arbeitet analog hierzu.

```
gxl2uml
  theRFG : RFG;
begin
  theRFG.loadAsGXL(gxlFileName);
  theUmlModel := theRFG.transformToUML;
  theUmlModel.saveAsXMI(xmiFileName);
end;
```

Listing 6-1: Ablauf des Werkzeugs *gxl2uml*

Schwierigkeiten bei der Implementierung der Werkzeuge entstanden durch die Größe mancher RFGs. Es lag unter anderem ein Basis-RFG mit etwa 15000 Knoten und mehr als 100000 Kanten vor. Dieser erreichte schon in der GXL-Darstellung eine Größe von etwa 15 Megabyte (MB). Die durch *gxl2uml* erzeugte XMI-Datei war mehr als 100 MB groß. Um XML-Dateien dieser Größenordnung verarbeiten zu können, mußten einige Optimierungen an der Implementierung vorgenommen werden. So erfolgte z. B. der Zugriff auf die XML-Bibliothek über die ereignisbasierte Schnittstelle *Simple API for XML* (SAX), anstatt mit Hilfe der komfortableren Schnittstelle über das *Document Object Model* (DOM), bei der automatisch ein einheitlicher Parsebaum erzeugt wird. Zudem wurden die Knoten des RFGs in einer Hashtabelle abgelegt um einen schnelleren Zugriff zu gewährleisten.

Werkzeuge für die GXL-Anbindung

Zusätzlich benötigt werden Werkzeuge, die einen RFG als GXL-Datei speichern, bzw. einen im GXL-Format vorliegenden RFG ins native RFG-Format überführen. Die Realisierung dieser Werkzeuge ist eigentlich Teil des Projekts zur Entwicklung einer GXL-Anbindung für Bauhaus. Da dieses Projekt allerdings noch nicht ausreichend fortgeschritten war, wurden eigene prototypische Werkzeuge erstellt, die auf bereits existierenden prototypischen Ein-/ Ausgaberoutinen für GXL basieren:

- ***rfg2gxl*** konvertiert die Architektur-View eines RFGs in einen RFG im GXL-Format,
- ***gxl2rfg*** konvertiert eine in Form einer GXL-Datei vorliegende View eines RFGs in einen RFG im nativen RFG-Format.

Diese Werkzeuge wurden in *Ada* mit Hilfe der Bibliothek *GroupiusSE* erstellt.

6.2 Realisierung des Integrationswerkzeugs

Den zweiten Teil der prototypischen Realisierung stellt das Integrationswerkzeug *mergeRFGs* dar. Nachdem ein vom Benutzer manipuliertes UML-Modell in einen RFG transformiert wurde, integriert *mergeRFGs* die vorgenommenen Änderungen in den ursprünglich abgebildeten RFG. Dies ist eine sehr komplexe Tätigkeit. Für viele Situationen ist es sehr schwierig, Manipulationen des Benutzers den richtigen Knoten im RFG zuzuweisen. Schon z. B. das Umbenennen eines Subprogram-Knotens ist schwer bzw. teilweise gar nicht erkennbar, wenn der Knoten gleichzeitig in eine neue AC verschoben wurde. Zudem verdeutlicht dieses Beispiel ein weiteres Problem. Sub-

program-Knoten sind Teil der Base-View. Das Umbenennen eines in der Base-View vorliegenden Knotens ist bisher nicht erlaubt. Es ist nicht klar, wie das Integrationswerkzeug auf eine solche Situation reagieren soll, insbesondere, wenn es den ursprünglichen Knoten nicht eindeutig identifizieren kann. Die allgemeine Lösung der entstehenden Schwierigkeiten geht über den Rahmen dieser Arbeit weit hinaus. Ein entsprechendes Werkzeug wird in *Bauhaus* unter dem Namen *rfgdiff* derzeit erstellt. Es ist allerdings bisher noch nicht einsetzbar.

Unter der Annahme, daß der Benutzer nur solche Manipulationen vorgenommen hat, die in dieser Arbeit jeweils als zulässig definiert wurden, reicht ein relativ einfaches Werkzeug zur Integration der Änderungen in den ursprünglichen RFG aus. Dieses Werkzeug tauscht lediglich die User-View des ursprünglichen RFGs komplett gegen die des neu erzeugten RFGs aus. Bis *rfgdiff* hinreichend weiterentwickelt wurde, wird dieses Werkzeug verwendet.

Die Realisierung ist denkbar einfach. Sie erfolgt in *Ada* mit Hilfe der Bibliothek *GroupiusSE* und den in *Bauhaus* im Paket `Acd.References` vorliegenden Routinen zum Speichern und Laden einer View in Form einer sogenannten *Reference-Datei*. Die benötigten Schritte zeigt Listing 6-2.

```
mergeRFGs
  Changed_RFG : RFG := Load_RFG(Changed_RFG_Name);
  Org_RFG : RFG := Load_RFG(Original_RFG_Name);
begin
  Export_User_View(Changed_RFG, Ref_File_Name);
  Delete_User_View(Org_RFG);
  Import_User_View(Org_RFG, Ref_File_Name);
  Save_RFG(Org_RFG);
end;
```

Listing 6-2: Ablauf des Werkzeugs *mergeRFGs*

Der vom Transformationswerkzeug *uml2rfg* – bzw. genauer gesagt von der Verkettung der Werkzeuge *uml2gxl* und *gxl2rfg* – erzeugte RFG enthält zunächst ausschließlich eine Architektur-View, die die Architektur des SW-Systems enthält (siehe Kapitel „Realisierung der Transformationswerkzeuge“ auf Seite 88). Um diese Informationen in den ursprünglichen RFG zu übernehmen, muß die Architektur-View zunächst in User- und Base-View aufgeteilt werden, um einen RFG zu erhalten, der mit dem beschriebenen Integrationswerkzeug verarbeitet werden kann. Dazu dient das Hilfswerkzeug *splitArchitectureView*, das zusammen mit weiteren realisierten Hilfswerkzeugen im nächsten Kapitel behandelt wird.

6.3 Realisierte Hilfswerkzeuge

Neben den bisher beschriebenen Werkzeugen wurden verschiedene Hilfswerkzeuge realisiert.

6.3.1 Erstellen der Architektur-View

Um mit den erstellten Werkzeugen arbeiten zu können, wird ein Werkzeug benötigt, das einem bestehenden RFG eine Architektur-View hinzufügt. Als Ausgangsbasis für die Architektur-View dienen in dieser Arbeit die User- und die Base-View. Daher wurde ein Hilfswerkzeug ***createArchitectureView*** erstellt, daß User- und Base-View eines RFGs vereinigt und das Resultat als Architektur-View im selben RFG ablegt. Zu beachten ist, daß eine Vereinigung zum Verlust von Information in der neuen View führt. Bestimmte Knotenattribute, z. B. die Sichtbarkeit oder die Klassifizierungsinformation werden pro View angegeben, so daß sie u. U. doppelt vorhanden sind. Dieses Problem besteht für den RFG an sich und ist nicht spezifisch für diese Arbeit. Bisher werden in solchen Fällen die Werte der entsprechenden Attribute aus der User-View verwendet, da sie in der Base-View derzeit als unbekannt definiert sind.

6.3.2 Auftrennen der Architektur-View in User- und Base-View

Der aus einer Rücktransformation mit Hilfe des Werkzeugs *uml2rfg* bzw. der Kombination der Werkzeuge *uml2gxl* und *gxl2rfg* neu erstellte RFG enthält zunächst ausschließlich eine einzige View, die Architektur-View, in der alle relevanten architektonischen Informationen enthalten sind. Bei den üblicherweise im Projekt *Bauhaus* verwendeten RFGs ist diese Information in mehrere Views aufgeteilt, insbesondere in die User- und die Base-View. Das Integrationswerkzeug *mergeRFGs* erwartet einen RFG in dieser Form. Daher wird ein weiteres Hilfswerkzeug benötigt, das aus einem RFG mit einer Architektur-View einen RFG mit User- und Base-View erstellt. Dieses Werkzeug wurde ***splitArchitectureView*** genannt. Nach seiner Ausführung enthält der übergebene RFG zusätzlich zu seinem vorherigen Inhalt eine User- und eine Base-View. Die Architektur-View bleibt erhalten. Enthielt der übergebene RFG bereits eine User- oder eine Base-View, so wird diese View überschrieben und eine Warnung gegeben. Knotenattribute, die im RFG pro View angegeben werden, z. B. die Sichtbarkeit des Knotens, könnten in beiden View den selben Wert erhalten. Da – wie im letzten Teilkapitel beschrieben – diese Attribute aus der User-View entnommen wurden und für die Base-View verloren gingen, kann dies streng genommen dazu führen, daß sich die Architekturbeschreibung nach der Rücktransformation geändert hat, weil Attribute in der Base-View andere Werte enthalten als zuvor. Derzeit werden die Werte dieser Attribute in der Base-View auf den definierten Wert ‘unbekannt’ gesetzt, wie dies auch bei der Erstellung des Basis-RFGs geschieht. Bei der Integration mit dem ursprünglichen RFG hat dies keine Auswirkungen, da dabei nur der Inhalt der User-View betrachtet wird. Änderungen an der schreibgeschützten Base-View sind unzulässig und dürfen nicht übernommen werden. Sie gehen verloren.

6.3.3 Einfügen zusätzlicher Abhängigkeiten

Die betrachteten UML-Werkzeuge sind nicht in der Lage, Abhängigkeiten zwischen Modellelementen, wie beispielsweise Operationen oder Attributen, zu verarbeiten, obwohl die UML diese explizit vorsieht. Solche Abhängigkeiten werden von den UML-Werkzeugen ignoriert und gehen beim Reexport verloren. Dadurch wird ein

großer Teil der transformierten Informationen nicht dargestellt und der Benutzer erkennt Zusammenhänge zwischen Elementen nicht. Beispielsweise werden keine Call-Abhängigkeiten zwischen Operationen gezeigt, der Benutzer erkennt also nicht, welche Aufrufe vorliegen. Um dieses Problem abzuschwächen, wurde ein Hilfswerkzeug ***addDependsOn*** entwickelt. Es liest einen RFG ein und fügt für alle Kanten, die auf eine ignorierte Abhängigkeit abgebildet werden würden, neue Depends_On-Kanten zwischen den ACs ein, deren Teilelemente die ursprüngliche Kante verband. Ein Beispiel zeigt Abbildung 6-6. Die Call-Kante zwischen den Unterprogrammen *X* und *Y* würde auf eine ignorierte Call-Abhängigkeit abgebildet werden. Daher fügt *addDependsOn* eine neue Depends_On-Kante ein, die zwischen der AC *A*, die *X* enthält und der AC *B*, die *Y* enthält, verläuft.

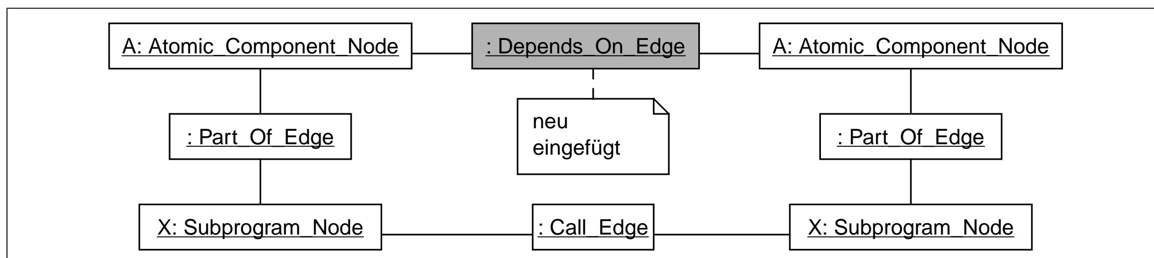


Abbildung 6-6: Einfügen einer neuen Kante durch *addDependsOn*

6.3.4 Minimieren des UML-Modells

Im Laufe dieser Diplomarbeit wurden die realisierten Werkzeuge mit verschiedenen RFGs getestet. Dabei zeigte sich, daß die produzierten XMI-Dateien sehr groß werden können. Für den größten getesteten RFG mit etwa 15000 Knoten und 100000 Kanten ergab sich eine XMI-Datei mit mehr als 100 MB Größe. Dateien dieser Größe können von den verwendeten UML-Werkzeugen nicht mehr bearbeitet werden. Daher wurde ein Werkzeug ***stripGXL*** realisiert, das die Größe der produzierten XMI-Dateien minimiert, indem es die Teile eines als GXL-Datei vorliegenden RFGs, die nicht abgebildet werden können oder deren Abbildung von den verwendeten UML-Werkzeugen ignoriert werden würde (siehe Kapitel „Notwendige Anpassungen der Transformationen“ auf Seite 89) entfernt. Mit Hilfe dieses Werkzeugs konnte die Größe der XMI-Dateien etwa halbiert werden.

6.3.5 Korrigieren der XMI-Ausgabe von *Together Control Center*

Das UML-Werkzeug *Together Control Center* hat einen Fehler, der die direkte Verwendung der XMI-Ausgabe dieses Werkzeugs verhindert. Der Fehler resultiert aus der Eigenschaft von *Together Control Center*, Attribute gleichzeitig als Assoziationen anzuzeigen. Beispielsweise wird das in Abbildung 6-7 (a) dargestellte UML-Modell in *Together Control Center* so angezeigt, wie dies Abbildung 6-7 (b) zeigt. Beim Export als XMI-Datei wird dieses Modell falsch interpretiert, so daß das in Abbildung 6-7 (c) angegebene Modell exportiert wird, bei dem ein neuer Datentyp *B* eingefügt wurde, von dessen Typ das Attribut *x* fälschlicherweise ist. Die Assoziation bleibt erhalten und hat als Ziel weiterhin die ursprüngliche Klasse *B*. Das Attribut x wurde also in ein Attribut des falschen Typs und eine Assoziation aufgespalten. Beim Reimport dieses Modells erzeugt *Together Control Center* eine weitere Assoziation, usw. Das expor-

tierte Modell weicht von dem importierten Modell ab. Es ist zudem nach der UML-Spezifikation nicht zulässig, da eine Klasse und ein Datentyp des gleichen Namens im selben Namenraum liegen.

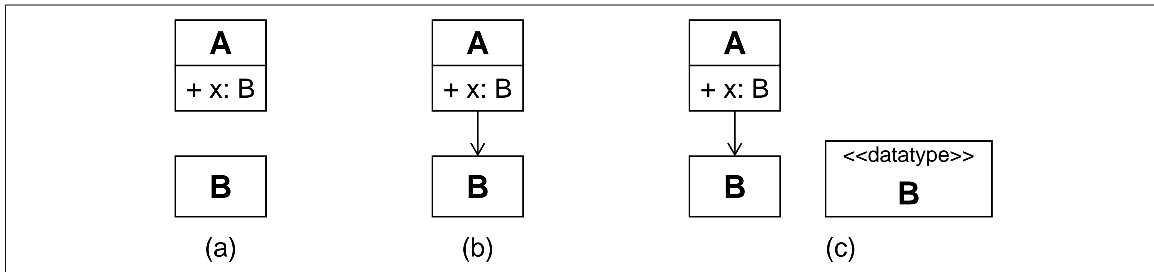


Abbildung 6-7: Fehler beim XMI-Export mit *Together Control Center*

Um diesen Fehler zu beheben, wurde ein Hilfswerkzeug ***correctTogetherUML*** erstellt. Es lädt ein UML-Modell aus einer von *Together Control Center* erstellten XMI-Datei ein und vereinigt alle gleich benannten Klassen und Datentypen. Danach paßt es den Typ aller Attribute an und entfernt überzählige Assoziationen. Das resultierende UML-Modell wird unter einem neuen Namen gespeichert und kann normal weiterverarbeitet werden.

Abschluß

Kapitel 7

7.1 Zusammenfassung

Thema dieser Diplomarbeit war die Modellierung einer in Form eines RFGs vorliegenden SW-Architekturbeschreibung mit UML-Ausdrucksmitteln. Um Konflikte mit verschiedenen in einem gemeinsamen RFG abgelegten Architekturbeschreibungen zu vermeiden, wurde festgelegt, daß alle relevanten architektonischen Informationen Teil einer speziellen Architektur-View des RFGs sind. Untersucht wurden Modellierungen für die existierenden Teilbereiche der Architekturerkennung:

- die AC-Erkennung,
- die Subsystemerkennung,
- die Konnektorenerkennung und
- die Protokollerkennung.

Es wurden jeweils geeignete Abbildungen erarbeitet, die Tabelle 7-1 zusammenfaßt. Ausgangspunkt war die Anforderung, eine Modellierung mit standardkonformer UML zu erreichen, bei der die UML ausschließlich durch hierfür in ihr vordefinierte Mechanismen, z. B. Stereotypen, ergänzt und erweitert wird. Dies ermöglicht die Darstellung der UML-Modellierung mit existierenden UML-Werkzeugen.

RFG	**UML**
View	Modell
AC	Utility
Unterprogramm	Operation & Methode
Typ	Klasse
Variable | Konstante	Attribut
Subsystem	Subsystem
Konnektor	Kollaboration
Protokoll	Activity Graph
diverse Kanten	Abhängigkeiten

Tabelle 7-1: Zuordnung zwischen Konstrukten im RFG und in der UML

Das Hauptaugenmerk dieser Arbeit lag auf der semantisch möglichst äquivalenten Modellierung der Architekturbeschreibung mit Hilfe der UML, nicht auf der tatsächlichen Präsentation dieser Modellierung. Die Präsentation unterliegt zum großen Teil der Verantwortung des verwendeten UML-Werkzeugs. Die Werkzeuge *Rational Rose*, *Together Control Center* und *Innovator* wurden genauer betrachtet. Da keines der Werkzeuge alle Anforderungen vollständig erfüllen konnte, wurde eine werkzeugunabhängige Anbindung der *Bauhaus*-Ergebnisse, bzw. des RFGs an ein UML-Werkzeug angestrebt. Hierfür wurde das Standardformat XMI verwendet. Die benötigten Werkzeuge wurden prototypisch realisiert. Um einen möglichst geringen Realisierungsaufwand zu erreichen, wurden die Transformationswerkzeuge in *Java* mit Hilfe der XML-Bibliothek *Apache Xerces for Java* implementiert. Sie greifen auf einen RFG über die in *Bauhaus* enthaltene GXL-Schnittstelle zu. Das Integrationswerkzeug konnte nur sehr eingeschränkt realisiert werden. Es setzt voraus, daß der Benutzer nur festgelegte, zulässige Manipulationen an der UML-Modellierung vornimmt. Seine Implementierung erfolgte in *Ada* mit Hilfe der Bibliothek *GroupiusSE*.

7.2 Bewertung des Projekts und der Ergebnisse

Derzeit sind nur Teile der vorgeschlagenen UML-Modellierung tatsächlich realisiert bzw. darstellbar. Dies liegt zum einen daran, daß die Form der Ergebnisse der Protokollerkennung zum Zeitpunkt der Realisierung noch nicht hinreichend festgelegt war. Zum anderen unterstützen alle betrachteten UML-Werkzeuge die UML nur teilweise, bzw. fehlerhaft. Diese Situation ist unbefriedigend. Hoffentlich werden hier in Zukunft Fortschritte erzielt.

Die Anbindung an ein UML-Werkzeug über XMI stellte sich als problematischer heraus, als zunächst gedacht. Die gesamte Layoutinformation geht hierbei verloren. Die resultierenden XMI-Dateien sind teilweise so groß, daß die verwendeten UML-Werkzeuge nicht mehr in der Lage sind, sie zu verarbeiten. Die XMI-Schnittstellen der UML-Werkzeuge sind zudem noch fehlerbehaftet. Dennoch birgt die Verwendung von XMI genug Potential, um an der getroffenen Entscheidung festzuhalten, besonders, da bisher kein eindeutig am besten geeignetes UML-Werkzeug vorliegt. Die Schwächen von XMI sind bereits Gegenstand intensiver Überarbeitungen des XMI-Standards.

Sehr viel zeitintensiver als gedacht war die Suche nach einer geeigneter XML-Bibliothek. Insbesondere die vergeblichen Versuche, die auftretenden Probleme bei der Kompilierung der Bibliotheken zu beheben, kosteten viel Zeit.

Auch die Verwendung der GXL-Schnittstelle war nicht problemlos möglich. Die Fertigstellung einer endgültigen Version der GXL verzögerte sich immer wieder und ist noch immer nicht erfolgt. Daher ist auch die Realisierung der GXL-Schnittstelle für *Bauhaus* noch nicht abgeschlossen, es existieren nur sehr prototypische Routinen für die Ein- und Ausgabe. Das verkomplizierte die Realisierung und insbesondere den Test deutlich. Im Nachhinein wäre es wohl besser gewesen, die Transformationswerkzeuge in *Ada* mit Hilfe der Bibliothek *GroupiusSE* zu realisieren und ein zwischen den Transformations- und den UML-Werkzeugen gelegenes Werkzeug

prepareXMI zu realisieren, das eine XMI-Datei in eine definierte Form bringt, insbesondere eine bestimmte Textkodierung, die dann von einfachen, rudimentären XML-Routinen eingelesen werden kann.

Viel Zeit kostete auch die Tatsache, daß eine aktuelle *Bauhaus*-Dokumentation noch immer nur teilweise existiert. Die Semantik vieler Konstrukte in Bauhaus ist eher informell festgelegt, was zu Mißverständnissen führt. In einem Fall herrschte sogar im Projektteam selbst keine Einigkeit. Empfehlenswert wäre auch ein regelmäßiges Projekttreffen, an dem auch Studenten, die in der Abteilung tätig sind, teilnehmen. Bisher ist es sehr schwer, einen Überblick über das gesamte *Bauhaus*-Projekt zu gewinnen und abzuschätzen, in welche Richtungen es sich entwickeln wird. Dies war für diese Arbeit sehr wichtig.

Dennoch war die Arbeit im Projekt *Bauhaus* sehr interessant und vergleichsweise problemlos möglich. Dies lag vor allem an der sehr guten Betreuung, die mich immer wieder zurück auf die richtige Spur brachte. Neben dem Betreuer der Arbeit waren auch alle anderen Mitglieder des Projektteams immer bereit und in der Lage, Fragen zu beantworten und Hilfestellung zu geben, was die oben genannten Schwierigkeiten deutlich abschwächte.

Sehr bewährt hat sich meiner Meinung nach die Praxis, nach etwa der Hälfte der Bearbeitungszeit einen Zwischenvortrag zu halten. Die überraschend engagierte Diskussion offenbarte viele Schwächen und Mängel in der bis zu diesem Zeitpunkt geplanten Modellierung. Insbesondere wurde die ganze Problematik der Modellierung von ACs mit Klassen offenbar. Leider führte das auch dazu, daß viele bereits getätigte Arbeiten überarbeitet oder verworfen werden mußten. Da noch genug Zeit vorhanden war, war dies aber problemlos möglich.

7.3 Ausblick

Diese Diplomarbeit kann als Ausgangspunkt der Untersuchungen angesehen werden, die eine sinnvolle, verständliche und akzeptierte Präsentation der Ergebnisse der Architekturerkennung in *Bauhaus* zum Ziel haben. Sie ist nicht deren Endpunkt. Im folgenden werden einige Punkte genannt, die weiter verfolgt werden sollten.

Verbesserung der Präsentation

Der eigentlichen Präsentation wurde im Rahmen dieser Arbeit noch nicht genügend Aufmerksamkeit geschenkt. Ein wichtiger Aspekt der UML ist es, Modelle in mehreren Diagrammen darzustellen, die sich jeweils auf bestimmte Aspekte des modellierten Systems konzentrieren. Dieser Teil der UML sollte genauer untersucht werden. Es sollten Verfahren entwickelt werden, mit denen übersichtliche Diagramme generiert werden können. Da dies wahrscheinlich nicht ohne Benutzerinteraktion möglich sein wird, sollten Präsentationsinformationen im RFG gespeichert werden. Zusätzlich ist zu untersuchen, inwiefern Präsentationsinformationen für die Architekturerkennung eingesetzt werden können. Sobald UML 2.0 mit den entsprechenden Erweiterungen des UML-Metamodells veröffentlicht wurde, kann eine Erweiterung dieser Arbeit um Präsentationselemente erfolgen.

Erweiterungen der Modellierung

Verschiedene Erweiterungen der bisherigen Modellierung sind denkbar. Eine genauere Modellierung von Protokollen steht bisher aus. Sie sollte erfolgen, sobald sich die Form von Protokollen im RFG gefestigt hat.

Der RFG soll möglichst sprachunabhängig sein. Bisher ist er teilweise (z. B. bei den Namen von Typen) aber auf die Programmiersprache *C* als Quellsprache der untersuchten Systeme ausgerichtet. Dies wird in Zukunft einige Änderungen und Erweiterungen erforderlich machen, insbesondere, wenn in objektorientierten Sprachen implementierte SW-Systeme verarbeitet werden sollen. Eine sinnvolle Anpassung und Erweiterung dieser Diplomarbeit wird wird dadurch nötig werden. Es wäre möglich, eine Klasse in einer objektorientierten Sprache als Typ abzubilden. Dabei ginge aber die Gruppierung mit den Unterprogrammen verloren. Diese könnte eine AC ergeben. Es wäre also ein denkbarer Weg, eine Klasse im SW-System auf eine AC und einen Typ abzubilden, die z. B. über eine neue Kante miteinander in Beziehung stehen. Dadurch könnten bisherige Analysen weiterverwendet werden, ohne daß Information verlorengeht. Die Modellierung mit UML sollte angepaßt werden, so daß eine Klasse im System auch wieder eine Klasse in der UML-Darstellung ergibt, indem AC und Typ verschmolzen werden, wie dies in dieser Arbeit bereits angedacht wurde (siehe Kapitel „Modellierung mit Klassen“ auf Seite 45).

Ein Sprachelement der UML, das in dieser Arbeit relativ unbeachtet blieb sind Notizen. Sie zeigen Kommentare oder auch Bedingungen. Beispielsweise werden Notizen zur Präsentation von Vor- und Nachbedingungen eingesetzt (siehe Abbildung 5-14 auf Seite 84). Sie wären auch verwendbar, um viele weitere Informationen abzubilden, z. B. Benutzerkommentare, Quelltextausschnitte, Hinweise auf unerreichbare Quelltextabschnitte, Abweichungen zwischen Soll- und Ist-Architektur oder Informationen über den Grund, aus dem bestimmte Knoten bzw. Kanten erzeugt wurden.

Unbeachtet blieben bisher Module-Knoten. Sie können über Komponenten in der UML modelliert werden (siehe auch Kapitel „Modellierung mit UML-Komponenten“ auf Seite 44). Die Modellierung der in ihnen enthaltenen Knoten und Kanten kann analog zur Modellierung des Inhalts von ACs erfolgen. Diese Abbildungen sind parallel zu den mit ACs ermittelten zu sehen, um eine genauere Sicht auf die Ist-Architektur zu vermitteln.

Verbesserung der Realisierung

Auch bei der bisher nur prototypischen Realisierung sind verschiedene Verbesserungen denkbar.

Zum einen sollte sobald möglich die Modellierung für Protokolle und Konnektoren erfolgen. Bei Protokollen muß dazu zunächst die Implementierung der Protokollerkennung abgeschlossen werden. Bevor Konnektoren tatsächlich modelliert werden können, muß ein UML-Werkzeug gefunden werden, das in der Lage ist, die Resultate in Form generischer Kollaborationen darzustellen.

XMI 1.1 wird das Problem der großen Dateien abschwächen. Sobald die neue Version des Standards endgültig verabschiedet wurde und UML-Werkzeuge sie unterstützen, wäre eine – eventuell zusätzliche – Verwendung denkbar.

Mittelfristig wird in einem RFG wahrscheinlich mehr als eine Architektur gleichzeitig vorliegen, z. B. Ist- und Soll-Architektur. Eine statische Festlegung der Namen der zu bearbeitenden Views, wie diese bisher erfolgt, ist dann nicht mehr sinnvoll. Die Werkzeuge sollten so abgeändert werden, daß sie die Namen der Views als Parameter erhalten.

Obwohl die in dieser Arbeit realisierten prototypischen Werkzeuge durchaus auch für große Systeme einsetzbar sind, könnte es von Vorteil sein, in einiger Zeit eine erneute Realisierung anzustreben. Diese kann die bis dahin gemachten Erfahrungen einbeziehen und direkt in *Ada* mit Hilfe von *GroupiusSE* implementiert werden, wenn sich bis zu diesem Zeitpunkt eine geeignete XML-Bibliothek fand. Die Entwicklung der XML-Bibliothek *expat* 2.0 (siehe Kapitel „Realisierung in Ada“ auf Seite 90) sollte hierbei verfolgt werden. Bei der Realisierung der GXL-Schnittstelle in *Bauhaus* wird sich ebenfalls die Schwierigkeit ergeben, XML-Dateien einlesen bzw. schreiben zu müssen. Falls sich dort eine einfache Lösung findet, kann diese in dieser Arbeit wiederverwendet werden. Gleichzeitig würde dadurch u. U. auch ein etwas größerer Aufwand für die Anbindung einer XML-Bibliothek lohnenswert. Die Verwendung von *GroupiusSE* hätte den Vorteil, daß Änderungen am RFG nicht gesondert nachvollzogen werden müßten.

Warnungskonzept

Im Verlauf dieser Arbeit wurde ein Warnungskonzept angedacht. Allerdings zeigte sich, daß eine sinnvolle Ausgestaltung sehr viel mehr Zeit in Anspruch nehmen würde, als zur Verfügung stand. Es muß abgesichert werden, daß ein Benutzer nicht für Sachverhalte, die er in einem früheren Schritt der Arbeiten bereits bestätigt hat, wieder Warnungen erhält. Gleichzeitig soll es möglich sein, bereits erledigte Warnungen wieder zu aktivieren, z. B. weil ein anderer Benutzer die Arbeiten übernimmt. Dazu muß das Warnungskonzept in alle Phasen der durch Bauhaus unterstützten Arbeiten integriert werden. Warnungen könnten beispielsweise erzeugt werden, wenn:

- in der Architektur Problemstellen gefunden werden, z. B. Abweichungen zwischen Soll- und Ist-Architektur oder Fehler, die bei der Wartung entstanden (siehe auch [Eis99]),
- eine Analyse unsichere Ergebnisse weitergibt oder
- bei der Modellierung in UML semantische Ungenauigkeiten auftreten.

Das Ziel ist es, den Benutzer zunächst auf wichtige, bzw. problematische Stellen in der Architekturbeschreibung hinzuweisen, damit eine effizientere Bearbeitung möglich wird. Eine Modellierung solcher Warnungen in UML könnte z. B. über

- die Farbgebung der dargestellten Modellelemente oder
- die oben angesprochenen Notizen

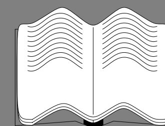
erfolgen.

Architekurbeschreibungssprachen

Interessant könnte auch eine Untersuchung verschiedener ADLs sein. Sie sind eher für die exakte Festlegung einer Architektur geeignet, als für eine übersichtliche und leicht verständliche Darstellung, wofür die UML optimiert wurde. Als sinnvoll könnte sich eine Kombination dieser Ansätze erweisen. In einem ersten Schritt erfolgt hierbei eine Formulierung der in einem RFG enthaltenen Architekturbeschreibung mit einer geeigneten ADL. Danach wird diese in die UML transformiert. (siehe [Rob98])

Literaturverzeichnis

Quellen



[Blo00] Bloch, J. : *Effective Java Programming;* Online-Vorveröffentlichung von Teilen des gleichnamigen, bisher nicht erschienenen Buches; http://developer.java.sun.com/developer/Books/shiftintojava/ zuletzt besucht: 2000-11-07.

[Boo94] Booch, G. : *Object-oriented analysis and design with applications;* zweite Ausgabe, The Benjamin/ Cummings Publishing Company Inc., 1994.

[Boo99] Booch, G., Rumbaugh, J, Jacobson, I. : *The Unified Modeling Language User Guide;* Addison Wesley, 1999.

[CoaYou91] Coad, P., Yourdan, E. : *Object-oriented Analysis;* zweite Ausgabe, Yourdan Press, Englewood Cliffs, N.J., 1991.

[Coa99] Coad, P., Lefebvre, E., De Luca, J. : *Java Modeling In Color With UML: Enterprise Components and Process;* Prentice Hall, 1999.

[Cze00] Czeranski, J., Eisenbarth, T., Kienle, H., Koschke, R., Simon, D. : *Wiedergewinnung von Architekturinformationen: Ausblicke;* erscheint in: 2. Workshop Software-Reengineeering (WSRE2000), 11. - 12. Mai, Bad Honnef, Deutschland, 2000.

[Cze00b] Czeranski, J., Eisenbarth, T., Kienle,H., Koschke, R., Simon, D. : *Analyzing xfig Using the Bauhaus Tool;* internes Arbeitspapier, Institut für Informatik, Universität Stuttgart, zur Präsentation auf der 7. Working Conference on Reverse Engineering (WCRE2000), 23. - 25. November, Brisbane, Australien, 2000.

[Eis98] Eisenbarth, T. : *GropiusSE - Eine Resource Flow Graph Bibliothek in Ada95 für das Speichern und Aufbereiten von Reengineringinformationen;* Studienarbeit Nr. 1663, Institut für Informatik, Universität Stuttgart, 1998.

[Eis99] Eisenbarth, T., Koschke, R., Plödereder, E., Girard, J. F., Würthner, M. : *Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen;* Workshop Software-Reengineering (WSRE99), Bad Honnef, Universität Koblenz-Landau, 27. - 28. Mai 1999, Fachberichte Informatik, Nr. 7-99, Seite 17 - 26, 1999.

[Eis00] Eisenbarth, T. : *Applying Pattern-Matching to Recover Mid-Level Connectors from Source Code;* internes Arbeitspapier, Institut für Informatik, Universität Stuttgart, 2000.

[expat] *expat;* http://www.jclark.com/xml/expat.html zuletzt besucht: 2000-10-07.

[Fli00] Flint, S., Dobbing, B. : *Using Java APIs with Native Ada Compilers;* 5. in: International Conference on Reliable Software Technologies Ada-Europe 2000, 26. - 30. Juni, Potsdam (Berlin), Deutschland, LNCS, Vol. 1845, S. 41 - 55, Springerverlag, 2000.

[FowSco99] Fowler, M., Scott, K. : *UML Distilled: Applying the Standard Object Modeling Language;* Addison Wesley, 1999.

[GirKos97] Girard, J. F., Koschke, R. : *Finding Components in a Hierarchy of Modules: a Step Towards Architectural Understanding;* in: International Conference on Software Maintenance (ICSM97), 29. - 30. September, 1. - 3. Oktober 1997, Bari, Italien, S. 58 - 65, IEEE Computer Society Press, 1997.

[GirKos98] Girard, J. F., Koschke, R. : *An Intermediate Representation for Integrating Reverse Engineering Analyses;* Proceedings of the 5th Working Conference on Reverse Engineering (WCRE98), 12 - 14 Oktober 1998, Honolulu, Hawai, USA, S. 241 - 250, IEEE Computer Society Press, 1998.

[gxl] *GXL - Graph Exchange Language;* http://www.gupro.de/GXL zuletzt besucht: 2000-10-31.

[Han00] Hanssen, S. : *Extraktion statischer Traces zur Wiedergewinnung von Protokollen;* Studienarbeit, Institut für Informatik, Universität Stuttgart, 2000.

[Hei00] Heiber, T. : *Semi-automatische Herleitung von Komponentenprotokollen aus statischen Verwendungsmustern;* Diplomarbeit Nr. 1822, Institut für Informatik, Universität Stuttgart, 2000.

[Hol00] Holt, R. C., Winter, A., Schürr, A. : *GXL: Towards a Standard Exchange Format;* erscheint in: Proceedings of the 7th Working Conference on Reverse Engineering (WCRE2000), 23. - 25. November 2000, Brisbane, Australien, IEEE Computer Society Press, 2000.

[Jac92] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G. : *Object-Oriented Software Engineering: A Use Case Driven Approach;* Addison-Wesley, 1992.

[Jac99] Jacobson, I., Booch, G., Rumbaugh, J. : *The Unified Software Development Process;* Addison-Wesley, 1999.

[jgnat] *JGnat;* http://www.gnat.com/texts/products/prod_java.htm zuletzt besucht: 2000-10-29.

[KerRit90] Kernighan, B. W., Ritchie, D. M. : *Programmieren in C: mit dem C-Reference-Manual in deutscher Sprache;* 2. Ausgabe, ANSI C, Hanser, Prentice-Hall International, 1990.

[KosPlö96] Koschke, R., Plödereder, E. : *Ansätze des Programmverstehens.* in: Franz Lehner (Hrsg.): Softwarewartung und Reengineering, Gabler Edition Wissenschaft, Deutscher Universitätsverlag, S. 159 - 176, 1996.

[Kos00] Koschke, R. : *Atomic Architectural Component Recovery for Program Understanding and Evolution;* Dissertation, Institut für Informatik, Universität Stuttgart, 2000.

[libxml] *The XML C library for Gnome;* http://www.xmlsoft.org zuletzt besucht: 2000-10-22.

[MedTay00] Medvidovic, N., Taylor, R. : *A Classification and Comparison Framework for Software Architecture Description Languages;* IEEE Transactions on Software Engineering, 26 (1), S. 70 - 93, 2000.

[mof] *Meta-Object Facility (MOF) V1.3;* ftp://ftp.omg.org/pub/docs/formal/00-04-03.pdf zuletzt besucht: 2000-09-09.

[Oes99b] Oesterreich, B. : *Wie setzt man Use-Cases wirklich sinnvoll zur Anforderungsanalyse ein;* in: OBJEKTspektrum, 1/ 1999, SIGS Conferences GmbH, 1999.

[omgspecs] *OMG Modeling Specifications Available Electronically;* http://www.omg.org/technology/documents/formal/omg_modeling_specifications_avai.htm zuletzt besucht: 2000-09-09.

[projectX] *Java Project X: Technology Release 2;* http://developer.java.sun.com/developer/products/xml/ zuletzt besucht: 2000-10-31.

[Rob98] Robbins, J. E., Medvidovic, N., Redmiles, D. F., Rosenblum, D. S. : *Integrating Architecture Description Languages with a Standard Design Method;* in: Proceedings of the 20th International Conference on Software Engineering (ICSE98), 19. - 25. April 1998, Kyoto, Japan, Seite 209 - 218, IEEE Computer Society Press, 1998.

[Roh98] Rohrbach, J. : *Erweiterung und Generierung einer Zwischensprache für C-Programme;* Studienarbeit Nr. 1662, Institut für Informatik, Universität Stuttgart, 1998.

[Rum91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. : *Object-Oriented Modeling and Design;* Prentice Hall, Englewood Cliffs, N.J., 1991.

[Rum99] Rumbaugh, J., Jacobson, I., Booch, G. : *The Unified Modeling Language Reference Manual;* Addison Wesley, 1999.

[ShaGar96] Shaw, M., Garlan, D. : *Software Architecture: Perspectives on an Emerging Discipline;* Prentice Hall, Englewood Cliffs, N.J., 1996.

[ShlMel88] Shlaer, S., Mellor, S.J. : *Object-Oriented Systems Analysis: Modeling the World in Data;* Yourdon Press, Englewood Cliffs, N.J., 1988.

[Til93] Tilley, S. R., Müller, H. A., Withney, M. J., Wong, K. : *Domain-Retargetable Reverse Engineering;* in: Proceedings of the International Conference on Software Maintenance (ICSM93), Montréal, Québec, Canada, 27. - 30. September 1993, S. 142 - 151, IEEE Computer Society Press, 1993

[uml1.3] *Unified Modeling Language (UML) V1.3;* ftp://ftp.omg.org/pub/docs/formal/00-03-01.pdf zuletzt besucht: 2000-09-09.

[umlglossar] *Glossar + Notationsübersicht UML 1.3;* http://www.job-agent.de/download/glossar.pdf zuletzt besucht: 2000-11-07.

[umlnews] *Unified Modeling Language (UML) Resource Center;* http://www.jeckle.de/unified.htm zuletzt besucht: 2000-10-03.

[umltools] *UML Tools (Case & Drawing);* http://www.jeckle.de/umltools.htm zuletzt besucht: 2000-10-09.

[xercesC++] *Xerces C++ Parser;* http.//xml.apache.org/xerces-c/index.html zuletzt besucht: 2000-10-07.

[xercesJ] *Xerces Java Parser Readme;* http://xml.apache.org/xerces-j/index.html zuletzt besucht: 2000-10-07.

[xmi] *XML Metadata Interchange (XMI) V1.0;* ftp://ftp.omg.org/pub/docs/formal/00-06-01.pdf zuletzt besucht: 2000-09-09.

[xml] *Extensible Markup Language (XML) 1.0 (Second Edition);* http://www.w3.org/TR/2000/REC-xml-20001006 zuletzt besucht: 2000-10-07.

[xml4j] *XML Parser for Java: another alphaWorks technology;* http://www.alphaworks.ibm.com/tech/xml4j zuletzt besucht: 2000-10-22.

[xmltools] *Free XML tools and software;* http://www.garshol.priv.no/download/xmltools/ zuletzt besucht: 2000-10-07.

[xp] *XP - an XML Parser in Java;* http://www.jclark.com/xml/xp/index.html zuletzt besucht: 2000-10-07.

[xslt] *XSL Transformations (XSLT);* http://www.w3.org/TR/xslt zuletzt besucht: 2000-10-31.

Hiermit versichere ich, daß ich diese Arbeit selbstständig verfaßt und bei der Erstellung nur die angegebenen Hilfsmittel verwendet habe.

____

Gregor Schiele