# Generierung interaktiver Informationssysteme und ihrer Benutzungsoberflächen für mehrere Benutzer

Alfons Brandl

Institut für Informatik
der Technischen Universität München

# Generierung interaktiver Informationssysteme und ihrer Benutzungsoberflächen für mehrere Benutzer

Alfons Brandl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Peter Paul Spies

Prüfer der Dissertation: 1. Univ.-Prof. Dr. Dr. h.c. Jürgen Eickel

2. Univ.-Prof. Dr. Johann Schlichter

Die Dissertation wurde am 17.01.2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 08.06.2002 angenommen.

# Kurzfassung

Ziel dieser Arbeit ist es zu zeigen, wie aus formalen Modellen der interaktiven Aspekte eines Informationssystems automatisch ein ablauffähiges System mit den zugehörigen Benutzungsoberflächen generiert werden kann. Dabei werden interaktive Informationssysteme betrachtet, die mehrere Benutzer bei der Verwaltung beliebig komplex strukturierter Daten sowie bei der Ausführung gemeinsamer Aufgaben unterstützen. Die generierten Benutzungsoberflächen passen sich dynamisch auf die von den jeweiligen Benutzern zu verwaltenden Daten und der von ihnen zu erledigenden Aufgaben an.

Um die Anwendbarkeit dieses Ansatzes zu zeigen, erfolgt die Entwicklung und prototypische Implementierung eines entsprechenden Generierungskonzepts. Dazu gehört die Entwicklung einer deklarativen Eingabesprache, mit der die interaktiven Aspekte eines Informationssystems kompakt beschrieben werden können und eines Generators, der daraus ablauffähige Prototypen generiert.

Die Eingabe besteht aus dem Daten-, Aufgaben- und Benutzermodell. Das Datenmodell ist eine Datentypbeschreibungssprache, die dem Entwickler neben üblichen Kompositionstypen, wie Tupel oder Variante, auch Referenzen und Prädikate zur Spezifikation komplexer Plausibilitäten anbietet. Im Benutzermodell werden Benutzertypen und ihre Sichtweise auf Daten und Aufgaben definiert. Das Aufgabenmodell umfasst einfache Aktionen und aus mehreren Aktionen gebildete komplexe Aufgaben, an denen auch mehrere Benutzer arbeiten können. Ferner wird im Generierungskonzept die Integration der Modelle berücksichtigt ("Welche Benutzer sehen welche Daten zu welchen Phasen der Aufgabenbearbeitung?").

Um Aufgaben flexibel graphisch zu modellieren, können anwendungsspezifische graphische Aufgabenmodellierungseditoren aus der Beschreibung der abstrakten Syntax und der graphischen Präsentation der jeweiligen Aufgabenmodellierungssprache generiert werden.

Im Vergleich zu verwandten Arbeiten ermöglicht das prototypisch implementierte Generierungskonzept dieser Arbeit erstmalig, mehrere Benutzer und ihre Eigenschaften deklarativ zu beschreiben und dies automatisch bei der dynamischen Anpassung der Benutzungsoberflächen zu berücksichtigen. Weiterhin neuartig ist die Möglichkeit, generierte graphische Aufgabenmodellierungseditoren in die generierten, ablauffähigen Prototypen interaktiver Informationssysteme für mehrere Benutzer zu integrieren.

# Danksagung

Ich danke Herrn Prof. Eickel für die Möglichkeit, an seinem Lehrstuhl promovieren zu können und für die zahlreichen Anregungen zu dieser Arbeit. Ferner hat er mir bereits seit den Zeiten der Einführungsvorlesung eine abstrakte Sichtweise auf Problemstellungen der Informatik vermittelt. Nur durch eine angemessene Abstraktion war es möglich, das in dieser Arbeit ausgearbeitete Generierungskonzept in seiner Allgemeinheit zu entwickeln und auch prototypisch zu implementieren. Herrn Prof. Schlichter danke ich für wichtige Hinweise und Anmerkungen zu einer vorläufigen Version der Arbeit und für die Übernahme des Zweitgutachtens.

Matthias Göbel, Franz Haßmann und Riitta Höllerer danke ich für das Durchlesen und Korrigieren der Arbeit. Auch den weiteren Mitarbeitern am Lehrstuhl für Informatik II, Roland Haratsch, Aurel Huber, Herta Remmes und Hans Wittner gebührt mein Dank für die gute Zusammenarbeit und Unterstützung. Zahlreiche Fachgespräche mit Matthias Göbel, Roland Haratsch, Riitta Höllerer und Aurel Huber hatten einen wichtigen Einfluss auf die Arbeit.

Gerwin Klein danke ich für die ideenreiche Entwicklung und sorgfältige Implementierung der Werkzeuge GRACE, CLASSGEN und JFlex, die ich bei meiner Implementierung benutzt habe. Insbesondere GRACE spielt nicht nur bei der Implementierung, sondern auch bei dem in dieser Arbeit entwickelten Generierungskonzept eine wichtige Rolle.

Einen weiteren bedeutsamen Einfluss auf dieses Generierungskonzept hatte die Dissertation von Siegfried Schreiber, dem ich dafür danke, mir durch seine Arbeit einen sehr genauen Einblick in die Spezifikations- und Generierungstechniken für graphische Benutzungsoberflächen verschafft zu haben.

Ganz besonders möchte ich meiner Mutter dafür danken, dass sie mir meinen Ausbildungsweg ermöglicht hat.

Abschließend danke ich meiner Frau Marion Sperle für ihre Bereitschaft, viele fachliche Diskussionen auch in der Freizeit ertragen zu haben, für das sehr gründliche Durchlesen der Arbeit und für eine Reihe inhaltlicher Anmerkungen, die sehr zum Gelingen der Arbeit beigetragen haben.

# Inhaltsverzeichnis

**1 Einführung** ..... **5**
1.1 Motivation ..... 5
1.2 Computergestütztes Design von Benutzungsoberflächen ..... 6
1.3 Spezifikation interaktiver Informationssysteme ..... 8
1.3.1 Datenaspekt ..... 9
1.3.2 Aufgabenaspekt ..... 10
1.3.3 Benutzeraspekt ..... 11
1.4 Generierungskonzept im Überblick ..... 12
1.5 Aufgabenstellung der Arbeit ..... 14
1.6 Ergebnisse ..... 14
1.7 Inhalt und Lesehinweise ..... 15
1.7.1 Inhalt ..... 15
1.7.2 Lesehinweise ..... 16

**2 Interaktive Informationssysteme** ..... **17**
2.1 Benutzung interaktiver Informationssysteme ..... 17
2.1.1 Formulare ..... 18
2.1.2 Grapheditoren ..... 19
2.1.3 Persönliches Rollen-Management ..... 20
2.1.4 Benutzung von Datenbank- und Workflow-Management-Systemen ..... 20
2.2 Architekturen interaktiver Informationssysteme ..... 23
2.2.1 Seeheim-Modell ..... 23
2.2.2 MVC-Architektur ..... 25
2.2.3 HIT-Instanzen ..... 25
2.3 Entwicklung interaktiver Informationssysteme ..... 27
2.3.1 Entwicklungsphasen ..... 27
2.3.2 Werkzeuge ..... 28

**3 EMU-Systeme** ..... **33**
3.1 Benutzung von EMU-Systemen ..... 33
3.1.1 Konkrete Ausführung mit Formularen und Grapheditoren ..... 33
3.1.2 Mehrbenutzerzugriff ..... 34
3.1.3 Systemanmeldung ..... 34
3.2 Architektur von EMU-Systemen ..... 34
3.2.1 Form- und Graphadapter ..... 35
3.2.2 Zugriffsadapter ..... 36
3.2.3 Layoutattribute ..... 37
3.3 EMU-Generierungskonzept ..... 38
3.4 Entwicklungsszenarien im Überblick ..... 40
3.4.1 Entwicklung von Mehrbenutzeranwendungen ..... 40
3.4.2 Entwicklung von Grapheditoren für Diagrammsprachen ..... 41

**4 Datenmodell** ..... **43**
4.1 Sprachelemente ..... 43
4.1.1 Tupelproduktionen ..... 45
4.1.2 Variantenproduktionen ..... 45

4.1.3 Listen und Listenproduktionen ... 46
4.1.4 Referenzenproduktionen ... 46
4.1.5 Bedingungen ... 47
4.1.6 Implizite Ergänzungen ... 47
4.1.7 Graphische Darstellung ... 47
4.2 Beispielausführungen generierter EMU-Systeme ... 47
4.2.1 Abstrakte Ausführung ... 52
4.2.2 Ausführung mit Formularen ... 56
4.2.3 Ausführung mit Grapheditoren ... 59
4.3 Semantik des EMU-Datenmodells ... 66
4.3.1 Zustand eines EMU-Systems ... 67
4.3.2 Interaktionen auf EMU-Systeme ... 68
4.3.3 Zustandsübergangssystem ... 71
4.4 Begründung und mögliche Alternativen ... 72
4.4.1 Warum ein hierarchisches Datenmodell? ... 72
4.4.2 Alternative Datenmodelle ... 74
4.4.3 Algebraische Spezifikationen ... 74
4.4.4 Klassendiagramme ... 75
4.4.5 Relationenmodell ... 77

**5 Aufgaben- und Benutzermodell ... 79**
5.1 Sprachelemente ... 79
5.1.1 Aktionen ... 82
5.1.2 Aktivitätsnetze ... 85
5.1.3 Benutzertypen ... 87
5.1.4 Zugriffe ... 88
5.1.5 Implizite Ergänzungen ... 90
5.1.6 Graphische Darstellung durch TUMs ... 91
5.2 Beispielausführungen generierter EMU-Systeme ... 95
5.2.1 Abstrakte Ausführung ... 99
5.2.2 Ausführung mit Formularen ... 100
5.2.3 Ausführung mit Grapheditoren ... 101
5.3 Semantik der EMU-Eingabemodelle ... 105
5.3.1 Zustandserweiterungen ... 106
5.3.2 Semantik von Bedingungen und Pfadausdrücken ... 109
5.3.3 Erweiterte Interaktionen ... 111
5.3.4 Erweitertes Zustandsübergangssystem ... 112
5.4 Begründung und mögliche Alternativen ... 113
5.4.1 Warum kein Dialogmodell? ... 113
5.4.2 Alternative Aufgabenmodelle ... 114

**6 Entwicklung von EMU-Systemen ... 117**
6.1 Vorgehensweise ... 118
6.2 Beispiele ... 119
6.2.1 EMU-System *TUM* ... 119
6.2.2 EMU-System *Geldausgabeautomat* ... 121
6.2.3 EMU-System Talk ... 126
6.2.4 EMU-System *Praktikum* ... 128
6.2.5 EMU-System *Ad-hoc Workflows* ... 132

**7 Verwandte Arbeiten ........ 141**
7.1 FUSE ........ 142
7.2 JANUS ........ 145
7.3 TEALLACH ........ 147
7.4 MASTERMIND ........ 149
7.5 MOBI-D ........ 150
7.6 Zusammenfassung des Vergleichs ........ 152
**8 Zusammenfassung und Ausblick ........ 153**
8.1 Ergebnisse ........ 153
8.2 Weitere Arbeiten ........ 154
**Anhang ........ 157**
**Literaturverzeichnis ........ 163**

# 1 Einführung

Im Einführungskapitel wird nach einem motivierenden Abschnitt die Aufgabenstellung konkretisiert. Dazu erfolgt ein kurzer Überblick über das umgebende Forschungsgebiet CADUI im Abschnitt 1.2, sowie die Klärung grundlegender Begriffe, die für das Verständnis der Aufgabenstellung von Bedeutung sind.

## 1.1 Motivation

Bei der Modellierung von Informationssystemen stand lange Jahre der statische, d.h. datenorientierte Aspekt eines Systems im Mittelpunkt. Durch das Entity-Relationship-Modell (ER-Modell, [Che76]) stand früh eine graphische Modellierungssprache auf hoher Abstraktionsebene für diese Aufgabe zur Verfügung. Die Akzeptanz des ER-Modells ist unter anderem darauf zurückzuführen, dass es algorithmische Verfahren zur Überführung eines ER-Modells in ein relationales Datenbank-Schema [Cod70] gibt. Dieses wiederum wird automatisch auf geeignete effiziente Datenstrukturen abgebildet. Dadurch entfällt eine Implementierung auf programmiersprachlicher Ebene, was eine erhebliche Arbeitserleichterung darstellt.

Diese Möglichkeiten stehen bei der Entwicklung ***interaktiver Systeme*** erst in vergleichsweise beschränktem Maße zur Verfügung. Interaktive Systeme sind stark durch die Interaktionsmöglichkeiten der Benutzer geprägt. Diese Interaktionsmöglichkeiten werden durch ***Benutzungsoberflächen*** (***UI***, *u*ser *i*nterfaces) realisiert, die nach [End94] als die für die Benutzer sichtbaren Teile interaktiver Systeme aufgefasst werden.

Es gibt zwar Workflow-Management-Systeme (WFMS, wie z.B. FlowMark [JBS97]), welche die Möglichkeit zur Modellierung dynamischer und auch interaktiver Aspekte eines Informationssystems bieten, jedoch können aus den jeweiligen Modellen keine ausführbaren Benutzungsoberflächen generiert werden. Der Grund dafür liegt unter anderem darin, dass zur Entwicklung ausführbarer, benutzerfreundlicher Benutzungsoberflächen eine Vielfalt weiterer Aspekte bzw. Modelle berücksichtigt werden müssen.

Diese Aspekte zu erkennen, zu klassifizieren, entsprechende Modellierungssprachen und zugehörige Werkzeuge zu entwickeln, ist die Aufgabe des Forschungsgebiets CADUI (Computer-Aided Design of User Interface) [VP99], einem Teilgebiet der HCI (Human Computer Interaction) [HCI01].

## 1.2 Computergestütztes Design von Benutzungsoberflächen

In CADUI werden laut [VP99] folgende, teilweise voneinander abhängige Modelle verwendet, um interaktive Applikationen mit Benutzungsoberflächen zu erstellen:

i. Ein ***Aufgabenmodell*** modelliert die Aufgaben und Ziele des Endbenutzers. Ziele werden durch die Ausführung atomarer oder zusammengesetzter Aufgaben erreicht.

ii. Ein ***Datenmodell*** modelliert die Daten, welche die Benutzer über die Benutzungsoberfläche manipulieren.

iii. Ein ***Benutzermodell*** modelliert die Eigenschaften und Rollen des Benutzers in Bezug auf die Kommunikation mit der interaktiven Anwendung. Ein Benutzermodell kann spezifizieren, wie individuelle Benutzungsoberflächen erstellt, neu konfiguriert (z.B. wenn sich die Rolle des Benutzers geändert hat), adaptive Benutzungsoberflächen bereitgestellt und eine benutzerspezifische Hilfe und Führung ermöglicht werden.

iv. Ein ***Kontrollmodell*** modelliert die Dienste, die von der interaktiven Anwendung angeboten werden. Es umfasst üblicherweise objektorientierte Sprachelemente. Dabei sind Instanzmethoden direkt von Benutzern ausführbar und entsprechen den atomaren Aufgaben des Aufgabenmodells.

v. Ein ***Präsentations- oder Layoutmodell*** modelliert die Darstellung der Objekte des Kontrollmodells und ihrer Methoden mit Hilfe der ***Interaktionsobjekte*** des zugrundeliegenden Systems (z.B. Buttons).

vi. Ein ***Konversationsmodell***[1] modelliert die Kommunikation zwischen Benutzern und der interaktiven Anwendung.

vii. Ein ***Verhaltensmodell*** kann benutzt werden, um das Eingabeverhalten von Benutzern zu spezifizieren. Die gleichzeitige Verwendung eines Präsentationsmodells und eines Verhaltensmodells erlaubt die unabhängige Spezifikation der Präsentation und des dynamischen Verhaltens einer Benutzungsoberfläche.

viii. Ein ***Plattformmodell*** kann benutzt werden, um plattformspezifische Eigenschaften, wie Ein-/Ausgabegeräte zu spezifizieren.

ix. Ein ***Umgebungsmodell*** kann Arbeitsplatzeigenschaften, wie kulturelle oder benutzersprachliche Aspekte spezifizieren.

Werkzeuge, mit deren Hilfe man aus obigen Modellen (oder Teilen davon) interaktive Applikationen mit Benutzungsoberflächen erstellen kann, werden in der Literatur ***MB-UIDE*** (Model-Based User Interface Design Environment) genannt. Wir werden dafür in

[1] Das Konversationsmodell verallgemeinert das in der Literatur ebenfalls bekannte ***Dialogmodell*** ([Sch97]) auf mehrere Kommunikationspartner.

dieser Arbeit den Begriff ***CADUI-Werkzeug*** verwenden, weil wir uns primär für die Generierungsfunktionalität eines Werkzeugs interessieren und weniger für die Entwicklungsumgebung, d.h. die Benutzungsoberfläche, des Werkzeugs selbst. Es werden in der Arbeit jedoch einige Beispiele vorgeführt, die zeigen, wie Benutzungsoberflächen für CADUI-Werkzeuge (also MB-UIDEs) aus der formalen Beschreibung ihrer Modellierungssprachen generiert werden können.

Aus Sicht der Entwickler, die mit CADUI-Werkzeugen arbeiten, unterscheiden sich diese Werkzeuge etwa darin, welche Modelle und Entwicklungsphasen verwendet werden und welcher Automatisierungsgrad vorliegt. Ähnlich zu allgemeinen, nicht benutzungsoberflächenspezifischen Software-Entwicklungsprozessen wird bei CADUI eine Software-Entwicklung in mehreren Entwicklungsphasen [VP99, LS96] propagiert. Dabei werden beim Phasenübergang abstraktere (problemnahe) Modelle in weniger abstrakte (implementierungsnähere) Modelle transformiert. Bei CADUI-Werkzeugen mit hohem Automatisierungsgrad erfolgen diese Transformationen weitgehend automatisiert. Im Allgemeinen gibt es schon innerhalb einer Phase mehrere Modelle, aus denen zusammen ein implementierungsnäheres Modell gewonnen wird. Oft sind dies unter anderem das Daten- und das Aufgabenmodell zur Analysephase und das Präsentations- und Dialogmodell zur Designphase [LS96, Sch97]. Abbildung 1-1 soll diese Arbeitsweise von CADUI-Werkzeugen verdeutlichen.

Die Aufgabe, der sich die vorliegende Arbeit widmet, ist die Erstellung eines CADUI-Werkzeugs mit einem hohen Automatisierungs- und Abstraktionsgrad. Das bedeutet, aus problemnahen Modellen soll automatisch ein ausführbares, interaktives System mit graphischer Benutzungsoberfläche erzeugt werden.

Bei der Konzeption eines CADUI-Werkzeugs ist zu überlegen, wie der Entwickler Bezüge zwischen den Modellen herstellen kann. Sehr mächtige CADUI-Werkzeuge, wie das BOSS-System [Sch97], ermöglichen es, den Bezug zwischen einzelnen Modellen auf einer abstrakten Ebene herzustellen. Dies ermöglicht es beim BOSS-System beispielsweise, für eine ganze Klasse von interaktiven Applikationen das Layout der entsprechenden Benutzungsoberflächen zu beschreiben und nicht etwa nur das einer speziellen Anwendung.

Eine weitere Aufgabe, die sich bei der Konzeption eines CADUI-Werkzeugs ergibt, ist die Definition der einzelnen Modellierungssprachen. Für jede Modellierungssprache ist die abstrakte und konkrete Syntax sowie eine Semantik festzulegen, welche die Überführung der verschiedenen Modelle in ein ausführbares System gestattet.

Um die Aufgabenstellung der vorliegenden Arbeit genauer zu verstehen, sind demnach folgende Fragen zu klären:

- Welche Anforderungen werden an interaktive Informationssysteme und ihre Benutzungsoberflächen und damit auch indirekt an die Spezifikationsmöglichkeiten eines CADUI-Werkzeugs gestellt?
- Welche Modelle und Generierungsphasen sind notwendig?
- Wie nehmen die Modelle aufeinander Bezug?

Der Beantwortung dieser Fragen widmet sich der folgende Abschnitt.

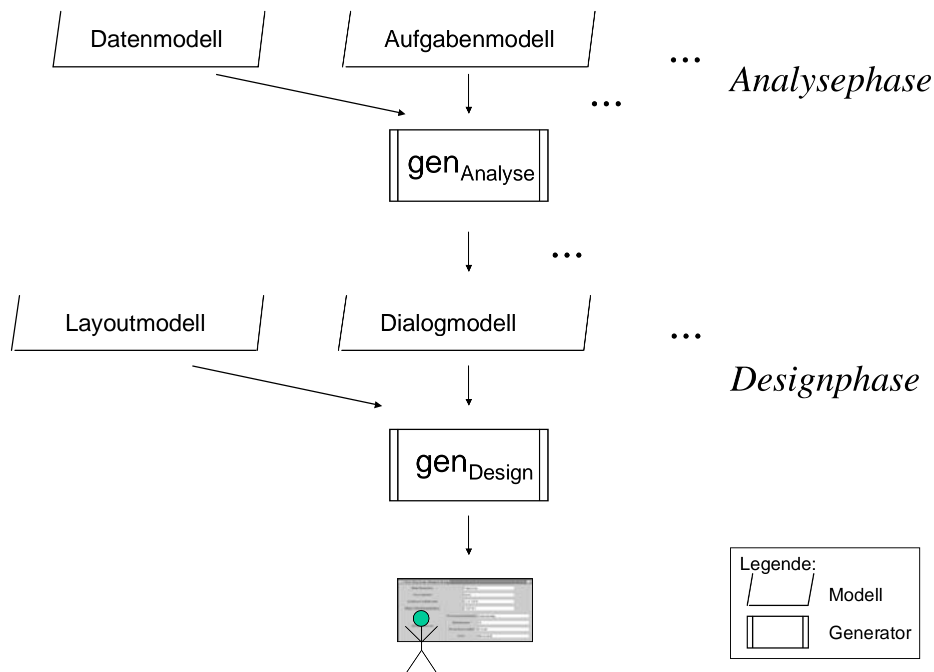


**Abbildung 1-1: Arbeitsweise von CADUI-Werkzeugen (nach [LS96, VP99])**

## 1.3 Spezifikation interaktiver Informationssysteme

Zunächst soll durch eine Begriffsklärung motiviert werden, welche grundlegenden Anforderungen an die "Generierung interaktiver Informationssysteme und ihrer Benutzungsoberflächen für mehrere Benutzer" zu stellen sind. Ein (interaktives) Informationssystem wird in [Vos00] wie folgt definiert:

**Definition 1-1: Informationssystem (nach [Vos00])**

*Ein Informationssystem ist ein Werkzeug zur Erfassung und Kommunikation von Information zum Zwecke der Erfüllung der Anforderungen seiner Benutzer, der (Geschäfts-) Aktivitäten ihres Unternehmens und zur Erreichung der Unternehmensziele. Ein Informationssystem unterstützt die Unternehmensaktivitäten durch Bereitstellung der benötigten Informationen oder durch Automatisierung der mit den Aktivitäten zusammenhängenden Vorgänge. Es umfasst sämtliche, zu diesem Zweck im Unternehmen vorhandenen Ressourcen, d.h. die Daten, die Datenbank-Software, die nötige Rechner-Hardware, die Personen, welche die Daten benutzen und verwalten, die relevante Anwendungssoftware sowie die Programmierer, die diese entwickeln.*

Definition 1-1 hebt neben dem zentralen Aspekt der Datenverwaltung die Bedeutung der (Geschäfts-) Aktivitäten und der ausführenden Personen (wir sprechen im Folgenden von ***Benutzern*** bzw. ***Entwicklern***) sowie der damit verbundenen Interaktionen hervor. Ein Informationssystem wird insbesondere als eine interaktive Applikation betrachtet, wir sprechen daher von ***interaktiven Informationssystemen***.

Voraussetzung für jede Art von Generierung ist eine Spezifikationssprache. Bei der Spezifikation interaktiver Systeme ist zwischen technischen und fachlichen Anforderungen

zu unterscheiden [JBS97]. Zu den technischen Anforderungen gehört bei interaktiven Systemen beispielsweise eine echtzeitfähige Prozessorganisation [Den91], während etwa die Beschreibung von Anwendungsfällen [OMG01] den fachlichen Anforderungen zugeordnet wird. In dieser Arbeit betrachten wir vorrangig die Spezifikation und Generierung interaktiver Informationssysteme aus ihren fachlichen Anforderungen. Diese können, wie in den folgenden Teilabschnitten gezeigt wird, in den Daten-, Aufgaben- und Benutzeraspekt aufgeteilt werden.

### 1.3.1 Datenaspekt

Zentraler Aspekt eines Informationssystems ist die Datenverwaltung. Wir betrachten Informationssysteme, bei denen die verwalteten Daten strukturiert sind und daher eine Instanz (Datenstruktur) eines Datentyps (z.B. ein Datenbankschema) darstellen. Da sich die Daten im Laufe der Systemausführung im Allgemeinen ändern, ist auch eine geeignete Datenmanipulationssprache [Vos00] notwendig. Ein Datentyp bestimmt nicht nur die Struktur der Daten sondern auch in weiten Teilen die Struktur der zugehörigen Benutzungsoberflächen, die eine Bearbeitung der Daten ermöglichen.

Bei der Entwicklung der Benutzungsoberflächen ist es im Hinblick auf ihre Benutzbarkeit (engl. usability [CL99]) notwendig, Normen bzw. Richtlinien zu beachten [ISO94]. Liegen diese in formalisierter Form vor, ([Sch97] spricht dabei von *Layout-* und *Dialog-Richtlinien* bzw. *Layout-* und *Dialogmodellen*), so können Benutzungsoberflächen zum Bearbeiten von Datenstrukturen sofort aus dem zugehörigen Datentyp generiert werden [Sch97, BK99]. Die so generierten Benutzungsoberflächen können insbesondere auch als Prototyp verwendet und während der Analysephase mit den Anwendern besprochen werden. Dabei sollte es möglich sein, einen Prototyp entsprechend den Wünschen der Anwender anwendungsspezifisch zu verfeinern. Konkret könnte dies z.B. bedeuten, die Anordnung der Eingabefelder zu verändern.

Techniken und Werkzeuge zur Beschreibung allgemeiner und anwendungsspezifischer Layout- und Dialogmodelle wurden bereits in FUSE ([LS96], siehe auch Abschnitt 7.1) behandelt und sind daher nicht vorrangiges Thema dieser Arbeit. Die entsprechenden Konzepte werden daher als gegeben vorausgesetzt und in das Generierungskonzept dieser Arbeit an den entsprechenden Stellen integriert.

Im Hinblick auf die Benutzbarkeit ist es sinnvoll, die generierten Benutzungsoberflächen aus Standardinteraktionsobjekten wie Buttons oder Listen aufzubauen. Damit erhält der Benutzer die Möglichkeit, schon vorhandenes Wissen über den Umgang mit Standardinteraktionsobjekten einzusetzen. Wir nennen Benutzungsoberflächen aus Standardinteraktionsobjekten nach [Shn98] ***Formulare***. Je nach Anwendungsbereich kann es jedoch notwendig sein, speziellere Benutzungsoberflächen oder komplette andere interaktive Applikationen wie z.B. Bildverarbeitungs- oder CAD-Programme einzubetten.

---

**Anforderung 1-1 (Datenmodell)**

*Es muss möglich sein, die Typen der vom Informationssystem zu verwaltenden Daten in einem **Datenmodell** zu spezifizieren. Eine entsprechende Datenmodellierungssprache ist daher zu implementieren. Aus einem Datenmodell sollen bereits ablauffähige Benutzungsoberflächen generiert werden, die eine Manipulation entsprechender Datenstruktu-*

*ren ermöglichen. Durch ein optionales Layoutmodell soll die Möglichkeit zur Verfeinerung der Benutzungsoberflächen bestehen. Außerdem sollen speziellere Benutzungsoberflächen eingebettet werden können.*

---

### 1.3.2 Aufgabenaspekt

In Anlehnung an die bei CADUI übliche Terminologie sprechen wir im Folgenden in Abweichung von Definition 1-1 von ***Aufgaben*** anstelle von Aktivitäten, berücksichtigen aber den allgemeinen Sachverhalt, dass möglicherweise mehrere Benutzer an einer Aufgabe beteiligt sein können. Die Aufgabenbearbeitung kann sehr unterschiedlich sein. Aufgaben können durch einen einzelnen Buttonklick bearbeitet werden (z.B. wenn eine Überprüfung einer Kontobewilligung von einem Verantwortlichen bestätigt werden muss), kann aber auch über einen längeren Zeitraum stattfinden (z.B. wenn das Informationssystem dazu dient, ein Praktikum abzuwickeln, was der Koordination der Interaktionen des Leiters und der Teilnehmer während eines ganzen Semesters bedarf). Aufgaben können kausal voneinander abhängen und einen Bezug zum Datenmodell aufweisen.

Die Definition der kausalen Abhängigkeit von Aufgaben ist Thema einer ganzen Reihe von Arbeiten. Diagrammsprachliche Formalismen, wie z.B. Statecharts [Har87] oder (gefärbte) Petrinetze [Obe96], wurden nicht nur im akademischen Umfeld des Bereichs Workflow-Management konzipiert, sondern werden auch in kommerziellen Werkzeugen wie FlowMark oder Aris [JBS97] eingesetzt.

Die verschiedenen diagrammsprachlichen Konzepte haben je nach Einsatzbereich ihre Stärken und Schwächen und es ist auch nicht Ziel dieser Arbeit eine weitere oder verbesserte Notation einzuführen. Vielmehr soll für die jeweils gewünschte Diagrammsprache die entsprechende Benutzungsoberfläche automatisch aus der Beschreibung der abstrakten Syntax und dem diagrammsprachlichen Layout der jeweiligen Sprache generiert werden. Wir werden im folgenden Benutzungsoberflächen für diagrammsprachliche Notationen als ***(graphische) Grapheditoren*** bezeichnen.

Die Forderung nach Grapheditoren tritt nicht nur bei Aufgabenbeschreibungssprachen auf, sondern auch in anderen Bereichen der computergestützten Software-Entwicklung (CASE) oder beispielsweise bei Telekommunikation-Management-Werkzeugen. CASE-Werkzeuge weisen häufig sogar eine ganze Reihe verschiedener graphischer Grapheditoren auf, um Software-Entwicklungsdokumente in verschiedenen Entwicklungsphasen zu verarbeiten und zu dokumentieren (z.B. die auch in UML [OMG01] verwendeten Klassen- oder Zustandsübergangsdiagramme).

---

**Anforderung 1-2 (Generierung graphischer Grapheditoren)**

*Das Datenmodell soll mächtig genug sein, um die abstrakte Syntax gängiger Aufgabenbeschreibungssprachen wie Petrinetze oder Statecharts zu erfassen und daraus graphische Grapheditoren zu generieren. Ferner muss es möglich sein, das diagrammsprachliche Layout zu beschreiben und bei der Generierung zu berücksichtigen.*

---

Zur Implementierung eines Generierungskonzepts, bei dem Daten-, Benutzer- und insbesondere auch das Aufgabenmodell berücksichtigt werden, muss jedoch auch eine konkrete Aufgabenbeschreibungssprache entwickelt werden.

Eine formal festgelegte Semantik einer Aufgabenbeschreibungssprache kann einerseits die Grundlage für die Verifikation einer Aufgabenbeschreibung bilden [WWK97] und andererseits auch die Möglichkeit bieten, den Benutzern bei der Aufgabenbearbeitung zu helfen [Bra95, LS96].

---

**Anforderung 1-3 (Aufgabenmodell)**

*Die vom Informationssystem zu verwaltenden Aufgaben müssen in einem Aufgabenmodell spezifizierbar sein. Bei der Entwicklung einer entsprechenden Aufgabenbeschreibungssprache ist die unterschiedliche Komplexität von Aufgaben, mögliche kausale Abhängigkeiten sowie der Bezug zum Datenmodell zu berücksichtigen. Eine diagrammsprachliche Aufgabenbeschreibungsnotation ist zu entwickeln. Das Aufgabenmodell soll eine formal festgelegte Ausführungssemantik besitzen. Zusammen mit dem Datenmodell soll die Generierung ablauffähiger Benutzungsoberflächen möglich sein.*

---

### 1.3.3 Benutzeraspekt

Die verschiedenen Benutzer, die mit dem Informationssystem arbeiten, haben in der Regel unterschiedliche Sichtweisen auf Daten und Aufgaben des Informationssystems. Dies ergibt sich einerseits aufgrund unterschiedlicher Fähigkeiten und Interessen (z.B. Online-Bankkunden mit und ohne Aktienhandel), andererseits aufgrund verschiedener Verantwortlichkeiten (z.B. Projektleiter und Projektmitarbeiter). Bei Datenbank-Management-Systemen (DBMS) [Vos00] stehen dem Entwickler zur Modellierung solcher Sachverhalte geeignete Sprachelemente zur Verfügung. Dort gibt es die Möglichkeiten, Sichten (*Views*) auf Tabellen zu definieren und den Benutzern (bzw. Benutzergruppen) unterschiedliche Schreib- und Lesezugriffe auf Teile der Datenbank zuzuordnen. Um die Verwaltung der Zugriffe möglichst flexibel zu gestalten, wird dabei ein Rollenkonzept verwendet. Eine Rolle beschreibt einen bestimmten Umfang von Schreib- und Lesezugriffen auf Teile der Datenbank. Einem Benutzer der Datenbank können eine oder mehrere Rollen zugewiesen werden. Entsprechend erweitert sich seine Sichtweise auf die Datenbank.

Ähnlich flexible Möglichkeiten sind bei der Spezifikation bzw. Verwaltung von Benutzungsoberflächen für ein Informationssystem wünschenswert. Es soll möglich sein, die Sichtweisen der Benutzer auf die Daten und Aufgaben des Informationssystems, etwa mithilfe eines Rollenkonzepts, flexibel zu gestalten. Aus den Rollen, die ein Benutzer zu einem bestimmten Zeitpunkt einnimmt, ergibt sich seine Benutzungsoberfläche zum Informationssystem. Der Benutzer soll also auf genau die Daten und Aufgaben zugreifen können, die seinen jeweils eingenommenen Rollen entsprechen. Diese Anforderung führt einerseits zu einer wesentlich besseren Übersicht der Benutzer, da keine unnötigen Daten angezeigt werden, und kann andererseits auch eine bessere Performanz des gesamten Systems ermöglichen, da keine unnötigen Daten transportiert werden müssen.

Die Realisierung dieser Anforderung ist mit herkömmlichen, d.h. nicht modellbasierten UI-Entwicklungswerkzeugen, kaum zu realisieren, weil diese i.d.R. von einem statischen

Oberflächenlayout ausgehen, während sich die Eigenschaften und Rollen der Benutzer und damit auch ihre Benutzungsoberflächen zur Laufzeit ändern können.

---

**Anforderung 1-4 (Benutzermodell)**

*Die Benutzer sollen über genau auf sie abgestimmte Benutzungsoberflächen auf die Daten und Aufgaben des Informationssystems zugreifen können. Dieser Zugriff ist in einem Benutzermodell zu spezifizieren. Bei der Entwicklung der Benutzermodellierungssprache sollen die Sichtweisen (Rechte) der Benutzer flexibel modellierbar sein, etwa so, wie beim Rollenkonzept relationaler Datenbanken. Abhängig von den Sichtweisen soll der Zugriff auf die Daten und Aufgaben des Informationssystems definiert werden. Dabei soll es möglich sein, die Sichtweisen der Benutzer zur Laufzeit des Systems zu verändern, was eine entsprechende Veränderung der Benutzungsoberflächen zur Folge hat. Zusammen mit dem Daten- und Aufgabenmodell soll die Generierung ablauffähiger Benutzungsoberflächen für die einzelnen Benutzer möglich sein.*

---

## 1.4 Generierungskonzept im Überblick

Im Generierungskonzept dieser Arbeit gilt es, die in Abschnitt 1.3 genannten Anforderungen konkret umzusetzen. Das Generierungskonzept wurde ***EMU*** (**E**xtensible Models for **M**ultiple-**U**ser Interfaces) und das entwickelte CADUI-Werkzeug ***EMUGEN*** genannt.

Aus der Diskussion im vorherigen Abschnitt ergibt sich die Verwendung folgender Modelle bzw. Teile der Eingabespezifikation:

- Datenmodell
- Aufgabenmodell
- Benutzermodell
- Layoutmodell

Heutzutage existiert kaum ein System ohne eine Kommunikation mit externen Systemen. Daher ist ferner eine Schnittstelle zu externen Systemen vorgesehen.

Abbildung 1-2 zeigt das Generierungskonzept im Überblick. Die Eingabe besteht aus den oben genannten Modellen. Der Generator EMUGEN erzeugt aus den Eingabemodellen automatisch ein interaktives System (***EMU-System***), das aus dem Mehrbenutzer-Applikationsschnittstellenkern ***MAK*** und einer Menge von Benutzungsoberflächen besteht.

Zur Übersichtlichkeit wurde EMUGEN in Abbildung 1-2 als Black-Box, d.h. ohne Phasen und Teilkomponenten, dargestellt. Wenn wir – wie in Abbildung 1-2 – hervorheben wollen, dass es sich um das jeweils konkrete Modell dieser Arbeit handelt (z.B. beim Vergleich mit verwandten CADUI-Werkzeugen), so stellen wir den Präfix „EMU-" voran (z.B. *EMU-Datenmodell*).

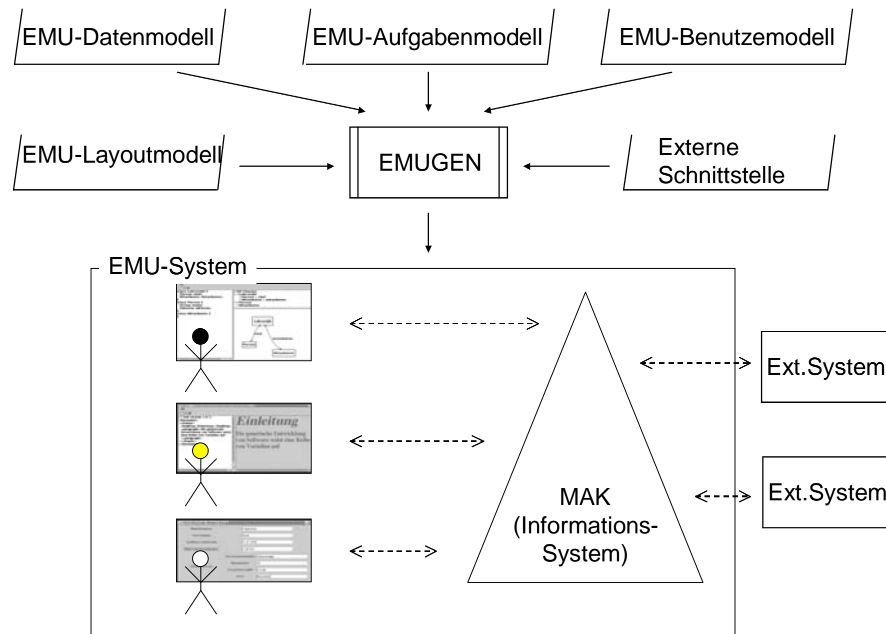


**Abbildung 1-2: EMU-Generierungskonzept**

Für jeden Benutzer verwaltet ein EMU-System eine zugehörige Benutzungsoberfläche. Diese kann aus Formularen (Masken), Grapheditoren oder vorgegebenen Basisinteraktionsobjekten (z.B. Buttons) bestehen und bietet den Benutzern die Möglichkeit, gemäß ihrer aktuellen Rechte (Rollen) den MAK zu lesen und zu bearbeiten.

Der MAK – man kann ihn auch als das eigentliche, interaktive Informationssystem betrachten – wird im Wesentlichen durch eine komplexe, hierarchisch organisierte Datenstruktur gebildet, auf der Aktionen ausführbar sind. Die hierarchisch organisierte Datenverwaltung erleichtert die automatische Konstruktion und die Aktualisierung der Benutzungsoberflächen der einzelnen Benutzer. Infolge von Interaktionen der Benutzer oder externer Systeme ändert sich der MAK. Dies kann zu einer Aktualisierung des Inhalts und des Neuaufbaus der Benutzungsoberflächen der einzelnen Benutzer führen. Letzteres ist beispielsweise dann der Fall, wenn bei einer gemeinsamen Aufgabenbearbeitung im neuen Zustand ein Benutzer bestimmte Daten einzugeben hat. In diesem Fall wird seine Benutzungsoberfläche um die dazu notwendigen Interaktionsobjekte ergänzt.

Das generierte System kann eigenständig oder als Schnittstelle zu einem externen System, wie beispielsweise einer objektorientierten oder relationalen Datenbank verwendet werden. Im ersten Fall stellt der MAK ein interaktives Informationssystem dar und kann als komplexe Daten- und Aufgabenverwaltungskomponente betrachtet werden.

In Abweichung von Abbildung 1-1 und den meisten anderen CADUI-Werkzeugen legt das EMU-Generierungskonzept keine Entwicklungsphasen fest. Es wird zwar eine iterative Systementwicklung unterstützt, bei der aus noch unvollständigen Teilmodellen bereits ablauffähige Systeme generiert werden können, jedoch kann der Entwickler auch nach anderen Methoden vorgehen.

Zur Laufzeit interagiert eine dynamische Menge von Benutzern anhand der Benutzungsoberflächen mit dem MAK. Sie bearbeiten Daten und erledigen Aufgaben, was aus abstrakter Sicht den Zustand bzw. die Struktur des Informationssystems ändert und gegebenenfalls zu einer Benachrichtigung der anderen Benutzer führt. Auf diese Weise kann auch die direkte asynchrone oder synchrone Kommunikation [Sch99] zweier oder mehrerer Benutzer ermöglicht werden, weil auf abstrakter Ebene eine Kommunikation eine Datenbearbeitung darstellt, wenn das entsprechende Kommunikationsmedium als Datenstruktur betrachtet wird.

Die folgende Reihe von Beispielinteraktionen soll die Dynamik eines EMU-Systems zur Laufzeit illustrieren:

- Benutzer können die Werte und die Struktur der vom Informationssystem verwalteten Daten verändern (vgl. Struktureditor [RT88, Sch97]).
- Benutzer können sich an- und abmelden.
- Die Menge der Benutzer kann sich ändern.
- Die Eigenschaften von Benutzern können sich ändern, was gegebenenfalls zu einer Veränderung der betroffenen Benutzungsoberflächen führen kann.
- Aufgaben werden durch die Benutzer bearbeitet und ändern dementsprechend ihren Zustand, was evtl. zur Benachrichtigung von weiteren Benutzern führt.

## 1.5 Aufgabenstellung der Arbeit

Die Aufgabe dieser Arbeit besteht darin, das in Abschnitt 1.4 im Überblick gezeigte Generierungskonzept auszuarbeiten und – darauf basierend – das Werkzeug EMUGEN prototypisch zu implementieren. Dabei sind die unter Abschnitt 1.3 dargestellten Anforderungen zu berücksichtigen. Die Aufgabenstellung beinhaltet daher die folgenden Teilaufgaben:

- Konzeption und Implementierung von Modellierungssprachen für das Daten-, Benutzer- und Aufgabenmodell (Eingabespezifikation) mit Berücksichtigung der Schnittstelle zu externen Systemen
- Konzeption und Implementierung eines Generators (batch-orientiertes CADUI-Werkzeug), der gemäß Abschnitt 1.3 aus geeigneten Eingabemodellen einen ablauffähigen Prototyp (EMU-System) generiert
- Darstellung der Arbeitsweise und Beleg der Anwendbarkeit des erstellten Generators durch Modellierung einer Reihe von Beispielen
- Formale Definition der Ausführung generierter EMU-Systeme

## 1.6 Ergebnisse

Zur Entwicklung praktisch einsetzbarer CADUI-Werkzeuge liefern die Modellierungs- und Implementierungskonzepte dieser Arbeit eine wichtige Grundlage, weil ihre Mächtigkeit über die bisherigen Werkzeugprototypen aus dem Bereich CADUI hinausgeht.

Dies betrifft vor allem die folgenden Aspekte:

- Modellierung mehrerer Benutzer und ihrer Eigenschaften
- Generierung von Grapheditoren
- Integration in ablauffähige Systeme

Der implementierte Generator EMUGEN kann auch zur Generierung ablauffähiger horizontaler Prototypen [Den91] interaktiver Informationssysteme aus einer formalen Anforderungsbeschreibung eingesetzt werden. Diese aus Benutzersicht funktional vollwertigen Prototypen eignen sich als Hilfsmittel zur Anforderungskonkretisierung bei der partizipativen Systementwicklung. Dies schließt sowohl die Entwicklung von Mehrbenutzersystemen als auch die Entwicklung von Grapheditoren – bzw. damit in Zusammenhang stehenden Diagrammsprachen – und deren Integration mit ein.

Das mit EMUGEN generierte Beispiel *Ad-hoc Workflows* [Aal98] zeigt, dass es möglich ist, diese beiden Einsatzgebiete (Mehrbenutzersystem, Grapheditoren) miteinander zu verbinden. In diesem Beispiel wird ein Aufgabenverwaltungssystem für mehrere Benutzer generiert, bei dem die Aufgaben zur Laufzeit graphisch (d.h. mit einem generierten Grapheditor) modelliert und von den jeweils verantwortlichen Benutzern ausgeführt werden. Die Kompaktheit der EMUGEN-Eingabe zeigt sich darin, dass diese Beispielanwendung in weniger als 200 Zeilen spezifizierbar ist.

Aus Sicht des Autors kann EMUGEN damit auch als mögliche Implementierung der von [Den91] beschriebenen sogenannten „idealen Maschine“ betrachtet werden. Diese dient der (gedachten) Ausführung einer Systemspezifikation, die aus einem Datenmodell, einem Funktionenmodell und einem Modell einer Dialog-Benutzerschnittstelle besteht, und idealisiert – ähnlich wie die vorliegende Arbeit – den technisch notwendigen Ausführungsapparat.

Die Implementierung von EMUGEN (genauer: jedes generierte EMU-System, vgl. Abbildung 1-2) wendet Lösungskonzepte aus dem Übersetzerbau an, wie etwa die inkrementelle Attributauswertung [RT88] zur effizienten (dynamischen) Neuberechnung der Benutzungsoberflächen nach einer Interaktion bzw. Veränderung des MAKs. Der damit in Zusammenhang stehende, komplexe Formalismus ist jedoch für den Entwickler transparent. D.h. es ist keine formale Schulung notwendig, um mit EMUGEN zu arbeiten.

## 1.7 Inhalt und Lesehinweise

### 1.7.1 Inhalt

In Kapitel 2 wird beschrieben, was in dieser Arbeit unter dem Begriff *Interaktives Informationssystem* verstanden wird und welche Aspekte damit in Verbindung stehen. Darauf aufbauend erfolgt in Kapitel 3 die Betrachtung der grundlegenden Eigenschaften der EMU-Systeme (vgl. Abbildung 1-2), also der in dieser Arbeit generierten interaktiven Informationssysteme, und des EMU-Generierungskonzepts.

Kapitel 4 führt das EMU-Datenmodell ein und zeigt die Ausführung und die formale Semantik von EMU-Systemen, die ausschließlich aus dem EMU-Datenmodell generiert werden. Kapitel 5 führt das Aufgaben- und Benutzermodell von EMU ein und zeigt die

Ausführung und die formale Semantik von EMU-Systemen, die aus allen drei Modellen generiert wurden. Die schrittweise Einführung der Eingabemodelle in zwei getrennten Kapiteln soll der besseren Verständlichkeit dienen.

Kapitel 6 beschreibt die Vorgehensweise zur generativen Entwicklung von interaktiven Informationssystemen mit EMUGEN und zeigt durch eine Reihe von Beispielen die Anwendbarkeit des Generierungskonzepts der vorliegenden Arbeit. In Kapitel 7 erfolgt ein Vergleich von EMUGEN mit anderen CADUI-Werkzeugen. Kapitel 8 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick ab. Der Anhang beschreibt die konkrete und abstrakte Syntax der Eingabemodelle.

### 1.7.2 Lesehinweise

Um sich lediglich einen schnellen Überblick über die Möglichkeiten des Generierungskonzepts der vorliegenden Arbeit zu verschaffen, sollte es für den Leser mit Vorkenntnissen im Bereich *Entwicklung und Spezifikation interaktiver Systeme* möglich sein, nach dem Einführungskapitel Abschnitt 3.4 (Entwicklungsszenarien im Überblick) und anschließend Kapitel 6 mit den Beispielen zu lesen und nur bei Bedarf die Sprachelemente der Eingabemodelle in den Kapiteln 4 und 5 nachzuschlagen.

Kapitel 2 stellt eine Motivation für die im Einleitungskapitel bereits ausgearbeiteten Anforderungen und einiger Lösungskonzepte (z.B. Grapheditoren, aufgabenbezogene dynamische Formulare) dar. Ferner erfolgt dort eine Einführung in die Begriffswelt der *Spezifikation interaktiver Systeme* (z.B. Seeheim-Modell, Werkzeuge). Daher ist das Lesen von Kapitel 2 nur notwendig, wenn man beispielsweise wissen möchte, warum in EMUGEN ausgerechnet Formulare und Grapheditoren generiert und eingesetzt werden oder wenn eine Kurzeinführung zu interaktiven Systemen notwendig ist.

In Kapitel 3 wurde versucht, die grundlegenden Ideen des Generierungskonzepts, die sich nur schwer mit formalen Mitteln oder durch Beispiele darstellen lassen, zu vermitteln. Um dieses Kapitel vollständig zu verstehen, könnte es notwendig sein, es nach dem Durchlesen der nachfolgenden Kapiteln zu wiederholen.

Die jeweils ersten beiden Abschnitte der Kapitel 4 und 5 stellen eine informelle Einführung in die EMU-Eingabemodelle dar. Hilfreich, könnte es sein, diese Abschnitte gemeinsam mit den Beispielen von Kapitel 6 zu lesen.

Die formalen Abschnitte 4.3 und 5.3 stellen jeweils eine exakte Beschreibung der Generierungsfunktionalität dar, beschreiben aber auch auf abstrakte und damit sehr kompakte Weise die Anforderungen an die Implementierung. Ihr Studium ist jedoch für das intuitive Verständnis der Generierungsfunktionalität nicht unbedingt erforderlich. Insbesondere sollte es für einen Benutzer von EMUGEN, d.h. für einen Entwickler, der mit EMUGEN arbeiten möchte, nicht notwendig sein, die formalen Abschnitte zu studieren.

Um eine möglichst klare Ausdrucksweise zu erhalten, wird in der Arbeit durchgängig die männliche Form verwendet, obwohl damit natürlich stets auch die weibliche Form gemeint ist.

# 2 Interaktive Informationssysteme

Im Einleitungskapitel wurde bereits die Spezifikation interaktiver Informationssysteme mittels verschiedener Modelle motiviert (vgl. Abschnitt 1.3, Abbildung 1-2). Neben der genauen Betrachtung der entsprechenden Modellierungssprachen, die in nachfolgenden Kapiteln erfolgt, ist jedoch auch zu klären, welche Aspekte die vorliegende Arbeit mit einem *interaktiven Informationssystem* verbindet.

Daher werden in diesem Kapitel die Benutzersicht, die Architekturen und die Entwicklung interaktiver Informationssysteme in jeweils einzelnen Abschnitten betrachtet.

Die Aufgabenstellung stellt vorrangig *fachliche* (im Gegensatz zu *technischen* [JBS97]) Anforderungen an das Generierungskonzept dieser Arbeit. Daher verzichten wir hier auf die vielfältigen technischen Details, die etwa aufgrund der Verteiltheit und Heterogenität solcher Systeme in der Praxis zu berücksichtigen sind und konzentrieren uns vielmehr auf die abstrakte Funktionalität, die darin besteht, mehrere Benutzer bei der Daten- und Aufgabenverwaltung zu unterstützen.

## 2.1 Benutzung interaktiver Informationssysteme

Die Benutzung interaktiver Informationssysteme wird im Forschungsgebiet *Mensch-Maschine-Interaktion* [HCI01] behandelt. Dort werden fächerübergreifend aus Informatiker- (etwa [CL99, Shn98, Sta96]) und Psychologensicht (etwa [CMN83, Joh92]) Regeln, Methoden und Modelle diskutiert, welche die Interaktion zwischen Mensch und Benutzungsoberfläche verbessern sollen. Ergebnisse dieser Untersuchungen finden sich in Normen wie der ISO 9241 [ISO93] wieder. Die Teile ISO 9241-17 und ISO 9241-16 beschreiben explizit die Interaktionsformen bzw. die Benutzungsoberflächentypen *Formular* und *direkte Manipulation*.

In den folgenden Teilabschnitten wird klar, warum gerade diese Interaktionsformen für die Benutzung interaktiver Informationssysteme vorrangig von Bedeutung sind. Daneben gibt es natürlich eine Vielzahl weiterer denkbarer Interaktionsformen, wie beispielsweise Kommandosprache, Menüs oder natürliche Sprache, die möglicherweise an

zusätzliche physikalische Interaktionsmedien verknüpft sind [Shn98, Sta96]. Eine alternative Interaktionsform kann für Informationssysteme in unserem Sinne verwendet werden, wenn sie Möglichkeiten zur

- Navigation,
- Manipulation strukturierter Daten und
- Interaktionsausführung

bietet.

Die Interaktionsform *direkte Manipulation* betrachten wir in dieser Arbeit unter dem Aspekt, dass sie uns ermöglicht, Graphen (d.h. aus Knoten und Kanten bestehende, am Bildschirm sichtbare Strukturen, nicht etwa Kurvenverläufe in einem Koordinatensystem) zu bearbeiten. Um diese speziellere Anwendung zu betonen, sprechen wir im Folgenden vom Benutzungsoberflächentyp ***Grapheditor*** anstelle von der Interaktionsform *direkte Manipulation*.

### 2.1.1 Formulare

Formulare (engl. *form-fillin* [Shn98], *Maske* in [Sta96, Den91]) werden häufig bei Informationssystemen zum Bearbeiten strukturierter Daten oder Datenbanktabellen verwendet. [Sta96] nennt als Grund dafür die Möglichkeit der Überprüfung von Plausibilitäten, die durch Integritätsbedingungen spezifiziert werden können. Der Zusammenhang zwischen Formularen und Datenstrukturen ist sehr eng, weil ein Formular selbst als eine rekursive Datenstruktur – bestehend aus Teilformularen – betrachtet werden kann.

In vielen Bereichen ist die Verwendung ***dynamischer Formulare*** sinnvoll [FS98, BK99]. Diese Formulare ändern aufgrund der bisherigen Eingabe des Benutzers ihre Form und zeigen nur diejenigen Formularteile an, die zur Vervollständigung der Eingabe notwendig sind. Beispielsweise ist es bei einem Steuerausgleich nicht notwendig, die Formularteile zur Eingabe von Einkünften aus einer nebenberuflichen Tätigkeit anzuzeigen, wenn der Benutzer eine entsprechende Frage („Führen Sie eine nebenberufliche Tätigkeit aus?") bereits verneinte.

Ferner ist es mit dynamischen Formularen auch möglich, eine Navigationsmöglichkeit zu schaffen, was beispielsweise notwendig ist, wenn sich der Benutzer in einer sehr komplexen Datenstruktur „bewegt". Der Benutzer wählt in diesem Fall selbst den aktuell anzuzeigenden Bereich aus.

Formulare dienen jedoch nicht nur der Manipulation und Darstellung strukturierter Daten. Wenn es sich bei den Daten um Objekte im objektorientierten Sinne handelt, so ist es naheliegend, ihnen Interaktionen bzw. Methoden zuzuordnen, die durch den Benutzer über das Formular unmittelbar auszuführen sind. Man denke beispielsweise an ein einfaches Adressbuch, in welchem eine Liste von Personen geführt wird. Wird eine Person selektiert, so erscheint ein Formular mit den Daten der Person und mit Interaktionsmöglichkeiten, um die Person anzurufen, oder ihr eine Email zu schreiben (Abbildung 2-1). Bei beiden Interaktionen fungiert die Person als implizites Argument im objektorientierten Sinne, jedoch auf Ebene der Benutzerinteraktion. D.h. der Benutzer muss zur Aktionsausführung nicht jedes Mal eine Telefonnummer bzw. Emailadresse eingeben, wenn diese zu der entsprechenden Person bereits angegeben wurde. Ferner kann als

Vorbedingung der Aktionsausführung ein Prädikat zum Zustand der Person angegeben werden. In Abbildung 2-1 kann beispielsweise die Aktion *anrufen* bei der aktuell selektierten Person nicht ausgeführt werden, weil die dazu notwendige Telefonnummer noch nicht angegeben wurde.

Formulare sind auch bei der Anmeldung bzw. Authentifizierung in Mehrbenutzersystemen notwendig.

Der Aufbau eines Formulars ermöglicht es auf sehr natürliche Weise, bereits bestehende ***Interaktionsobjekte*** (Oberbegriff für Bildschirmelemente wie Buttons, Eingabefelder, Fenster usw. [Lar92]) einzubetten, welche die Darstellung und Manipulation primitiver oder komplexer Basisdatenstrukturen wie Zeichenreihen bzw. Graphen ermöglicht.

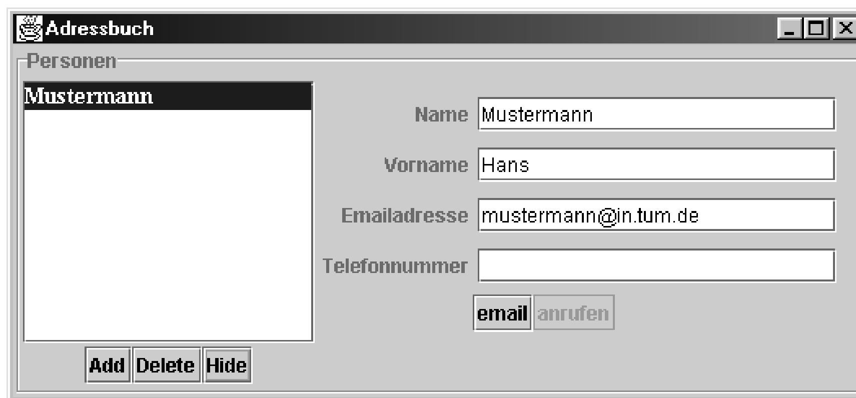


**Abbildung 2-1: Interaktionen mit impliziten Argumente bei Formularen**

### 2.1.2 Grapheditoren

***Grapheditoren*** ([Bal00, Gil98] sprechen von ***diagrammsprachlichen Editoren***) erlauben ähnlich wie Formulare die Bearbeitung strukturierter Daten. Sie stellen die strukturierten Daten jedoch als Graph durch eine Menge von Knoten und (i.d.R. gerichtete) Kanten dar. Die zugrundeliegende Interaktionsform *direkte Manipulation* [Shn98, Sta96] bedeutet, dass es auf intuitive Weise möglich ist, Elemente des Graphen zu bearbeiten, beispielsweise Knoten zu verschieben oder zu löschen („drag & drop“).

Manche dieser Interaktionen (etwa das Löschen von Knoten) haben unmittelbaren Einfluss auf die intern im Programm verarbeitete Datenstruktur (vgl. die in Abschnitt 2.2.2 besprochene MVC-Architektur), während andere nur die Veränderung der optischen Erscheinung (z.B. Koordinaten) am Bildschirm zur Folge haben.

Ähnlich wie ein Formular, kann auch ein Grapheditor als rekursive Datenstruktur betrachtet werden, wenn die Knoten und Kanten (bzw. die ihnen zugeordneten Datenstrukturen) mit weiteren Grapheditoren bearbeitet werden können.

Bei der Benutzung eines Grapheditors denkt der Benutzer meist vorrangig an die Semantik des visualisierten und manipulierbaren Graphen und weniger daran, eine zugrundeliegende Datenstruktur zu bearbeiten. Beispielsweise dient der Grapheditor in Abbildung 2-8 der Definition eines Phasenmodells für die Entwicklung interaktiver Systeme und damit der Aufgabendefinition. Der Benutzer dieses Grapheditors verbindet mit den graphisch dargestellten Knoten Entwicklungsphasen bzw. zu verrichtende Teilaufgaben mit ihren kausalen Abhängigkeiten und die dabei zu erstellenden Dokumente.

Neben der Aufgabendefinition werden Grapheditoren zur Aufgabenbearbeitungszeit als Analysewerkzeug verwendet. Dabei wird der Bearbeitungszustand, etwa durch farbige Kennzeichnung der momentan aktiven Teilaufgaben, visualisiert. Dieser Aspekt wird in Abschnitt 2.1.4 genauer erläutert.

Grapheditoren und Formulare können miteinander gekoppelt werden. Beispielsweise kann ein Teil eines Formulars durch einen Grapheditor ausgefüllt sein oder Details der Datenstruktur, die durch einen graphischen Knoten im Grapheditor visualisiert wird, können durch ein Formular dargestellt werden.

### 2.1.3 Persönliches Rollen-Management

Ein zuletzt häufig diskutiertes Thema bei HCI ist die Entwicklung aufgabengerechter Benutzungsoberflächen, welche die Produktivität und Zufriedenheit der Benutzer erhöhen sollen. [Shn98] schlägt dazu ein persönliches Rollen-Management vor, wobei dem Benutzer nur die für ihn (aufgrund seiner aktuellen Aufgaben) relevanten Teile visualisiert werden, wodurch eine einfachere und damit wesentlich effizientere Benutzung ermöglicht werden soll. Im Vergleich dazu sind beim derzeit verbreiteten Desktop-Ansatz [Shn98] nur sehr wenige der am Bildschirm dargestellten Interaktionsobjekte zur Erledigung aktuell anstehender Aufgaben relevant.

Es gibt natürlich eine Reihe von Standardanwendungen (z.B. CAD-, CASE- oder Dokumentverarbeitungsprogramme), die aufgrund ihrer Komplexität viele Interaktionsobjekte benötigen und bei denen es meist nicht sinnvoll ist, diese zu reduzieren. Es geht bei diesem Ansatz vielmehr darum, die Benutzerinteraktionen, die zur Erledigung aktueller Aufgaben benötigt werden, auf das notwendige Maß zu minimieren. Typischerweise bestehen diese Interaktionen darin, eine Aufgabe anzunehmen, zu bearbeiten und abzuschließen. Wenn zur Aufgabenbearbeitung komplexe Standardanwendungen notwendig sind, so sind diese einzubinden. Handelt es sich um verhältnismäßig einfache Dialoge [Lar92], die Entscheidungen (z.B. Kontobewilligung ja/nein) oder die Eingabe von Datenstrukturen (z.B. Eingabe eines Kreditantrags) betreffen, so ist es auch während der Aufgabenbearbeitung möglich, die Menge der aktuell sichtbaren Interaktionsobjekte auf die aktuell notwendigen zu beschränken. Bei der Verwendung solch aufgabenangepasster Benutzungsoberflächen ist eine kürzere Einarbeitungszeit von neuen Aufgabenbearbeitern zu erwarten.

Dieser Vorschlag zur Benutzung interaktiver Systeme aus [Shn98] hat eine Ähnlichkeit mit der Benutzung von Datenbank- und Workflow-Management-Systemen.

### 2.1.4 Benutzung von Datenbank- und Workflow-Management-Systemen

Interaktive Informationssysteme im Sinne dieser Arbeit beinhalten aus Sicht der fachlichen Anforderungen (insbesondere bezüglich der Benutzerinteraktionen) ähnliche Möglichkeiten, wie sie von Workflow- und Datenbank-Managementsystemen (DBMS bzw. WFMS) geboten werden [WfM96, Vos00].

- Ähnlich wie mit einem WFMS kann man mit einem interaktiven Informationssystem im Sinne dieser Arbeit Workflows beschreiben und ausführen sowie die Ausführung überprüfen (*Workflow-Monitoring* [WfM96]).

- Ähnlich wie mit einem DBMS können durch ein interaktives Informationssystem Daten definiert werden, die anschließend (zur Laufzeit) bearbeitet werden können.

Daher betrachten wir in nun die Benutzung von WFMS und DBMS.

**Workflow-Managementsysteme**

Alles was mit der rechnergestützten Verwaltung von (Arbeits-)Abläufen zusammenhängt, wird im Bereich *Workflow-Management* [WfM96, JBS97] untersucht. Ein entsprechendes Werkzeug wird *Workflow-Management-System* (*WFMS*) genannt. Ein ***WFMS*** ist nach [JBS97] ein reaktives Basissoftwaresystem zur Steuerung des Arbeitsflusses (***Workflows***) zwischen beteiligten Stellen nach Vorgaben einer Ablaufspezifikation (***Workflow-Schema***). Ein WFMS unterstützt mit seinen Komponenten sowohl die Entwicklung von *Workflow-Management-Anwendungen* (*WMA*) als auch die Steuerung von Workflows. Nach [JBS97] ist eine ***WMA*** zur effektiven Ausführung eines Workflows notwendig. Daher beinhaltet sie mehr als die Ablaufspezifikation, nämlich die zugehörigen Daten, Bearbeiter, Benutzungsoberflächen und externe Applikationen.

WFMSe sind hochgradig interaktiv und deshalb ist die Bereitstellung entsprechender Benutzungsoberflächen eine wichtige fachliche Anforderung an solche Systeme [JBS97]. Die übliche Benutzung eines WFMS wird durch Abbildung 2-2 deutlich.

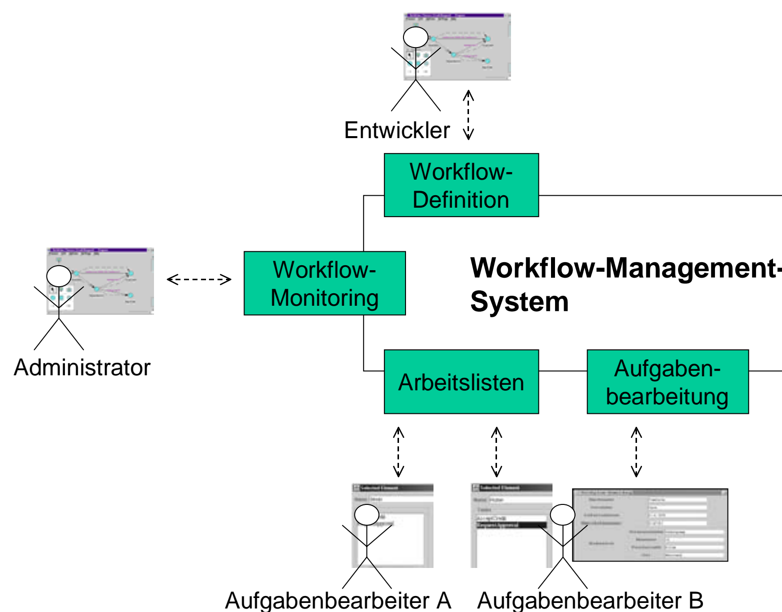


**Abbildung 2-2: Benutzung eines Workflow-Management-Systems [WfM96, JBS97]**

In Abbildung 2-2 werden die Benutzertypen *Workflow-Entwickler* und *-Administrator* sowie *Aufgabenbearbeiter* unterschieden.

Die Benutzung des WFMS durch einen Workflow-Entwickler besteht vorrangig aus der Definition von Workflows. Üblicherweise werden dazu Grapheditoren verwendet. Abbildung 2-3 zeigt ein entsprechendes Beispiel aus dem kommerziellen WFMS Flow-Mark von IBM [JBS97, Tee98]. Grapheditoren werden auch vom Workflow-Administrator zum Monitoring von Workflow-Ausführungen benutzt. Die Aufgabenbe-

arbeiter werden über sog. *Arbeitslisten* (engl. *Work-List* [WfM96]) über ihre aktuell zu bearbeitenden Aufgaben informiert und können damit auch die Aufgabenbearbeitung auswählen und starten. In Abbildung 2-2 wird angedeutet, dass Aufgabenbearbeiter B eine Aufgabe aus seiner Aufgabenliste bereits gewählt (selektiert) hat und sie mit Hilfe eines Formulars bearbeitet. Zur eigentlichen Aufgabenbearbeitung werden neben Formulare auch externe Anwendungen verwendet.

In Abschnitt 6.2.5 wird gezeigt, dass es mit EMUGEN, also dem Generierungswerkzeug dieser Arbeit, möglich ist, ein einfaches WFMS zu generieren, das die hier beschriebene Funktionalität bzgl. der Benutzerinteraktionen aufweist.

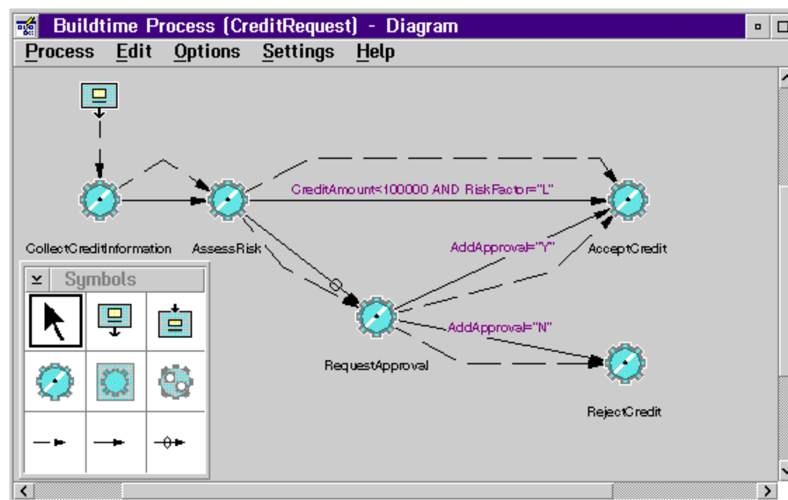


**Abbildung 2-3: Workflow-Definition und -Monitoring mit FlowMark (aus [Tee98])**

### Datenbank-Management-Systeme

Die Benutzung eines Datenbank-Management-Systems (DBMS) erfolgt abstrakt gesehen ähnlich wie die eines WFMS (Abbildung 2-4). Datenbank-Entwickler definieren in einer Datendefinitionssprache Datenbank-Schemata [Vos00] und Datenbank-Administratoren verwalten unter anderem die einzelnen Datenbanken sowie die Endanwender und ihre Zugriffsrechte. Die Endanwender lesen und bearbeiten entsprechend ihrer Zugriffsrechte den Inhalt der verwalteten Datenbanken. Sowohl die Interaktionen des Entwicklers als auch die des Administrators können aus abstrakter Sicht auch als Datenbearbeitung betrachtet werden (vgl. die Meta- und die Benutzertabellen in relationalen DBMS).

Die Benutzungsoberflächen werden bei der Benutzung eines DBMS häufig durch Formulare, textuelle Kommandosprachen wie z.B. SQL [Vos00] oder anwendungsspezifische Applikationen gebildet. Da es uns wiederum lediglich um den Aspekt der Benutzung solcher Systeme auf abstrakter Ebene geht, abstrahieren aus Übersichtsgründen Abbildung 2-2 und Abbildung 2-4 von vielfältigen Aufgaben, die insbesondere bei der Administration solcher Systeme anfallen, wie beispielsweise Performanzoptimierungen.

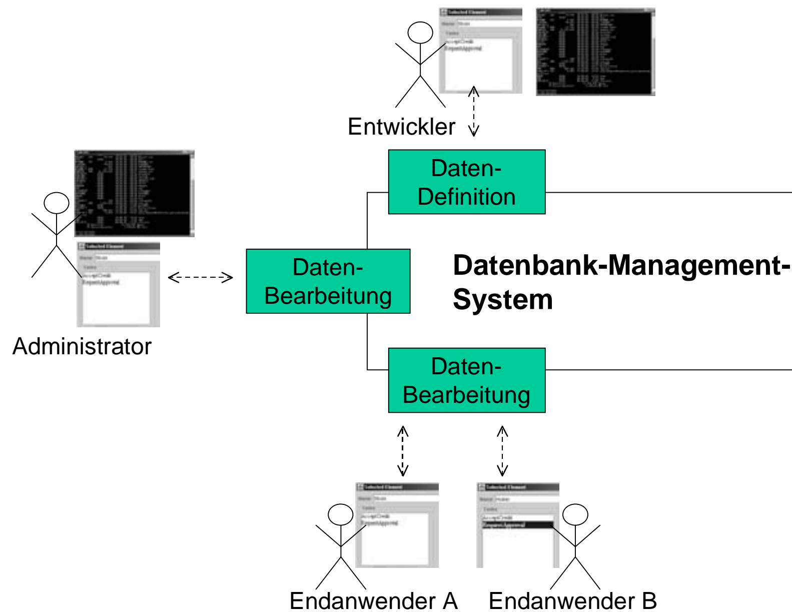


**Abbildung 2-4: Benutzung eines Datenbank-Management-Systems**

Die Verwaltung von Benutzerrechten erfolgt bei DBMS üblicherweise über ein Rollenkonzept. Dabei werden die Benutzerrechte durch bestimmte Daten festgelegt (bei relationalen Datenbanken durch den Inhalt bestimmter Tabellen), die meist vom Administrator (bzw. einem Benutzer mit Administratorrechten) zu verwalten sind. Durch die Benutzerrechte wird der Zugriff der einzelnen Benutzer auf die Daten des DBMS festgelegt.

Im Hinblick auf die Benutzung eines DBMS ist es sinnvoll, den Benutzern Benutzungsoberflächen bereitzustellen, die genau die Daten darstellen, die für die Benutzer gemäß ihrer Zugriffsrechte les- bzw. schreibbar sind (vgl. Anforderung 1-4). Ferner ist es natürlich notwendig, dem Benutzer eine Navigationsmöglichkeit zur Verfügung zu stellen, da es sich meist um eine große Menge von Daten handelt.

Die Benutzung interaktiver Informationssysteme im Sinne dieser Arbeit umfasst die in Abbildung 2-2 und Abbildung 2-4 dargestellten Interaktionsmöglichkeiten.

## 2.2 Architekturen interaktiver Informationssysteme

Wir betrachten an dieser Stelle lediglich grundlegende Architekturen, die unmittelbaren Einfluss auf die vorliegende Arbeit hatten. Die Umsetzung und Anwendung dieser Architekturen in die generierten interaktiven Informationssysteme dieser Arbeit (EMU-Systeme) wird im nachfolgenden Kapitel beschrieben.

### 2.2.1 Seeheim-Modell

Als grundlegende Architektur interaktiver Systeme wird das Seeheim-Modell [Pfa85] betrachtet. Es teilt ein interaktives System in die Komponenten *Präsentation*, *Dialogkontrolle* und *Applikation* auf. Der Benutzer sieht am Bildschirm lediglich die aus Interaktionsobjekten zusammengesetzte Präsentation (wir sprechen auch vom *Layout*) des interaktiven Systems. Durch bestimmte Interaktionsobjekte können Zustandsübergänge in

der Dialogkontrolle ausgelöst werden. Beim Zustandsübergang können Aktionen der Applikationsschnittstelle ausgeführt werden und je nach Resultat können verschiedene Folgezustände eingenommen werden. In Abbildung 2-5 wird das Seeheim-Modell dargestellt, wobei seine Komponenten in Zusammenhang mit den Begriffen dieser Arbeit gestellt werden.

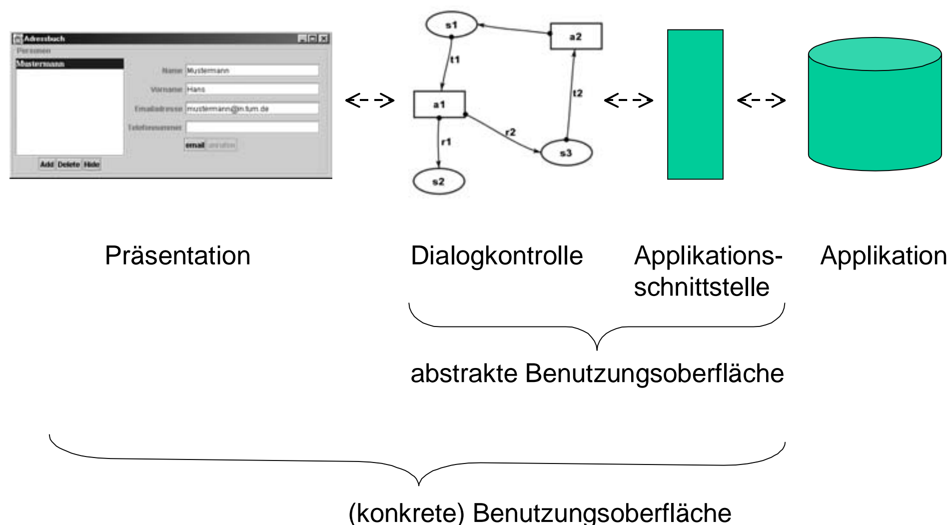


**Abbildung 2-5: Seeheim-Modell eines interaktiven Systems**

In Anlehnung an [Sch97] verwenden wir für die Dialogkontrolle und die Applikationsschnittstelle zusammen die Bezeichnung ***abstrakte Benutzungsoberfläche***. Den Begriff (***konkrete***) ***Benutzungsoberfläche*** verwenden wir, wenn zusätzlich zur abstrakten Benutzungsoberfläche auch die Präsentationskomponente mit eingeschlossen wird.

Das Seeheim-Modell zeigt lediglich eine grobe Strukturierung eines interaktiven Systems und lässt damit eine Reihe von Entwurfsentscheidungen offen, beispielsweise bzgl. der Art der Kommunikation zwischen den einzelnen Komponenten oder ihrer Aufteilung auf eine Client-Server-Architektur [Sch99].

Das Seeheim-Modell (bzw. Variationen davon) wird nicht nur als Architektur eines fertigen interaktiven Systems, sondern auch als Strukturierung der Spezifikation verwendet. Eine solche Strukturierung erfolgt beispielsweise bei den CADUI-Werkzeugen FUSE und MASTERMIND (vgl. Abschnitte 7.1, 7.4) und auch bei der in [Den91] beschriebenen Entwicklungsmethode für interaktive Informationssysteme. Das Seeheim-Modell wird also nicht nur zur Modularisierung des Quellcodes verwendet, sondern auch auf höherer Abstraktionsebene zur Strukturierung der Anforderungs- oder Entwurfsspezifikationen.

Wir betrachten im Folgenden die für diese Arbeit wichtigsten Variationen des Seeheim-Modells:

- MVC-Architektur [KP88]
- HIT-Instanzen [Sch97]

### 2.2.2 MVC-Architektur

Die Architektur ***Model-View-Controller*** (***MVC***) [KP88] wurde erstmals in der Sprache Smalltalk-80 [GR83] eingeführt. Die MVC-Komponente *Model* entspricht der Applikation (bzw. der Applikationsschnittstelle), während die MVC-Komponente *View* der Präsentation im Seeheim-Modell entspricht. Die MVC-Komponente *Controller* ermöglicht dem Benutzer die Bearbeitung des *Models*. Dabei ist es möglich, dass mehrere (meist paarweise auftretende) *View-* und *Controller*-Komponenten dasselbe *Model* oder Teile davon visualisieren bzw. bearbeiten. Sämtliche *View-* und *Controller*-Komponenten registrieren sich dazu beim *Model*. Wenn sich das *Model* ändert, so werden sämtliche *Views* davon benachrichtigt. Das Besondere an dieser Architektur ist die einfache Art der Kommunikation, die nur auf den beiden Primitiven *Registrierung* und *Benachrichtigung* basiert und damit abstrakt spezifizierbar ist. D.h. beispielsweise, dass das *Model* die konkrete Ausprägung einer *View*-Komponente nicht kennen muss, sondern lediglich weiß (technisch meist durch eine Schnittstelle vereinbart) wie sie zu benachrichtigen ist.

Diese Sichtweise – so trivial sie erscheinen mag – ist Grundlage einer Reihe von Überlegungen, die dieser Arbeit zugrunde liegen. So können die in Abschnitt 2.1 betrachteten Benutzungsoberflächentypen Formular und Grapheditor als unterschiedliche *Views* im Sinne der MVC-Architektur verstanden werden (vgl. Abbildung 1-2). Eine möglicherweise etwas weniger offensichtliche Anwendung liegt bei der Benutzung eines WFMS (vgl. Abschnitt 2.1.4) vor, wenn ein Administrator und ein Aufgabenbearbeiter dieselbe Workflow-Ausführung beobachten bzw. bearbeiten – jeweils jedoch mit unterschiedlichen Benutzungsoberflächen, also *View/Controller*-Paaren. So sieht der Workflow-Administrator eines WFMS mit dem ihm zugeordneten Grapheditor den aktuellen Zustand der Aufgabenbearbeitung etwa dadurch, dass aktuell aktive bzw. zu bearbeitende Aufgaben besonders markiert sind, während der Aufgabenbearbeiter in seiner Arbeitsliste sieht, dass er eine Aufgabe auszuführen hat und dies auch effektiv erledigen kann. Die MVC-Architektur wird dabei auf verschiedenen logischen bzw. sprachlichen Ebenen angewandt (Ebene der Aufgabendefinition, Ebene der Aufgabenbearbeitung).

In Abschnitt 6.2.5 wird gezeigt, dass diese Sichtweise sogar ermöglicht, die Aufgabendefinition und –bearbeitung miteinander zu verschränken, d.h., während der Aufgabenbearbeitung Änderungen an der Ablaufdefinition vorzunehmen (vgl. Abbildung 6-9). Beim Workflow-Management spricht man in diesem Zusammenhang von *Ad-hoc Workflows* [Aal98].

### 2.2.3 HIT-Instanzen

***HIT-Instanzen*** [Sch97] (HIT steht für *Hierarchic Interaction Templates*) sind die Laufzeitmodelle des BOSS-Systems (vgl. Abschnitt 7.1). Aus Übersetzerbausicht [WM97] ist eine HIT-Instanz ein attributierter Baum im Sinne dynamischer, attributierter Grammatiken [Gan78]. Als Architektur eines interaktiven Systems betrachtet, stellt dieser attributierte Baum zusammen mit einer Teilmenge der Attribute die Dialogkontrolle und die Applikationsschnittstelle im Sinne des Seeheim-Modells dar und wird daher als abstrakte Benutzungsoberfläche bezeichnet. Das zusätzlich zur abstrakten Benutzungsoberfläche geführte, synthetische Attribut *CurrentStateLayout* (genauer: der Wert dieses Attributs bei seinem Vorkommen an der Wurzel) stellt bei [Sch97] die zugehörige Präsentation einer HIT-Instanz dar. In Abbildung 2-6 wird der grundsätzliche, stark vereinfachte Aufbau einer HIT-Instanz visualisiert. Die Kreise in Abbildung 2-6 kennzeichnen

die Knoten des Baums, die Formularteile visualisieren die Werte des Attributes *CurrentStateLayout* an den jeweiligen Knoten.

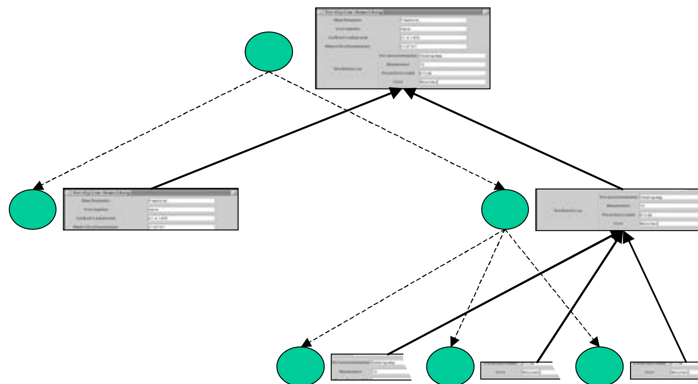

**Abbildung 2-6: Prinzipieller Aufbau einer HIT-Instanz [Sch97]**

Da die Struktur einer HIT-Instanz im Allgemeinen dynamisch ist, kann dieser Ansatz verwendet werden, um dynamische Formulare durch einen Datentyp zu modellieren und automatisch zu generieren [Sch97, FS98, BK99]. Es sei an dieser Stelle angemerkt, dass die Generierung dynamischer (also möglicherweise auch rekursiver) Formulare komplexer ist als die Generierung statischer Formulare mit Hilfe sogenannter *Maskengeneratoren*, die häufig zusammen mit kommerziellen Datenbank-Management-Systemen ausgeliefert werden (vgl. Abschnitt 2.3.2), weil bei dynamischen Formularen das Layout nicht statisch festgelegt werden kann.

Eine HIT-Instanz kann auch als spezielle Form einer MVC-Architektur betrachtet werden, weil der Wert des Attributes *CurrentStateLayout* (entspricht dem *View/Controller*-Paar) immer dann berechnet wird, wenn sich die abstrakte Benutzungsoberfläche (entspricht dem *Model*) ändert.

HIT-Instanzen weisen noch eine Reihe weiterer Aspekte und Möglichkeiten beispielsweise bezüglich der automatischen, intelligenten Anordnung von Interaktionsobjekten in der Präsentation auf, die hier aus Platzgründen nicht beschrieben werden können.

Ähnlich wie die MVC-Architektur ist das Konzept der HIT-Instanzen allgemein genug, um es in natürlicher Weise zu erweitern. Wenn man beispielsweise eine HIT-Instanz nicht nur mit einem Formular, sondern auch mit einem Grapheditor bearbeiten will, so liegt es nahe, den Knoten der HIT-Instanz ein zusätzliches Attribut hinzuzufügen, dessen Wert den Zustand des Grapheditors, d.h. den visualisierten Graph darstellt.

Ferner ist es im Hinblick auf ein Mehrbenutzersystem möglich, die Sichtweisen der verschiedenen Benutzer (vgl. wieder die Benutzung eines WFMS durch den Administrator und Aufgabenbearbeiter) durch jeweils ein benutzerspezifisches Attribut zu modellieren. In Abbildung 2-7 wird die Grundidee einer entsprechenden Erweiterung angedeutet.

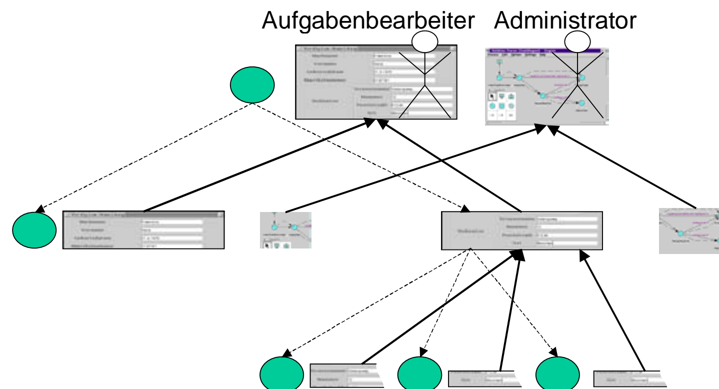


**Abbildung 2-7: Benutzung eines WFMS als Erweiterung einer HIT-Instanz**

## 2.3 Entwicklung interaktiver Informationssysteme

Zur Entwicklung interaktiver Informationssysteme gibt es grundsätzlich die Möglichkeit, ein bereits vorhandenes Standardprodukt (etwa *SAP R/3*) anzupassen [Kir99] oder anwendungsspezifische Programme ([Den91] spricht von *Individualsoftware*) zu entwickeln, die auf geeigneten Systemen (etwa einem DBMS) basieren. Letzteres wird in diesem Abschnitt im Überblick diskutiert. Anwendungsspezifische Programme werden üblicherweise in verschiedenen Entwicklungsphasen (Abschnitt 2.3.1) unter Verwendung geeigneter Werkzeuge (Abschnitt 2.3.2) entwickelt.

### 2.3.1 Entwicklungsphasen

Sowohl im akademischen Umfeld [BDD93] als auch in der Praxis [Bal00, Den91] wird zur Software-Entwicklung die Verwendung von Vorgehensmodellen mit verschiedenen Phasen propagiert. Abbildung 2-8 zeigt ein typisches, an das *Wasserfallmodell* [Boe81] angelehntes, sehr einfach gehaltenes Vorgehensmodell, das die bekannten Phasen und die damit in Verbindung stehenden Dokumente visualisiert. Ferner deutet Abbildung 2-8 die typische iterative Vorgehensweise an, die sich gerade bei der Entwicklung anwendungsspezifischer interaktiver Informationssysteme ergibt (siehe die Rückwärtspfeile). Hier lässt sich oftmals erst beim Testen von Prototypen eine Konkretisierung (bzw. Validierung) der von den Anwendern oft nur schwer zu formulierenden Anforderungen erreichen [Den91].

Ein ***Prototyp*** kann als eine Systemsimulation an der Benutzungsoberfläche betrachtet werden, die den Anwendern möglichst frühzeitig vorgeführt wird, um die Anforderungen und deren Umsetzung mit ihnen abzustimmen [Den91]. Wünschenswert wäre es daher, wenn ausführbare Prototypen bereits während der Analysephase zur Verfügung stehen würden. Entwickler und spätere Anwender (Fachexperten) könnten dann die Prototypen bereits während der Analysephase diskutieren und somit spätere, teuere Anpassungen der interaktiven Applikation vermeiden. Im Idealfall werden Prototypen mit entsprechenden generativen Werkzeuge aus einer Anforderungsbeschreibung generiert. Da während der Analysephase selten bereits eine vollständige Systemspezifikation vorliegt, muss ein solches Werkzeug in der Lage sein, auch aus Teilen davon einen ablauffähigen

Prototyp zu generieren. Das implementierte Generierungswerkzeug dieser Arbeit (EMUGEN) kann als ein solcher Prototypgenerator verwendet werden.

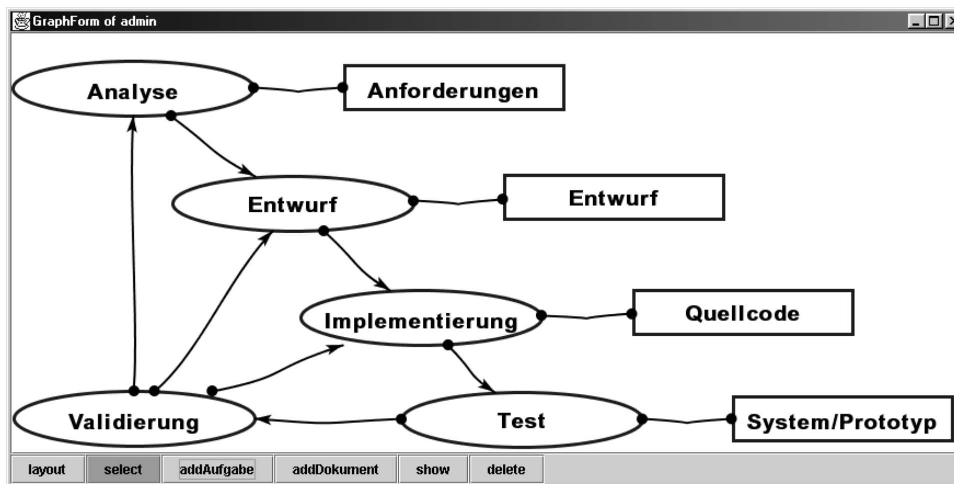


**Abbildung 2-8: Entwicklungsphasen dargestellt durch einen Grapheditor**

Auf einer anderen gedanklichen bzw. sprachlichen Ebene betrachtet, zeigt Abbildung 2-8 einen Grapheditor, der es ermöglicht, Phasen, ihre kausalen Abhängigkeiten und mit den Phasen in Verbindung stehende Dokumente auf diagrammsprachliche Weise einzugeben bzw. zu visualisieren.

Abbildung 2-8 stellt also auch ein Werkzeug zur diagrammsprachlichen Aufgabendefinition dar, wobei als aktuelle Aufgabendefinition die Entwicklung interaktiver Systeme dargestellt wird. Jedoch ist die verwendete Notation (zumindest für sich alleine betrachtet) zu ungenau, um sie sinnvoll als konstruktive Spezifikation zur automatischen Generierung eines interaktiven Aufgabenverwaltungssystems zu verwenden. In Kapitel 5 wird jedoch eine entsprechende Erweiterung dieser Notation vorgestellt, die man auch zur konstruktiven Aufgabendefinition verwenden kann.

### 2.3.2 Werkzeuge

Neben Vorgehensmodellen bieten Werkzeuge eine wichtige Unterstützung bei der Entwicklung interaktiver Informationssysteme [Bal00]. Wir beschränken uns auf die für diese Arbeit wesentlichen Werkzeuge(kategorien), die vorrangig zur Entwicklung des interaktiven Aspekts und der dazu notwendigen Benutzungsoberflächen benötigt werden. In Abbildung 2-9 werden diese Werkzeuge in Schichten eingeteilt. Werkzeuge höherer, abstrakterer Schichten benutzen Werkzeuge niedrigerer Schichten.

Die Schichteneinteilung von Abbildung 2-9 ist vereinfacht und idealisiert. D.h. es ist natürlich möglich, dass ein CADUI-Werkzeug nicht auf einem UIMS oder einem Maskengenerator (die Unterschiede zwischen diesen Werkzeugtypen werden unten erklärt), sondern direkt auf einem UI-Toolkit, aufbaut. Die Namen der Werkzeugkategorien entstammen teilweise der Literatur zu CADUI [Sch97, GFJ96] und zur Software-Entwicklung [Bal00, Den91]. Wir werden diese Schichteneinteilung auch zur Beschreibung der wichtigsten CADUI-Werkzeuge im Kapitel 7 benutzen, wobei dann zusätzlich die jeweiligen Modelle und generierten Komponenten hinzufügen werden (vgl. z.B. Abbildung 7-1).

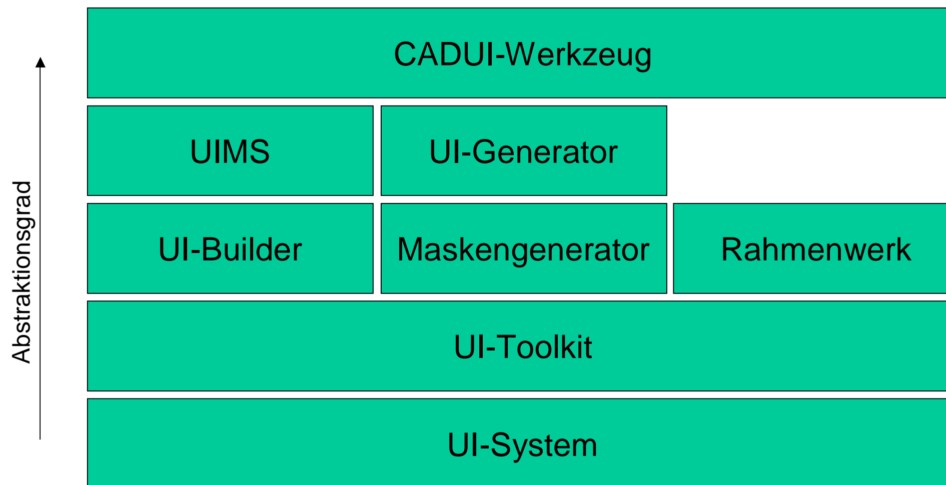


**Abbildung 2-9: Werkzeuge zur Entwicklung interaktiver Systeme**

**UI-System**

Ein ***UI-System*** (z.B. *X-Windows*, *Windows*) bietet dem Entwickler auf prozeduraler, programmiersprachlicher Ebene eine ***API*** (*application programming interface*) zur ereignisgesteuerten Programmierung eines interaktiven Systems. Meist führen dabei festgelegte Interaktionen des Benutzers (*Ereignisse*, z.B. ein Mausklick auf einem bestimmten Bildschirmbereich) zum Aufruf einer vorher durch den Programmierer festgelegten Prozedur bzw. bei objektorientierten Systemen einer entsprechenden Methode. Da moderne Benutzungsoberflächen eine Vielzahl möglicher Interaktionen bieten, sind die APIs von UI-Systemen sehr komplex und daher fehleranfällig zu programmieren. Der dafür erhaltene Performanzgewinn spielt bei den heutigen Rechnerleistungen meist nur noch bei speziellen Problemstellungen, etwa im CAD-Bereich, eine Rolle.

Ein neues Aufgabenfeld für die unmittelbare Nutzung der API von UI-Systemen ergibt sich jedoch durch die Anforderung nach effizienten Benutzungsoberflächen für mobile Endgeräte. Um die hier vorliegenden physikalischen und softwaretechnischen Engpässe zu überwinden, wird in [Har01] ein hardwarenahes UI-System vorgestellt, welches in Maschinensprache vorliegende UI-Programme trotz der technischen Einschränkungen in Echtzeit ausführen kann.

**UI-Toolkit**

***UI-Toolkits*** (z.B. *X-Motif*, *Microsoft Foundation Classes/MFC*, *Java-AWT* und *-Swing*) bieten eine abstraktere und damit für den Entwickler bequemere API zur Erstellung der Benutzungsoberfläche als ein UI-System. Darüber hinaus ermöglicht ein UI-Toolkit prinzipiell die Implementierung plattformunabhängiger Anwendungen (weil sie die API des plattformabhängigen UI-System verschatten). Oft stehen UI-Toolkits als objektorientierte Klassenbibliotheken zur Verfügung (z.B. MFC, AWT). Auch bei Verwendung eines UI-Toolkits arbeitet der Entwickler auf prozeduraler bzw. objektorientierter Ebene, wobei bei modernen UI-Toolkits (z.B. Swing) auch deklarative Implementierungskonzepte wie contraintbasiertes Layoutmanagement zum Einsatz kommen.

### UI-Builder

***UI-Builder*** (z.B. *VisualBasic*, *Forte*) ermöglichen es dem Entwickler, formularorientierte Benutzungsoberflächen visuell, also ohne Programmierung, zu gestalten. Ein Vorteil von UI-Buildern liegt daher in der Möglichkeit, auch Entwickler ohne Programmierkenntnisse in den Entwicklungsprozess mit einzubeziehen. Der größte Nachteil von UI-Buildern besteht darin, dass eine Änderung der Anforderungen (was in der Praxis häufig vorkommt) bei einer visuell entworfenen Benutzungsoberfläche meist zum Anpassen („Neuzeichnen") des gesamten Formulars führt.

Dieses Problem kann bei geschickter Verwendung von UI-Toolkits vermieden werden, weil die Präsentation eines Formulars auch weitgehend unabhängig von konkreten Anforderungen bezüglich der Eingabe-, Ausgabe- und Interaktionsmöglichkeiten (also der Elemente der abstrakten Benutzungsoberfläche nach [Sch97]) implementiert werden kann. Dazu wird algorithmisch definiert, wie die abstrakte Benutzungsoberfläche – möglicherweise abhängig von ergonomischen Regeln und physikalischen Größen wie der Bildschirmgröße – präsentiert wird. Dieser generative Ansatz ist eine der Grundideen von UI-Generatoren. Eine vereinfachte Form dieses Ansatzes, bei dem beispielsweise ergonomische Regeln nicht berücksichtigt werden, ist die Grundlage von Maskengeneratoren.

### Maskengenerator

***Maskengeneratoren*** (z.B. *OracleForms*, *MS-Access*, *(Sybase-)PowerBuilder*) ermöglichen die Generierung von Formularen und verwenden dazu als Eingabesprache meist ER-Modelle oder relationale Datenbanktabellendefinitionen. Dieser Ansatz eignet sich gut für interaktive Informationssysteme, bei denen der Datenaspekt im Vordergrund steht. Maskengeneratoren werden deshalb auch häufig von DBMS-Herstellern (z.B. Oracle) angeboten. Im Gegensatz zum bereits angesprochenen CADUI-Werkzeug BOSS ermöglichen es Maskengeneratoren jedoch i.d.R. nicht, dynamische Formulare zu erstellen, und benutzen keine ergonomischen Regeln bei der Generierung.

In den weiteren Kapiteln wird die Bedeutung dynamischer Formulare für die Berücksichtigung des Aufgaben- und Mehrbenutzeraspekts deutlich. Daher kann diese Arbeit nicht auf einem der oben genannten Maskengeneratoren basieren, sondern benötigt einen zu BOSS gleichwertigen UI-Generator.

### Rahmenwerk

Ein ***Rahmenwerk*** (engl. *Framework*) bildet einen objektorientierten, wiederverwendbaren Entwurf für eine allgemeine Problemstellung [JF88]. Ein Rahmenwerk wird durch eine Menge zusammengehörender, kooperierender Klassen gebildet. Zur Verwendung eines Rahmenwerks ist häufig die Implementierung konkreter Unterklassen von den gegebenen Klassen des Rahmenwerks notwendig. [Pre95] nennt die gegebenen Klassen eines Rahmenwerks anschaulich ***frozen-spots*** und die vom Entwickler zu implementierenden Klassen ***hot-spots***.

Rahmenwerke können sich in ihrer Komplexität stark unterscheiden. Beispiele für verhältnismäßig einfache Rahmenwerke ist die in Java [GJS96] mitgelieferte (durch das Entwurfmuster *Beobachter* [GHJ96] verallgemeinerte) MVC-Architektur (`java.util.observer`, `java.util.observable`) und die objektorientierte Struktur der von CUP [Hud99] generierten LALR-Parser. Bei CUP befindet sich der eigentliche Algo-

rithmus zum Parsen in einer Methode einer abstrakten Oberklasse (frozen-spot). CUP generiert einen konkreten Parser mit den erforderlichen Tabellen als Unterklasse davon (hot-spot).

Ein komplexes Rahmenwerk bildet beispielsweise das Grapheditor-Framework GEF [RHG99], womit anwendungsspezifische Grapheditoren – ebenfalls vorrangig durch Unterklassenbildung – erstellt werden können. Solche Rahmenwerke vermindern zwar den Implementierungsaufwand enorm, bedürfen jedoch auch einer komplexen Einarbeitungszeit, weil neben der Programmiersprache, in der sie entworfen wurden, die Funktionsweise der Oberklassen und die zugehörige API dem Entwickler bekannt sein müssen.

Ein Nachteil von Rahmenwerken besteht darin, dass die Implementierung verschiedener logischer Aspekte (z.B. Präsentation, Interaktionen bei Rahmenwerken für Grapheditoren) meist nicht in entsprechend ähnlich strukturierten Unterklassen vorgenommen werden kann. Der Grund dafür liegt in der sehr implementierungsnahen Form der Anpassung durch die Implementierung der hot-spots.

Dennoch hatte die Technik der Rahmenwerke starken Einfluss auf die Implementierung des Generierungskonzepts dieser Arbeit, weil es möglich ist, aus deklarativen Eingabemodellen die hot-spots eines Rahmenwerks zu erzeugen. Im Gegensatz zur Vorgehensweise bei CUP ist es dabei jedoch notwendig, beim Generierungsvorgang mehrere Eingabemodelle zu integrieren. Eine ähnliche Vorgehensweise wird auch bei den Arbeiten zum *aspektorientiertem Programmieren* [KLM97] propagiert.

#### UIMS

***User-Interface-Management-Systeme*** (***UIMS***, der Begriff wurde 1982 in Anlehnung an DBMS eingeführt) erweitern UI-Builder um die Möglichkeiten der Dialogdefinition [Lar92], die meist in Form einer deklarativen Regelsprache mit prozeduraler Schnittstelle erfolgt. Abstrakt betrachtet besteht eine Dialogdefinition aus Zuständen und Zustandsübergängen, die über Benutzerinteraktionen ausgelöst werden. Da UIMSs auf UI-Builder basieren, bleiben jedoch die oben angesprochenen Probleme der UI-Builder erhalten.

#### UI-Generator

***UI-Generatoren*** (z.B. BOSS [Sch97], ADAPTIVE FORMS [FS98], FORMGEN [BK99]) erweitern die Funktionalität von Maskengeneratoren um die Möglichkeit der Generierung dynamischer Formulare. Als Eingabe werden dabei meist Grammatiken als Datentypbeschreibungssprache verwendet. Wie in Abschnitt 7.1 genauer beschrieben, verfügt das BOSS-System über weitere Spezifikations- und Generierungsmöglichkeiten, wie z.B. die formale Beschreibung von applikationsunabhängigen Layout-Richtlinien.

# 3 EMU-Systeme

In diesem Kapitel werden die Benutzung, die Architektur und die generative Entwicklung von EMU-Systemen beschrieben. EMU-Systeme sind interaktive Informationssysteme, die mit dem in dieser Arbeit entwickelten CADUI-Werkzeug EMUGEN generiert werden. Das Kapitel ist daher wie das vorherige strukturiert, nur dass es nun um die automatisch generierten, interaktiven Informationssysteme dieser Arbeit geht.

Die zentrale Komponente eines EMU-Systems ist der Mehrbenutzer-Applikationsschnittstellenkern (MAK, vgl. Abbildung 1-2). Zum Verständnis der informellen Beschreibung der EMU-Systeme in diesem Kapitel sollte es genügen, den MAK vereinfacht als hierarchisch organisierte Datenstruktur zu betrachten. Eine genauere Betrachtung des MAKs erfolgt in den nachfolgenden Kapiteln.

## 3.1 Benutzung von EMU-Systemen

### 3.1.1 Konkrete Ausführung mit Formularen und Grapheditoren

Aufgrund der Diskussion in Abschnitt 2.1 ergibt sich die Verwendung von Formularen und Grapheditoren als grundlegende Typen von Benutzungsoberflächen zur Bearbeitung des MAKs (Abbildung 3-1). Abschnitt 2.2.2 motiviert ferner eine synchrone Bearbeitung im Sinne der MVC-Architektur.

Der rekursive Aufbau von Formularen über Basisinteraktionsobjekte (z.B. Textfelder zum Bearbeiten von Basiselementen wie Zeichenreihen) und zusammengesetzten Formularen ermöglicht die Integration bereits bestehender, komplexer Interaktionsobjekte (z.B. für Dokumente oder Tabellen) oder Interaktionsobjekte (z. B. Buttons oder Menus) zum Starten externer interaktiver oder batch-orientierter Applikationen.

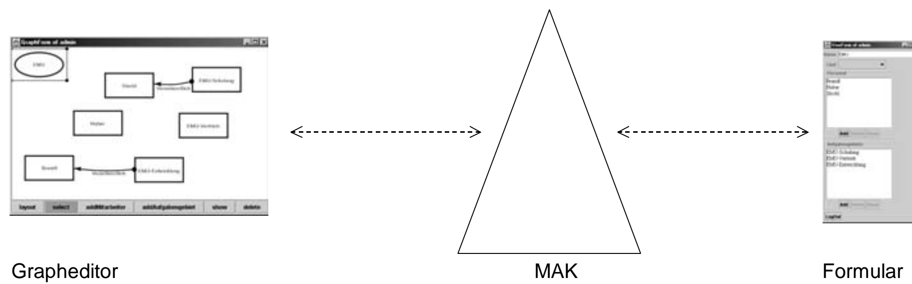


**Abbildung 3-1: Bearbeitung eines MAKs mit einem Grapheditor und einem Formular**

Ein Grapheditor ist beispielsweise dann von Nutzen, wenn durch Teile des MAKs auch Aufgaben und ihre kausalen Abhängigkeiten beschrieben werden (vgl. Abbildung 2-8).

### 3.1.2 Mehrbenutzerzugriff

Gemäß der Betrachtung der Benutzung von WFMS und DBMS im vorherigen Kapitel ergibt sich die Anforderung, dass im Allgemeinen mehrere Benutzer auf dem MAK arbeiten (Abbildung 3-2).

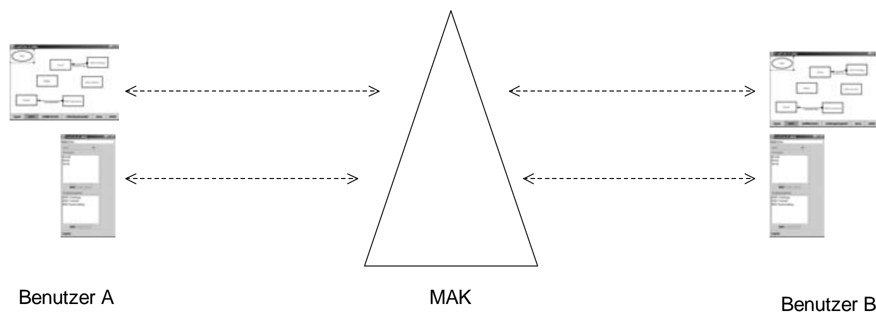


**Abbildung 3-2: Mehrbenutzerzugriff**

Jeder der Benutzer kann dabei mit einem Formular und/oder einem Grapheditor den MAK oder Teile davon bearbeiten. Dabei bearbeiten im Allgemeinen verschiedene Benutzer verschiedene Teile des MAKs. Dieser Zugriffsaspekt kann vom Entwickler des EMU-Systems im EMU-Benutzermodell festgelegt werden. Es ist jedoch auch ein dynamischer Ansatz zu berücksichtigen, bei dem bestimmte Benutzer (vgl. Administratoren eines DBMS, Abschnitt 2.1) zur Laufzeit des Systems den Zugriff der Benutzer auf den MAK bestimmen.

### 3.1.3 Systemanmeldung

Bei mehreren Benutzern ist eine Anmeldung beim System notwendig. Dabei ist eine Authentifizierung des Benutzers notwendig, um die entsprechende Benutzungsoberfläche zu initialisieren.

## 3.2 Architektur von EMU-Systemen

In diesem Abschnitt sollen die wichtigsten Aspekte der Architektur von EMU-Systemen motiviert und in ihren Grundzügen erklärt werden. Eine detailliertere Betrachtung folgt

bei der Diskussion der Ausführung generierter EMU-Systeme in den Abschnitten 4.2 und 5.2.

### 3.2.1 Form- und Graphadapter

Um die in Abbildung 3-1 und Abbildung 3-2 dargestellte Benutzung eines EMU-Systems zu ermöglichen, erfolgt eine Anwendung und Erweiterung der MVC-Architektur (vgl. Abschnitt 2.2.2). Dabei haben potentiell beliebig viele Benutzer über ihre Benutzungsoberflächen (Formulare und/oder Grapheditoren) synchronen Zugriff auf den MAK. Der MAK entspricht dabei der Model-Komponente von MVC, während die Benutzungsoberflächen die View/Controller-Teile von MVC bilden. Wird der MAK verändert, so werden sämtliche Benutzungsoberflächen benachrichtigt und aktualisiert.

Dabei ist jedoch zu berücksichtigen, dass ein Graph-Model aus technischen Gründen über andere, meist komplexere Attribute verfügt als ein Formular-Model. Daraus folgt, dass der MAK sowohl die Eigenschaften eines Graph-Models als auch die eines Formular-Models besitzen muss. Werden später weitere Benutzungsoberflächentypen eingebaut, so muss dabei auch der allgemeine Aufbau von MAKs erweitert werden.

Um bei der Integration der jeweils erforderlichen Modellattribute der beiden Modelltypen (Graph und Formular) flexibel und damit erweiterbar zu bleiben, bietet sich, sowohl zur formalen Beschreibung als auch zur technischen Realisierung, eine Entkopplung (funktionale Strukturierung) an. D.h. der Zugriff der Benutzungsoberflächen auf den MAK erfolgt über Zwischenstrukturen.

Wenn man im Sinne des Übersetzerbaus den MAK als Syntaxbaum auffasst, so kann eine solche Zwischenstruktur als synthetisches Attribut [WM97] (genauer: dessen Wert) der Wurzel aufgefasst werden. Wie bereits in Abschnitt 2.2.3 beschrieben, wird bei BOSS [Sch97] auf genau diese Weise das Layout der Benutzungsoberfläche errechnet.

Im Falle, dass genau ein Benutzer auf den MAK zugreift (der Mehrbenutzerzugriff wird im nächsten Abschnitt betrachtet), gibt es genau zwei dieser Attribute. Das eine davon stellt den MAK als graphisch zu visualisierenden Graph dar, das zweite dient als Model für das Formular.

Aus Sicht der Objektorientierung könnte dieses Konzept als Anwendung des *Adapter-Entwurfsmusters* [GHJ96] betrachtet werden. Der Adapter (also z.B. die Zwischenstruktur bzw. das Attribut für das Formular, genauer: dessen Wert) kann durch eine Benutzungsoberfläche unmittelbar am Bildschirm dargestellt werden. Dieses Vorgehen verhindert einerseits überflüssige Datenverwaltung, wenn nur eine Benutzungsoberfläche (Graph oder Formular) dargestellt wird und erleichtert es andererseits, später weitere Benutzungsoberflächentypen einzubauen. Wir verwenden im Folgenden den Begriff *Adapter*, weil wir ihn für diesen Zweck spezifischer als den Begriff *Attribut* erachten.

Pro Benutzungsoberfläche ergibt sich ein Adapter. Wir sprechen daher vom ***Graph***- bzw. ***Formadapter***. Der Zugriff der Benutzungsoberflächen über diese Adapter wird in Abbildung 3-3 veranschaulicht.

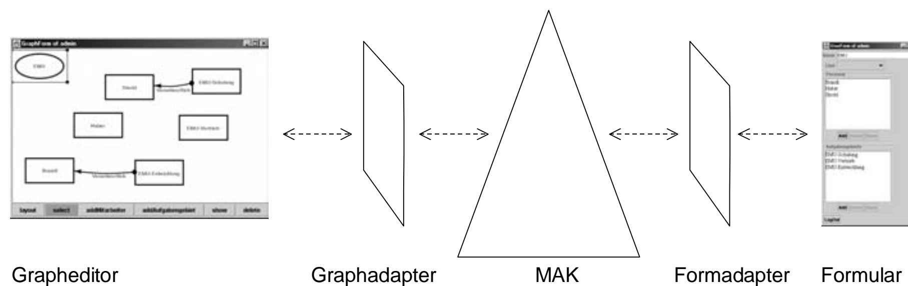


**Abbildung 3-3: Bearbeitung des MAKs über Graph- und Formadapter**

### 3.2.2 Zugriffsadapter

Nun soll der Fall diskutiert werden, dass mehrere Benutzer auf den MAK zugreifen (vgl. Abbildung 3-2). Technische Probleme beim Mehrfachzugriff auf den MAK werden von EMU durch eine Sequentialisierung aller Zugriffe gelöst.

Wie in der Literatur [Vos00, JBS97, Den91] üblich, erachten wir es also nicht als notwendig bei der Entwicklung eines interaktiven Informationssystems (vgl. Definition 1-1) eine Nebenläufigkeitskontrolle im Sinne der rechnergestützten Gruppenarbeit [Sch99] zu implementieren. Daher wird in einem EMU-System auch kein eingebauter Mechanismus zur Nebenläufigkeitskontrolle [Sch99] ausgeführt. Jedoch hat der Entwickler durch das Aufgaben- und Benutzermodell die Möglichkeit, eine (interaktive) Zugriffskontrolle zu spezifizieren, um einen zeitgleichen kritischen Zugriff auf einen gemeinsame Knoten des MAKs zur Laufzeit zu verhindern. Die entsprechenden Sprachelemente werden in Kapitel 5 und ein zugehöriges Beispiel in Abschnitt 6.2.3 diskutiert.

Durch das Benutzermodell wird festgelegt, auf welche Teile des gemeinsamen MAKs die einzelnen Benutzer lesend oder schreibend zugreifen können. Wiederum bietet es sich an, den im Allgemeinen eingeschränkten Zugriff mittels eines entsprechenden Adapters – dem sogenannten ***Zugriffsadapter*** – formal zu beschreiben und in gleicher Weise auch zu implementieren. Die grundlegende Idee wird in Abbildung 3-4 verdeutlicht.

Wie Abbildung 3-4 zeigt, greifen die einzelnen Graph- und Formadapter der verschiedenen Benutzer nicht direkt, sondern über die Zugriffsadapter auf den MAK zu.

Aus Übersetzerbausicht können die Zugriffsadapter erneut als ein synthetisches Attribut an der Wurzel des MAKs betrachtet werden, aus denen wiederum die Graph- und Formularattribute (bzw. Adapter) der einzelnen Benutzer berechnet werden.

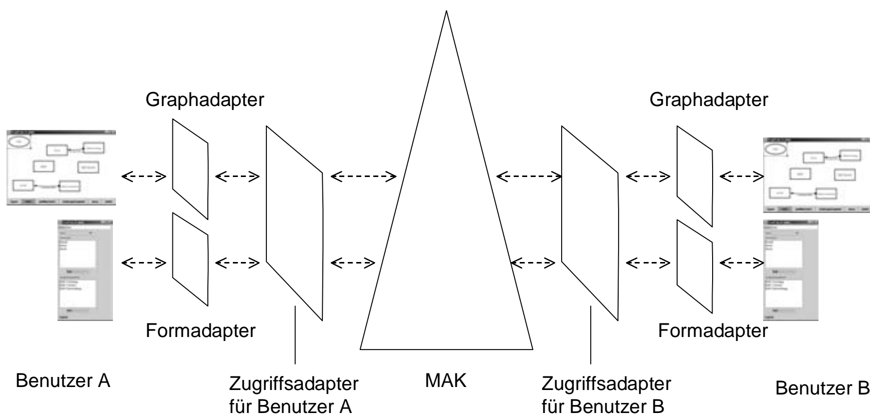


**Abbildung 3-4: Mehrbenutzerzugriff über Zugriffsadapter**

### 3.2.3 Layoutattribute

Aus formaler und auch aus implementierungstechnischer Sicht wäre es günstig, die bisherige Betrachtung der Adapter als synthetisches Attribut des MAKs zu verwenden, um damit den Zustand einer Benutzungsoberfläche vollständig zu definieren. Da wir uns jedoch auch dem Problem stellen, diese Benutzungsoberfläche effektiv den Benutzern am Bildschirm zu präsentieren, sind noch die folgenden Aspekte zu berücksichtigen:

- Zur graphischen Darstellung des MAKs mit Knoten und Kanten[2] werden Attribute benötigt, die z.B. die Koordinaten und Größe der graphisch visualisierten Knoten angeben.
- Komplexe Formulare benötigen eine Navigationsmöglichkeit.

Mit dem Begriff *Navigation* wird üblicherweise die Sichtweise verbunden, dass sich ein Benutzer durch eine komplexe Datenstruktur (hier der MAK) „bewegt" (vgl. die Benutzung eines Internetbrowsers), d.h. in einem Zustand auf einen bestimmten Teil der Datenstruktur lesend oder auch schreibend zugreift und über Interaktionsobjekte diesen Ausschnitt verändert (und sich damit im MAK „bewegt"). Aus abstrakter Sicht kann man dies ebenfalls als Attributierung (d.h. als Berechnung von Attributwerten [WM97]) auffassen, bei der für jeden Teil des MAKs durch ein Attribut angegeben ist, ob der Teil im aktuellen Zustand sichtbar sein soll (oder nicht) und dass in der zugehörigen Benutzungsoberfläche nur genau diese Teile gezeigt werden. Es liegt nahe, in diesem Zusammenhang allgemein von ***Layoutattributen*** zu sprechen.

Dabei stellt sich jedoch die Frage, an welcher Stelle Attribute verwaltet werden bzw. die Attributierung stattfindet. Der naheliegende Ansatz, die Layoutattribute alleine den Knoten des MAKs zuzuordnen, stößt beim Mehrfachzugriff auf Probleme, weil dann beispielsweise jeder Benutzer genau dieselbe Navigation durchführen müsste. Daher ergibt sich ein zweidimensionaler Zugriffsvektor auf die Layoutattribute, der aus einem Paar (MAK-Knoten, Benutzer) besteht. Die Layoutattribute werden dabei in Tabellen verwal-

[2] Der Graph könnte auch textuell oder als Tabelle dargestellt werden.

tet. Eine Interaktion auf Layoutebene [Sch97] (z.B. eine einfache Verschiebung eines graphischen Knotens) führt so nur zu einer Veränderung des entsprechenden Attributs (also z.B. der Koordinaten) in der zugehörigen Tabelle.

## 3.3 EMU-Generierungskonzept

In diesem Abschnitt wird der Zusammenhang der aus dem Einleitungskapitel bekannten Eingabemodelle, der EMUGEN-Teilkomponenten und der in den vorherigen Abschnitten dieses Kapitels besprochenen Komponenten generierter EMU-Systeme dargestellt (Abbildung 3-5).

Wir beschreiben im Folgenden im Überblick die einzelnen Teilkomponenten von EMUGEN. Die Modellierungssprachen werden detailliert in den beiden nachfolgenden Kapiteln diskutiert.

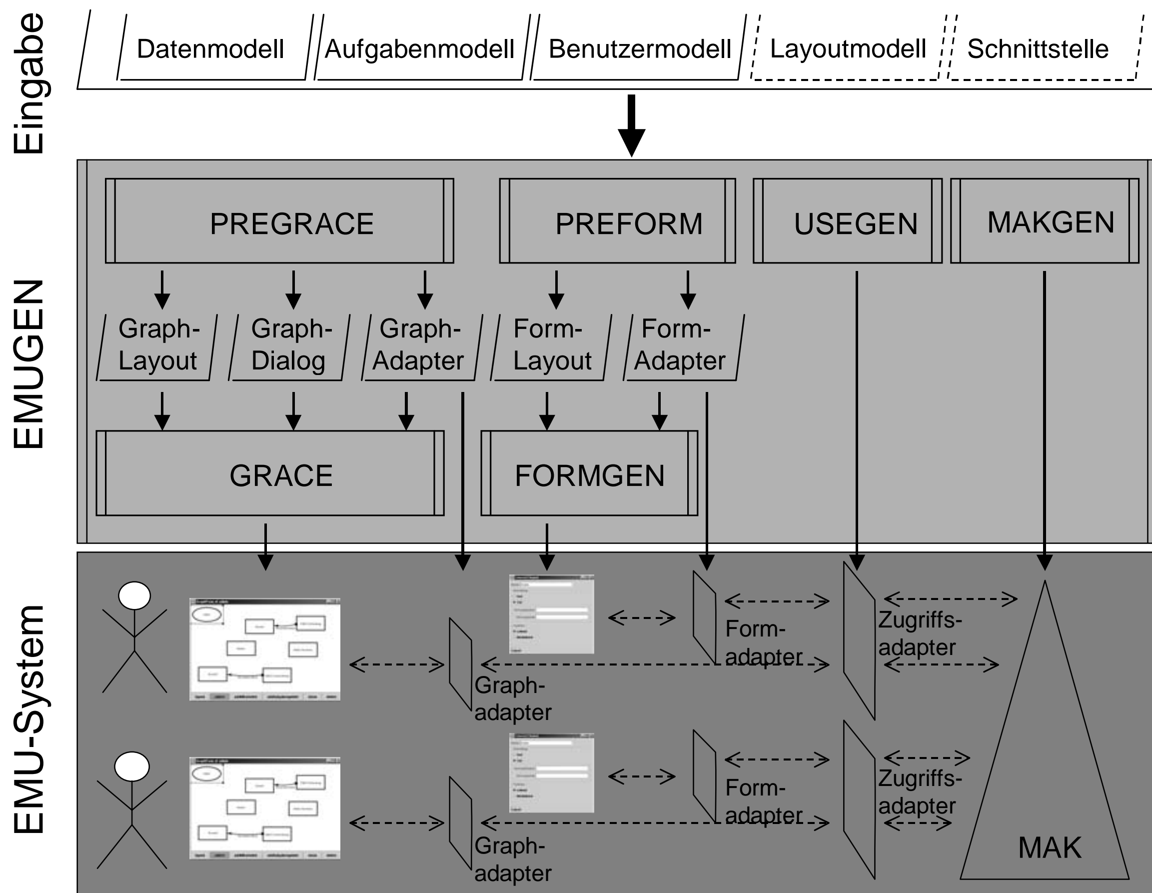


**Abbildung 3-5: Eingabemodelle, EMUGEN-Teilkomponenten und ein EMU-System**

EMUGEN benützt sechs Teilkomponenten, um EMU-Systeme zu erstellen. Die Komponenten PREGRACE und PREFORM können technisch als Präprozessoren betrachtet werden. Sie versorgen die auch unabhängig von EMUGEN einsetzbaren Generatoren GRACE [Kle99] und FORMGEN [BK99] mit entsprechenden Eingabemodellen. GRACE und FORMGEN erstellen die Grapheditoren bzw. Formulare, also die konkreten Benutzungsoberflächen, deren Applikationsschnittstelle (vgl. Abschnitt 2.2.1, Abbildung 2-5) durch die Graph- bzw. Formadapter gebildet werden. Die Komponenten USEGEN und

MAKGEN erstellen die (Benutzer-)Zugriffsadapter bzw. den Mehrbenutzer-Applikationsschnittstellenkern MAK.

Im Falle eines nicht explizit vom Entwickler definierten Layoutmodells greifen PREGRACE und PREFORM auf Standardlayoutregeln zurück. Wenn ein Layoutmodell verwendet wird, so werden üblicherweise nicht für alle Interaktionsdatentypen spezifische Layoutregeln angegeben. PREGRACE und PREFORM integrieren daher die spezifischen mit den Standardlayoutregeln in *Graph-* und *FormLayout*. Die entsprechenden Konzepte wurden in [Sch97] eingeführt.

***MAKGEN***

MAKGEN erstellt die zur Verwaltung des MAKs notwendigen Programmteile. Die Funktionalität dieser Programmteile beinhaltet nicht nur die eigentliche Datenverwaltung, sondern auch Funktionen, die bei einer abstrakten Interaktion ausgeführt werden. *Abstrakte Interaktion* bedeutet dabei, dass zunächst nicht festgelegt ist, wie diese Interaktion durch ein konkretes Interaktionsobjekt der Benutzungsoberfläche ausgelöst wird. Die Interaktionen können auch durch externe Systeme (vgl. Abbildung 1-2) ausgelöst werden. Durch eine Interaktion kann sich der MAK (auch strukturell) ändern. Formal betrachtet, bildet der MAK den Zustand eines durch die Eingabemodelle definierten Zustandsübergangssystems (formal definiert in den Abschnitten 4.3 und 5.3).

***USEGEN***

USEGEN erstellt die Zugriffsadapter, wobei das Benutzermodell als Eingabe eine besondere Rolle spielt. Zur Laufzeit ist jedem Zugriffsadapter ein Benutzer zugeordnet. Ändern sich die Eigenschaften des Benutzers, so kann dies – je nach der Definition des Benutzermodells – eine Änderung des Zugriffsadapters bewirken. Die Konstruktion und das dynamische Verhalten der Zugriffsadapter wird in Abschnitt 5.3.1 definiert.

***GRACE***

GRACE wurde im Rahmen dieser Arbeit durch eine Diplomarbeit [Kle99] entwickelt. GRACE benötigt in Anlehnung an das Seeheim-Modell [Pfa85] drei Modelle, um daraus Grapheditoren zu generieren:

- *GRACE-Layoutmodell* (in Abbildung 3-5: *Graph-Layout*): zur Beschreibung des Layouts graphischer Knoten und graphischer Kanten mit geometrischen Formen wie beispielsweise Rechtecke bzw. orthogonale Linien.
- *GRACE-Dialogmodell* (in Abbildung 3-5: *Graph-Dialog*): zur Beschreibung der Interaktionsmöglichkeiten zum Bearbeiten der durch graphische Knoten und Kanten dargestellten Teile des MAKs. Die Interaktionsmöglichkeiten werden mit Hilfe von Zustandsübergangssystemen spezifiziert.
- *GRACE-Problembereichsmodell* (in Abbildung 3-5: *Graph-Adapter*): zur Beschreibung der darzustellenden graphischen Knoten- und Kantentypen, woraus der zugehörige Graph-Adapter erstellt werden kann. Die Knoten- und Kantentypen ergeben sich aus bestimmten Datentypen des EMU-Datenmodells (erklärt in Abschnitt 4.2.3).

Im EMU-Layoutmodell können sowohl das GRACE-Layoutmodell als auch das GRACE-Dialogmodell bei der Konstruktion eines bestimmten Grapheditors angepasst werden.

Wir ordnen das GRACE-Dialogmodell dem EMU-Layoutmodell zu, weil es sich hier um Dialoge handelt, die auf verhältnismäßig niedrigem Abstraktionsniveau stattfinden. [Sch97] spricht in ähnlichen Fällen von Zustandsübergängen auf Layoutebene. Die Verwendung der GRACE-Eingabemodelle zur applikationsspezifischen Anpassung von Grapheditoren wird exemplarisch im nachfolgenden Kapitel behandelt.

***FORMGEN***

FORMGEN [BK99] benötigt die folgenden beiden Modelle, um daraus Formulare zu generieren:

- *FORMGEN-Layoutmodell* (in Abbildung 3-5: *Form-Layout*): zur Beschreibung von (Teil-)Formularen zum Bearbeiten des MAKs. Das Layout kann wie bei BOSS [Sch97] für konkrete Datentypen oder für eine Klasse von Datentypen (z.B. für alle Datentypen einer Applikation) festgelegt werden.
- *FORMGEN-Datenmodell* (in Abbildung 3-5: *Form-Adapter*): zur Beschreibung der logischen Datentypen der Formulare. Dazu gehören auch Aktionen, die den Datentypen zugeordnet werden, und die über Interaktionsobjekte (z.B. Buttons) am Bildschirm für den Benutzer ausführbar sind.

Das FORMGEN-Datenmodell ist nicht völlig identisch mit dem EMU-Datenmodell. Es beinhaltet zusätzlich zum EMU-Datenmodell Aktionen mit Vorbedingungen, die einen Teil des EMU-Aufgabenmodells darstellen (vgl. Abschnitt 5.1.1). Zur Laufzeit sind diese Aktionen genau dann durch das Formular ausführbar, wenn ihre Vorbedingung abhängig vom aktuellen Zustand des MAKs gültig ist. Ferner werden dem Benutzer über das Formular genau die durch den Zugriffsadapter festgelegten Teile des MAKs gezeigt. Aufgrund dieser Möglichkeiten (Aktionen und Zugriffsadapter) ist es nicht mehr notwendig, eine eigene Dialogmodellierungssprache im EMU-Generierungskonzept zu verwenden. Dieser Aspekt wird bei der Begründung des Aufgabenmodells in Abschnitt 5.4.1 und bei den Beispielen in Kapitel 6 genauer erläutert.

## 3.4 Entwicklungsszenarien im Überblick

Nun werden die beiden typischen EMU-Entwicklungsszenarien, *Mehrbenutzeranwendungen* und *Grapheditoren für Diagrammsprachen*, beschrieben. Wie das Beispiel *Adhoc Workflows* in Abschnitt 6.2.5 zeigt, können diese beiden Szenarien auch integriert werden. Konkrete Beispiele werden an dieser Stelle noch nicht präsentiert, weil die dafür notwendigen Modellierungssprachen erst in den beiden nachfolgenden Kapiteln eingeführt werden. Obwohl dynamische Formulare von EMUGEN (bzw. von der Teilkomponente FORMGEN) automatisch generiert werden, betrachten wir ihre generative Entwicklung nicht als typisch für EMU, weil sie bereits von anderen Arbeiten [Sch97] mächtige Generierungsmöglichkeiten dafür geschaffen wurden.

### 3.4.1 Entwicklung von Mehrbenutzeranwendungen

Soll eine Mehrbenutzeranwendung mit EMUGEN entwickelt werden, so verwendet man i.d.R. zumindest das Datenmodell, das Aufgabenmodell und das Benutzermodell.

Im Datenmodell werden sowohl die gesamte Struktur der Mehrbenutzeranwendung (also die des MAKs) als auch die Daten, die im Zusammenhang mit bestimmten Aufgaben einzugeben oder zu lesen sind, modelliert. Dazu werden im Datenmodell anwendungs-

spezifische Datentypen spezifiziert. D.h. es findet ein ähnliches Vorgehen statt wie bei der strukturierten [DeM79] und der objektorientierten Analyse [RBP91].

Im Aufgabenmodell werden den Datentypen Aktionen zugeordnet, durch deren Ausführung sich der Zustand des MAKs ändern kann und möglicherweise auch externe Systeme benachrichtigt werden. Ferner können hier Aufgaben definiert werden, an denen eventuell mehrere Benutzer über einen längeren Zeitraum arbeiten. Diese Art von Aufgaben können aus mehreren kausal voneinander abhängigen Phasen gebildet werden, wobei der Phasenübergang in Zusammenhang mit der Ausführung von Aktionen definiert werden kann. Auch die Aufgaben werden den Datentypen des Datenmodells zugeordnet.

Das Benutzermodell besteht aus einer Menge von Benutzertypen und einer Definition von Zugriffen der Benutzer auf den MAK. Ein Benutzertyp wird wie ein Datentyp des Datenmodells definiert, wodurch ihm auch eine Menge von Eigenschaften zugeordnet werden kann. Rollen im Sinne relationaler DBMSe können auch als Eigenschaft modelliert werden. Mit dem gleichen Konzept, mit dem die Ausführung komplexer Aufgaben mit mehreren Phasen modelliert wird, können die Aktivitäten von Benutzern modelliert werden (vgl. *Sachbearbeiter*-Konzept bei [Den91]). Dazu wird ein Modell der Benutzeraktivitäten entworfen und den Benutzertypen zugeordnet.

Bei vielen Mehrbenutzeranwendungen genügt die Verwendung von Formularen als Benutzungsoberfläche, wobei mögliche externe Applikationen durch die Aktionen des Aufgabenmodells gestartet und spezielle, anwendungsspezifische Interaktionsobjekte in die Formulare integriert werden können.

### 3.4.2 Entwicklung von Grapheditoren für Diagrammsprachen

Die graphischen Grundelemente einer Diagrammsprache sind graphische Knoten und Kanten (im Folgenden: ***Graphelemente***). Die Graphelemente dienen der Visualisierung bzw. Bearbeitung von (logischen) Knoten des MAKs im zugehörigen Grapheditor.

Bei der Entwicklung einer Diagrammsprache beschreibt man zunächst die abstrakte Syntax der Sprache durch die Datentypen des EMU-Datenmodells (woraus sich verschiedene Knoten- und Kantentypen ergeben) und bestimmt dadurch die Struktur des MAKs, der in diesem Fall als abstrakter Syntaxbaum dient.

Aus der abstrakten Syntax kann durch Anwendung der Standardlayoutregeln bereits ein ausführbarer Grapheditor generiert werden, der im Allgemeinen jedoch noch nicht den ergonomischen Anforderungen an die jeweils gewünschte Diagrammsprache genügt. D.h. durch die Standardlayoutregeln erfolgt zwar eine Zuordnung von Teilen des Syntaxbaums zu graphischen Knoten und Kanten, wodurch ein Syntaxbaum bereits durch den Grapheditor bearbeitet werden kann, jedoch haben die Graphelementen die in den Standardlayoutregeln festgelegte Form. Dort kann (je nach Festlegung der Standardlayoutregeln) beispielsweise definiert sein, dass graphische Knoten durch Rechtecke dargestellt werden. Wenn es sich nun bei der applikationsspezifischen Diagrammsprache beispielsweise um Zustandsübergangsdiagramme für Automaten handelt, so wären üblicherweise Ovale und nicht Rechtecke als geometrische Form für die graphischen Knoten der Zustände gewünscht.

Um die applikationsspezifische Darstellung zu erhalten, kann der Entwickler das EMU-Layoutmodell (genauer: durch die Teile die das GRACE-Layout- und Dialogmodell bil-

den) und damit das Layout des Grapheditors an die jeweiligen Anforderungen anpassen. Dem obigen Beispiel folgend könnte etwa zu definieren sein, dass Zustände durch Kreise oder Ovale zu präsentieren sind.

Bei der Entwicklung einer Diagrammsprache kann bzw. sollte die abstrakte Syntax durchaus so entworfen werden, dass sich die notwendigen semantischen Auswertungen möglichst einfach implementieren lassen. Sämtliche Anforderungen im Hinblick auf die graphische Darstellung können, bzw. sollten, allein durch das Layoutmodell vorgenommen werden. Wenn man – erneut dem obigen Beispiel folgend – die Ausführung von Zustandsübergangsdiagrammen (genauer: den durch sie dargestellten Automaten) implementieren will, so kann man in der abstrakten Syntax die Zustandsübergänge direkt den Zuständen zuordnen, womit man bereits die Automatentabelle vorliegen hat und damit die Verarbeitung einer Eingabefolge sehr bequem implementieren kann.

# 4 Datenmodell

In diesem und dem darauffolgenden Kapitel werden die drei für das EMU-Generierungskonzept zentralen Modelle genauer beschrieben:

- Datenmodell
- Aufgabenmodell
- Benutzermodell

Zunächst wird das Datenmodell betrachtet, weil es der einzige Teil der Eingabespezifikation ist, der auch alleine als komplette Eingabespezifikation verwendet werden kann und aus dem – zusammen mit dem optionalen Layoutmodell – bereits weitgehend die zur Ausführung notwendigen Benutzungsoberflächen konstruiert werden können. Aus dieser Tatsache ergibt sich ein enger Bezug zwischen dem Daten- und Layoutmodell. Daher betrachten wir die wesentlichen Aspekte des Layoutmodells ebenfalls in diesem Kapitel (Abschnitte 4.2.2, 4.2.3).

Durch das Datenmodell wird eine Menge von Datentypen spezifiziert, mit denen die Struktur, der Zustand und die Interaktionsmöglichkeiten zugehöriger EMU-Systeme festgelegt werden. Die entsprechenden Sprachelemente bilden das Thema von Abschnitt 4.1. Die Ausführung eines von einem Datenmodell konstruierten EMU-Systems wird in Abschnitt 4.2 informell dargestellt, wobei auch der wichtige Zusammenhang zu den zugehörigen konkreten Benutzungsoberflächen hergestellt wird. Es folgt im darauffolgenden Abschnitt eine formale Betrachtung der Ausführung von EMU-Systemen und abschließend eine Begründung einiger Entscheidungen bezüglich des Sprachdesigns.

## 4.1 Sprachelemente

Ein EMU-Datenmodell besteht aus einer Menge rekursiver Datentypdefinitionen, von denen genau ein Datentyp als Startdatentyp ausgezeichnet ist. Da diese Datentypen eine direkte Benutzerinteraktion ermöglichen, werden sie als ***Interaktionsdatentypen (IDT)*** bezeichnet.

Es stehen folgende Sprachelemente zur Verfügung:

- Tupel
- Variante
- Liste
- Referenz
- Basistyp

Wir bezeichnen die Sprachelemente des EMU-Datenmodells auch als ***Metatypen***. Implizit wird durch einen Interaktionsdatentyp *T* ein interaktives System festgelegt, das durch Formulare und Grapheditoren bearbeitet werden kann. Wir nennen solch ein interaktives System ***EMU-System vom Typ T***.

In vielen praktischen Anwendungen reicht eine implizite Festlegung der Interaktionen als vollständige Umsetzung der fachlichen Anforderungen nicht aus. Daher bietet EMU eine Kommandosprache zur Anpassung des implizit definierten Zustandsübergangssystems, die einen Teil der Schnittstelle zu externen Systemen (vgl. Abbildung 1-2) bildet. Dies wird mit dem Aufgabenmodell im nächsten Kapitel diskutiert. Durch das Aufgabenmodell und das Benutzermodell kommen also weitere Argumente für die Konstruktion eines EMU-Systems hinzu.

Wir wollen die Sprachelemente des Datenmodells exemplarisch anhand eines Informationssystems zur Verwaltung einer Abteilung einführen.

**Beispiel 4-1: Informationssystem *Abteilung***

*Zu einer Abteilung soll der Abteilungsname, das Personal, ein Chef sowie eine Liste von Aufgabengebieten, die von der Abteilung zu bearbeiten sind, verwaltet werden. Das Personal besteht aus Mitarbeitern, bei denen unterschieden werden soll, ob sie festangestellte oder freie Mitarbeiter sind und ob sie eine leitende oder eine nichtleitende Funktion besitzen. Zu festangestellten Mitarbeitern soll das Eintrittsdatum, zu freien der Vertragsbeginn und das Vertragsende verwaltet werden. Chef der Abteilung ist eine festangestellte Person, die eine leitende Funktion besitzt.*

*Zu Aufgabengebieten soll der Name, eine Liste zugehöriger Teilaufgaben, sowie ein verantwortlicher Mitarbeiter geführt werden. Teilaufgaben besitzen eine Beschreibung und können entweder wiederum ein eigenes Aufgabengebiet oder eine (terminale) Aufgabe darstellen. Pro Aufgabe ist ein Bearbeiter, die notwendigen Kenntnisse und die durchschnittliche Wochenbelastung zu führen.*

Im folgenden Datenmodell werden diese Anforderungen formalisiert dargestellt:

```
Abteilung::=String:Name Chef Mitarbeiter*:Personal
            Aufgabengebiet*:Aufgabengebiete
Mitarbeiter::=String:Name Funktion Anstellung
Anstellung::=Fest | Frei
Fest::=String:Eintrittsdatum
Frei::=String:Vertragsbeginn String:Vertragsende
Funktion::=Leitend | Nichtleitend
```

```
Chef::=Name->Mitarbeiter
       [Anstellung.Fest & Funktion.Leitend]
Aufgabengebiet::=String:Name Teilaufgaben
                 Verantwortlich
Teilaufgaben::=Teilaufgabe*
Teilaufgabe::=Aufgabentyp String:Beschreibung
Aufgabentyp::=Aufgabengebiet | Aufgabe
Aufgabe::=Bearbeiter Text:NotwendigeKenntnisse
            Integer:StundenProWoche
Bearbeiter::=Name->Mitarbeiter
Verantwortlich::=Name->Mitarbeiter [Funktion.Leitend]
```

Die textuelle Beschreibung eines EMU-Datenmodells ist an die BNF-Notation angelehnt, wobei wir damit keine formale Sprachen, sondern Datentypen des Problembereichs beschreiben (vgl. etwa [Eic93], Sortendeklarationen bei BOSS [Sch97], das Data Dictionary bei der strukturierten Analyse SA [DeM79]). Die vollständige Beschreibung der Eingabesyntax erfolgt in Anhang A).

Jeder Typidentifikator ist entweder Basistyp (etwa String, Integer) oder wird durch genau eine Produktion des EMU-Datenmodells definiert. Durch die erste Produktion eines Datenmodells *DM* wird der ***Starttyp*** $start_{DM}$ definiert (im Beispiel: `Abteilung`). Wir beschreiben im Folgenden die verschiedenen Arten von Produktionen.

### 4.1.1 Tupelproduktionen

Tupelproduktionen sind das EMU-Sprachelement zur Bildung zusammengesetzter Typen. Auf die ***Komponenten eines Tupels*** kann über die innerhalb einer Produktion paarweise verschiedenen Bezeichnungen der Komponenten (***Selektoren***) zugegriffen werden. Optional kann auf explizite Selektoren verzichtet werden, wie im Beispiel bei der Komponente `Chef` des Tupels `Abteilung`. Ist dies der Fall, so wird der Typ der Komponente als impliziter Selektor verwendet. Im Beispiel ist daher `Chef` sowohl Bezeichner eines Interaktionsdatentyps als auch Selektor einer Komponente von `Abteilung`.

### 4.1.2 Variantenproduktionen

Varianten ermöglichen die Modellierung einfacher Unterscheidungen (im Beispiel: `Funktion`) und von Interaktionsdatentypen mit unterschiedlichen strukturellen Ausprägungen (im Beispiel: `Anstellung` und `Aufgabentyp`). Die paarweise verschiedenen Interaktionsdatentypen auf der rechten Seite nennen wir die ***Alternativen einer Variante***. Bei der Modellierung einfacher Unterscheidungen kann darauf verzichtet werden, die Alternativen einer Variante explizit zu definieren. Nicht explizit definierte Alternativen einer Variante (Ausnahme: Basistypen) werden implizit als Tupel ohne Komponenten (wir sprechen von ***Nulltupel***) betrachtet (im Beispiel: `Leitend`). Formal führen wir für jede nicht definierte Alternative `A` eine implizite Produktion `A::=ε` ein, falls eine solche noch nicht vorhanden ist[3].

[3] Der Sinn dieser impliziten Ergänzungen wird in Abschnitt 4.1.6 erklärt.

Wie im Beispiel der Interaktionsdatentyp `Aufgabengebiet` zeigt, können durch Variantenproduktionen rekursive Datentypen modelliert werden. Anders als bei den Sortendeklarationen in [Bro98, Sch97] oder bei funktionalen Sprachen können beim EMU-Datenmodell die Alternativen eines Variantentyps beliebige Typen, bzw. sogar nicht explizit definiert (Nulltupel), sein und damit insbesondere auch als Alternative bei verschiedenen Varianten vorkommen (z.B. sind auch folgende Produktionen in einem EMU-Datenmodell erlaubt: `X::=Y|Z Y::=X|Z`).

Die Varianten des EMU-Datenmodells dienen vorrangig dazu, die Alternativen als mögliche (strukturelle) Ausprägungen der entsprechenden Variante zu definieren. Hier besteht ein – in Abschnitt 4.4.4 genauer erläuterter – Unterschied zur isa-Beziehung des ER-Modells [Che76] oder zur Vererbung im objektorientierten Sinne. D.h. es handelt sich bei den Varianten um ein implementierungsnäheres und für die Generierung von Benutzungsoberflächen besser geeignetes Konzept. Dennoch können durch Varianten isa-Beziehungen bzw. Vererbungsaspekte, wie sie bei der Datenmodellierung des Problembereichs notwendig sind, kompakt ausgedrückt werden.

### 4.1.3 Listen und Listenproduktionen

Mengenwertige Tupelkomponenten (Listen) vom Elementtyp `E` können – wie in der BNF-Notation üblich – durch `E*` definiert werden (im Beispiel: `Mitarbeiter*:Personal`). Die Anzahl der ***Listenelemente*** ist zur Laufzeit unbegrenzt.

Listen können auch als Typ, d.h. durch eine Listenproduktionen, definiert werden (im Beispiel: `Teilaufgaben::=Teilaufgabe*`). Formal führen wir für jede mengenwertige Tupelkomponente `T*` immer eine implizite Produktion `TList::=T*` ein, falls eine solche noch nicht vorhanden ist.

### 4.1.4 Referenzenproduktionen

Durch Referenzenproduktionen (im Beispiel: die Produktion von `Chef`) können Redundanzen und damit Eingabefehler verhindert werden, was im Kontext der Generierung von Benutzungsoberflächen besonders wichtig ist. Im Gegensatz zum programmiersprachlichen Referenzbegriff können diese Referenzen (genauer: deren Werte) zur Laufzeit direkt vom Endbenutzer interaktiv bearbeitet werden. Insbesondere bei der Entwicklung eines Grapheditors für eine Diagrammsprache sind Referenzen nützlich, weil sie graphisch in natürlicher Weise durch Kanten dargestellt werden können (vgl. etwa Abbildung 4-10).

Referenzen beziehen sich immer auf einen Tupeltyp $T$ (***referierter Typ***, im Beispiel `Mitarbeiter`) unter Angabe eines Selektors von $T$ (***Referenzselektor,*** im Beispiel `Name`). Der Referenzselektor erscheint an dieser Stelle evtl. überflüssig, er ist jedoch notwendig, weil daraus das Benutzungsoberflächenelement gewonnen wird, welches ermöglicht, den Wert der Referenz zur Laufzeit interaktiv zu bearbeiten (vgl. Abbildung 4-7).

Bei der Definition von Referenzen ist die Angabe des referierten Typs und des Referenzselektors obligatorisch, während die Angabe einer ***Referenzbedingung*** (ein boolescher Ausdruck, z.B. `[Anstellung.Fest & Funktion.Leitend]`) optional ist. Wir ergänzen fehlende Referenzbedingungen mit `[true]`.

Die interaktive Bearbeitung einer Referenz zur Laufzeit entspricht der Auswahl eines Referenzwerts (im Beispiel: eines Mitarbeiters) aus einer Menge von ***Referenzkandidaten*** (im Beispiel: alle leitenden, festangestellten Mitarbeiter der Abteilung beim Referenztyp `Chef`). Die Menge der Referenzkandidaten ergibt sich dabei aus allen Elementen vom referierten Typ, welche die Referenzbedingung erfüllen. Eine Referenzbedingung dient also der Einschränkung der Menge von Referenzkandidaten.

### 4.1.5 Bedingungen

Bedingungen, wie die oben beschriebene Referenzbedingung, ermöglichen die Abfrage von bestimmten Zuständen zur Laufzeit. Beim Aufgabenmodell werden Bedingungen ferner als Vorbedingung von Aktionen und beim Benutzermodell als Definition von Benutzereigenschaften verwendet.

### 4.1.6 Implizite Ergänzungen

Um in Abschnitt 4.3 dem EMU-Datenmodell eine formale Semantik zuweisen zu können und dennoch die in Abschnitt 2.3.1 geforderten, unvollständig definierten Eingabemodelle zuzulassen, sind die bereits erwähnten, impliziten Ergänzungen notwendig. Wir gehen bei der formalen Betrachtung des Datenmodells in Abschnitt 4.3 daher davon aus, dass alle bisher angesprochenen Ergänzungen (durch das Werkzeug) durchgeführt wurden. Sämtliche weitere Forderungen an EMU-Datenmodelle, wie etwa das Verbot direkter und indirekter Rekursion bei Tupelproduktionen (z.B. `X::= X Y`) werden im formalen Abschnitt durch den Begriff eines *konstruktiven EMU-Datenmodells* festgelegt.

### 4.1.7 Graphische Darstellung

Bei der Software-Entwicklung werden nicht erst seit UML [OMG01] diagrammsprachliche Darstellungsmöglichkeiten zur Modellierung propagiert. So nennt [Bal00] als Nachteil von Datentypdefinitionssprachen die mangelnde graphische Darstellungsmöglichkeit. Eine solche wird jedoch etwa in [Eic93, Sch97] beschrieben und verwendet.

Zur automatischen Visualisierung textueller Datentypdefinitionen nach [Eic93, Sch97] verwenden wir das im Rahmen dieser Arbeit durch ein Systementwicklungsprojekt realisierte Werkzeug ***VisualClassgen*** (***VCG***) [DW01]. In Abbildung 4-1 wird die Visualisierung des EMU-Datenmodells *Abteilung* aus Beispiel 4-1 gezeigt.

Tupelkomponenten werden in der graphischen Darstellung durch einfache Linien und Listenelemente durch doppelte Linien verbunden. Die Alternativen einer Variante werden in geschweiften Klammern eingeschlossen und die Referenzproduktionen werden genau wie Tupelproduktionen gezeichnet.

Die sehr kompakte Darstellung in Abbildung 4-1 erhält man, wenn die erste Produktion (Starttyp) ganz oben gezeichnet wird und beim jeweils ersten Vorkommen eines definierten Typs auch seine definierende Produktion gezeichnet wird.

## 4.2 Beispielausführungen generierter EMU-Systeme

EMU-Systeme, die nur aus dem Datenmodell generiert wurden, sind konzeptionell durch genau einen Benutzer interaktiv zu bearbeiten, wobei sämtliche Interaktionen bereits implizit durch die Interaktionsdatentypen des Datenmodells festgelegt sind.

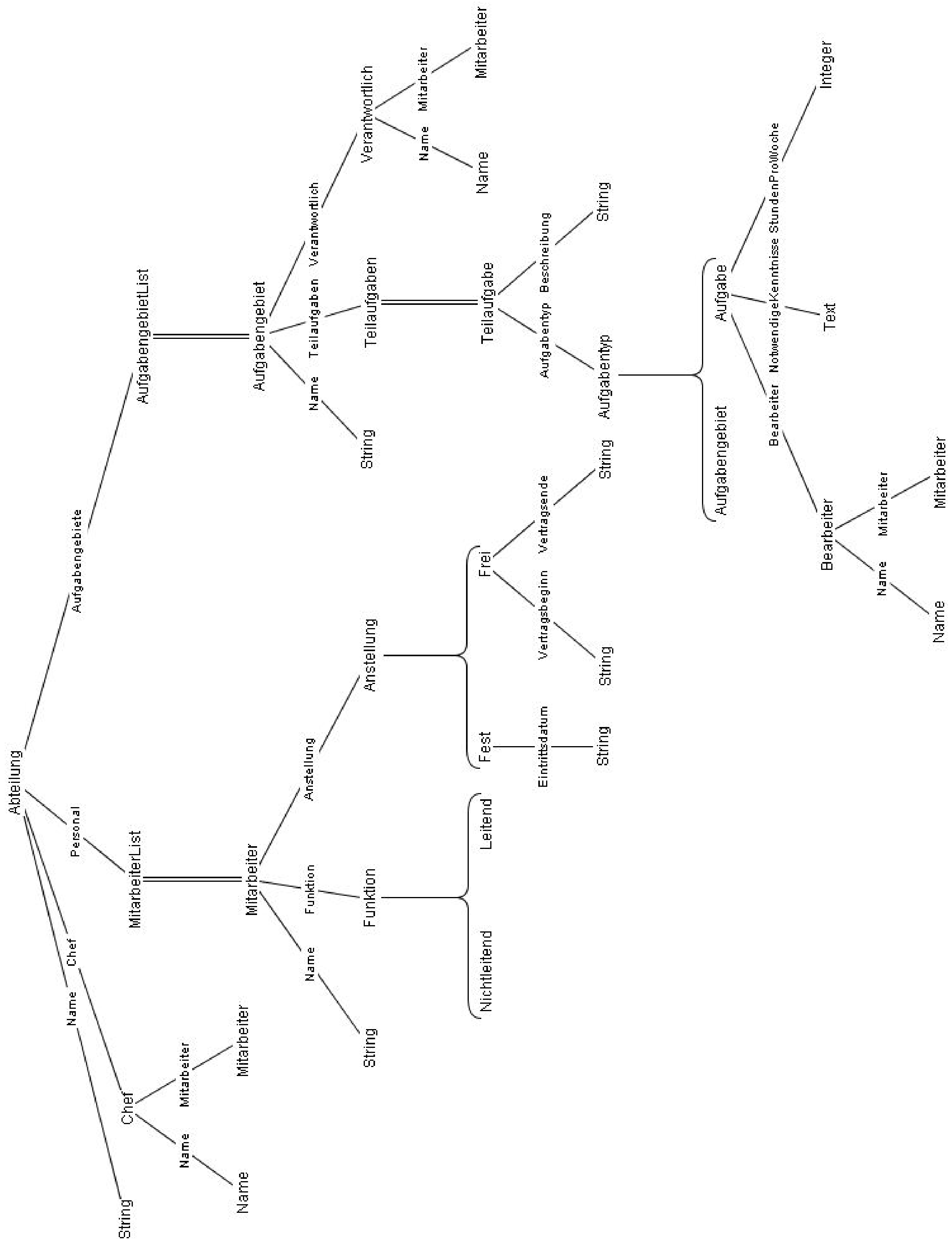


**Abbildung 4-1: Graphische Darstellung des EMU-Datenmodells `Abteilung`**

Diese vereinfachte Form von EMU-Systemen besteht aus dem hierarchisch strukturierten MAK (Mehrbenutzer-Applikationsschnittstellenkern) und aus einem Formular sowie einem Grapheditor zur konkreten Bearbeitung am Bildschirm (vgl. Struktureditor [RT88, Sch97]). Der MAK kann als interaktiv bearbeitbares Element einer komplexen Sortendeklaration im Sinne von [Sch97, Bro98] aufgefasst werden, wobei einige Unterschiede zu beachten sind. So kann ein MAK Referenzknoten beinhalten und stellt damit einen hierarchisch organisierten Graph dar, während Elemente einer komplexen Sortendeklaration Terme darstellen. Damit kann ein MAK komplexe Sachverhalte und Plausibilitätsbedingungen (etwa: *ein Chef ist immer ein leitender Mitarbeiter*) verwalten, während Terme lediglich syntaktische Informationen darstellen können.

Ferner sieht EMU einen iterativen bzw. inkrementellen Entwicklungsprozess [Bal00] vor, bei dem zunächst die Interaktionsdatentypen im Datenmodell definiert werden. Anschließend werden den Interaktionsdatentypen Aufgaben und Zugriffsrechte zugeordnet. Diese Art von Erweiterungen ist bei klassischen Sortendeklarationen in dieser Form nicht vorgesehen.

EMU-Systeme sind interaktive Informationssysteme und können daher nach [BDD93] als Transitionssysteme mit den folgenden Elementen beschrieben werden:

- Zustände
- Initialzustand
- Aktionen
- Zustandsübergang

Bevor in Abschnitt 4.3 diese Elemente formal eingeführt werden, sollen sie durch einen Beispielablauf eines EMU-Systems zum Beispiel 4-1 intuitiv vermittelt werden. Dazu werden in Abbildung 4-2 und Abbildung 4-3 drei Zustände eines Beispielablaufs visualisiert.

Bei diesem Beispielablauf werden, vom Initialzustand (Zustand (1) in Abbildung 4-2) eines EMU-Systems vom Typ *Abteilung* ausgehend, zunächst in der ersten Aktionsfolge $AF_1$ zwei Mitarbeiter und ein Aufgabengebiet eingegeben. Der Aufbau und die Bedeutung der (systeminternen) Aktionscodierung wird in Abschnitt 4.2.1 erklärt.

$$\begin{aligned} AF_1 = \langle & add(\langle 3\rangle,1), setValue(\langle 3,1\rangle,"Huber"), \\ & add(\langle 3\rangle,2), setValue(\langle 3,2\rangle,"Strobl"), \\ & add(\langle 4\rangle,1), setValue(\langle 4,1\rangle,"Schulung"), \\ & setValue(\langle 1\rangle,"EMUVermarktung")\rangle \end{aligned}$$

Vom Initialzustand gelangt das EMU-System, wie in Abbildung 4-2 skizziert, durch die Aktionsfolge $AF_1$ in den Zustand (2). Nun werden in der zweiten Aktionsfolge $AF_2$ Eigenschaften der Mitarbeiter sowie ein Mitarbeiter als Chef und ein weiterer als für das Aufgabengebiet verantwortlich definiert.

$$\begin{aligned} AF_2 = \langle & setAlt(\langle 3,1,2\rangle, Leitend), setAlt(\langle 3,1,3\rangle, Fest), setAlt(\langle 3,2,2\rangle, Leitend)), \\ & setRef(\langle 4,1,3\rangle,\langle 3,2\rangle), setRef(\langle 2\rangle,\langle 3,1\rangle)\rangle \end{aligned}$$

$AF_2$ führt von Zustand (2) in den Zustand (3) in Abbildung 4-3.

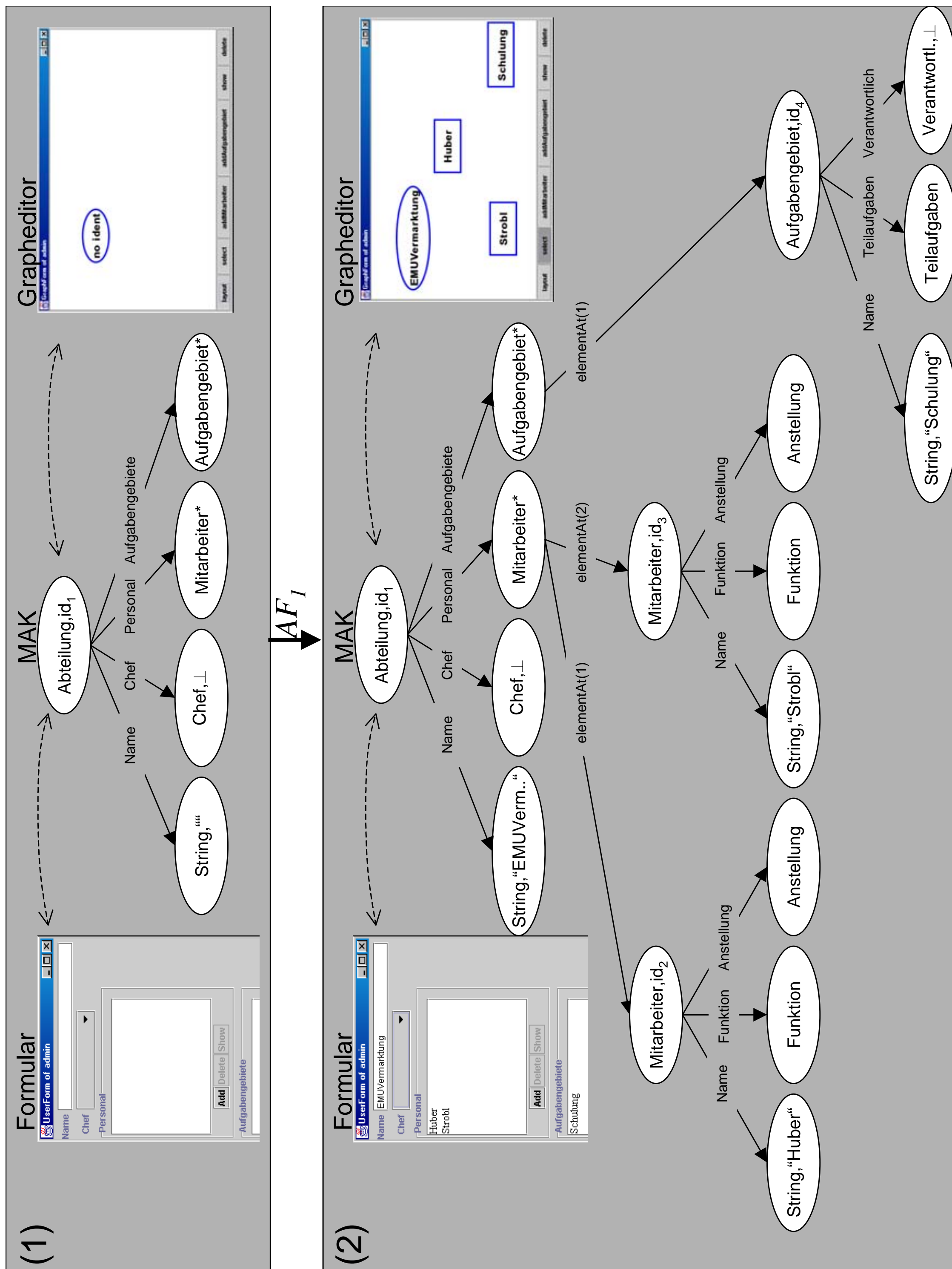


**Abbildung 4-2: Ausführung eines EMU-Systems vom Typ *Abteilung* (1)**

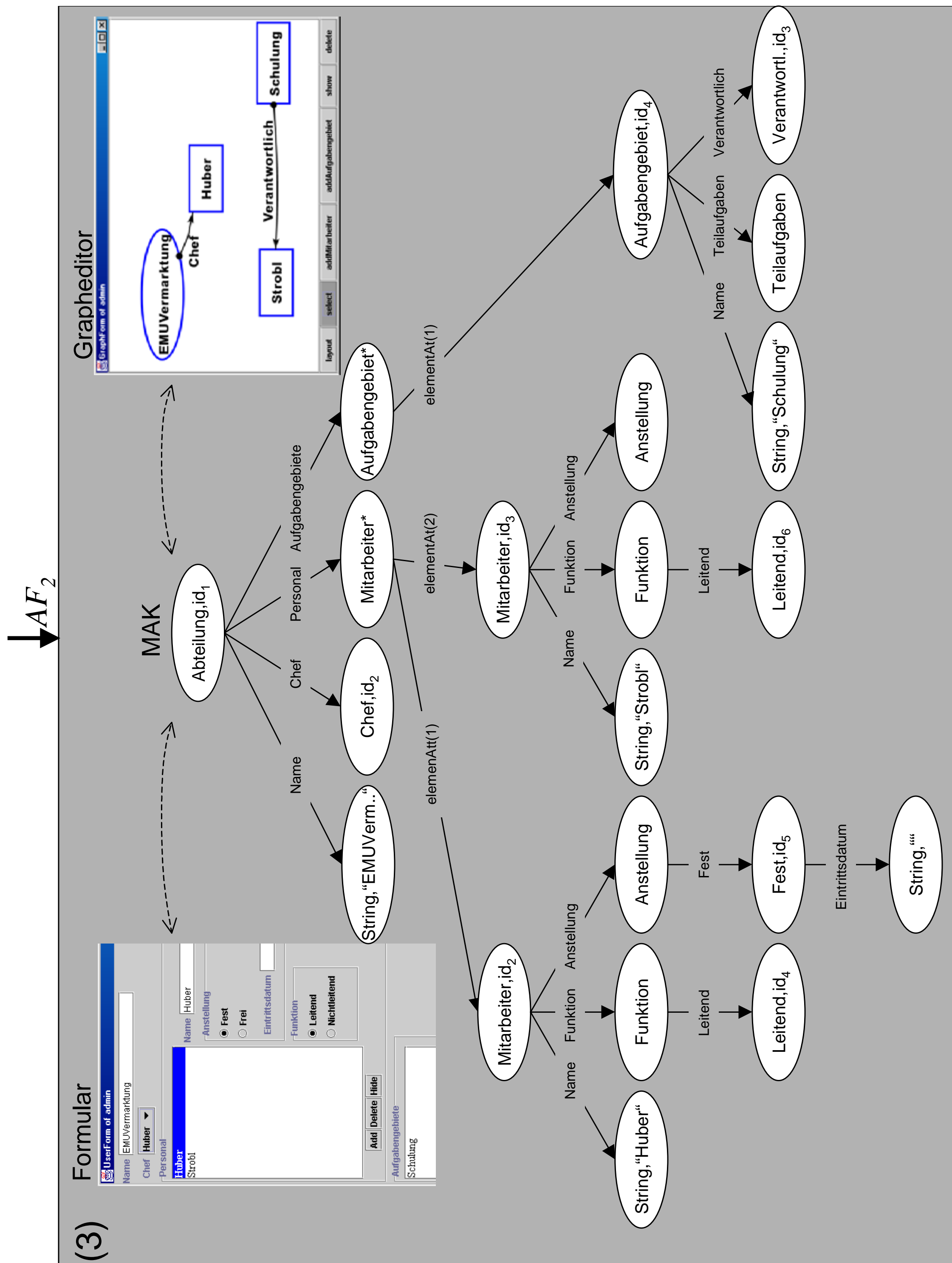


**Abbildung 4-3: Ausführung eines EMU-Systems vom Typ *Abteilung* (2)**

### 4.2.1 Abstrakte Ausführung

Die abstrakte Ausführung abstrahiert von der konkreten Darstellung im Formular und im Grapheditor und beschreibt lediglich die Elemente des zugehörigen Zustandsübergangssystems.

**Zustände**

Als Zustandsbegriff der abstrakten Ausführung dient der in Abbildung 4-2 und Abbildung 4-3 jeweils mittig dargestellte MAK. Der Zustand wird also durch einen markierten Baum gebildet. Die Knoten des MAKs sind abhängig von ihrem Metatyp wie folgt markiert (die Beispiele beziehen sich jeweils auf Abbildung 4-2 und Abbildung 4-3):

- Tupelknoten: Typname (z.B. *Abteilung*) und Identifikator (z.B. $id_1$)
- Varianten- und Listenknoten: Typname (z.B. *Mitarbeiter**)
- Referenzknoten: Typname und Identifikator des referierten Tupelknotens oder $\perp$, wenn kein Tupelknoten referiert wird (z.B. $(Chef, id_2)$ im Zustand (3) bzw. $(Chef, \perp)$ im Zustand (1))
- Basisknoten: Typname und Wert (z.B. *(String,“EMUVermarktung“)*)

Der Wurzelknoten ist mit dem Starttyp $start_{DM}$ des zugehörigen Datenmodells *DM* (*Abteilung*) markiert. Die Markierung aller weiteren Knoten erfolgt konform mit den Produktionen des Datenmodells.

Die Identifikatoren der Tupelknoten (nicht zu verwechseln mit ihren Selektoren) werden vom EMU-System zur Laufzeit intern vergeben und verwaltet.

Wie üblich verwenden wir das Symbol $\perp$ als Bezeichnung für „undefiniert“ und modellieren damit einen Wert, der zur Laufzeit vom Benutzer noch nicht eingegeben wurde.

Die Verwendung von Identifikatoren erscheint an dieser Stelle evtl. recht implementierungsnah. Da wir jedoch hier und im formalen Abschnitt klären wollen, wie die Konstruktion konkret ausführbarer Systeme erfolgt, ist ihre Verwendung unerlässlich, weil die Identifikatoren etwa dafür notwendig sind, die Darstellung der logischen Knoten des MAK in den konkreten Benutzungsoberflächen zu verwalten (vgl. *Layoutattribute* in Abschnitt 3.2.3). Beispielsweise werden bestimmten Tupelknoten des abstrakten Zustands (z.B. dem Wurzelknoten im Initialzustand) graphische Knoten im Grapheditor zugeordnet.

Im Initialzustand wird etwa im Grapheditor ein durch ein Oval präsentierter graphischer Knoten dargestellt, der dem (logischen) Wurzelknoten zugeordnet ist. Dieser graphische Knoten ist mit „no ident“ markiert, weil noch kein definierter Wert für das Attribut *GraphNodeValue* vorliegt (vgl. Abschnitt 4.2.3). Zur graphischen Darstellung sind noch eine Reihe weiterer Attributwerte notwendig, wie beispielsweise die X- und Y-Koordinaten der graphischen Knoten im Grapheditor.

Die technisch sinnvollste Variante für die Implementierung der Zuordnung besteht in der Verwendung von Layouttabellen, in der für jeden Identifikator die Werte seiner Layoutattribute stehen.

Als Alternative für die Identifikatoren wären auch die Baumpositionen der logischen Knoten denkbar (vgl. [PH94]). Jedoch kann sich die Baumposition eines Knotens zur Laufzeit ändern, etwa durch das Löschen von Listenelementen. Würde man beispielsweise im Zustand (3) von Abbildung 4-3 den Mitarbeiter mit dem Identifikator $id_2$ löschen, so wäre es für den Benutzer unzumutbar, wenn der graphische Knoten, der vorher $id_2$ angezeigt hat, nun $id_3$ anzeigen würde.

Es wäre auch möglich, die Layouttabellen bei Änderungen von Positionen zu aktualisieren, was jedoch unnötig ineffizient und damit, im Hinblick auf die Echtzeitfähigkeit praktisch einsetzbarer interaktiver Systeme, nicht annehmbar wäre.

Die Verwendung von Identifikatoren bringt aber auch bereits bei der Betrachtung der abstrakten Ausführung den Vorteil, dass damit wesentlich klarer die Semantik der Referenzen und deren interaktive Bearbeitung definiert werden kann, als dies durch Verwaltung von (Baum-)Positionen als Referenzwerte möglich wäre.

**Initialzustand**

Der Initialzustand eines EMU-Systems vom Typ *Abteilung* wird in Abbildung 4-2 dargestellt. Er entsteht durch Anwendung einer in Abschnitt 4.3 beschriebenen, durch das Datenmodell implizit festgelegten Funktion $create_{Abteilung}(..)$. Durch diese Funktionsanwendung erfolgt die Konstruktion der dargestellten Struktur und die Vergabe des Identifikators $id_1$, der einen Teil der Markierung des Wurzelknotens bildet.

**Aktionen**

Im Initialzustand des EMU-Systems ((1) in Abbildung 4-2) wird eine Folge von Aktionen ausgeführt, die eine Zustandsänderung bewirken. Da EMU-Systeme deterministisch sind (d.h. Zustand und Aktion ergeben genau einen Folgezustand), ist eine Erweiterung des Übergangsbegriffs von einzelnen Aktionen auf Aktionsfolgen eindeutig festgelegt.

Wie beispielsweise die erste Aktion der ersten Aktionsfolge $add(\langle 3 \rangle, 1)$ zeigt, werden Aktionen durch Terme dargestellt, die aus

- einem Aktionsbezeichner (hier: *add*, was das Einfügen eines neuen Listenelements bedeutet),
- einer in Dewey-Notation [Gan78] codierten Baumposition (hier: $\langle 3 \rangle$, was den Knoten für das Personal der Abteilung bezeichnet) und
- einem Aktionsargument (hier: 1, was den Index, an der in die Liste eingefügt werden soll, beschreibt).

Diese syntaktische Form gilt für alle Aktionen. Jedoch unterscheiden sich die bei den verschiedenen Knoten möglichen Aktionen nach dem Metatyp des Knotens:

- Tupelknoten: keine impliziten Aktionen;
- Variantenknoten: Umschalten der aktuell ausgeprägten Alternative (*setAlt*); der Name der Alternative ist Argument der Aktion;
- Listenknoten: Hinzufügen (*add*) und Löschen (*remove*) von Listenelementen; die Position, an der eingefügt werden soll ist Argument der Aktion;

- Referenzknoten: Ändern des aktuell referierten Knotens (*setRef*); die Position des neuen Knotens ist Argument der Aktion;
- Basistypen: Setzen des Wertes (*setValue*); der neue Wert ist Argument der Aktion;

Man beachte, dass diese Darstellung der Aktionen nur als Vorbereitung für die nachfolgende formale Darstellung der Ausführung von EMU-Systemen dient und nicht etwa der Entwickler verstehen muss, der mit EMUGEN später Systementwicklung betreibt. D.h. es handelt sich nur um eine interne Darstellung, die nicht nach außen sichtbar ist.

Bis auf die Interaktionen der Referenzknoten entsprechen diese Aktionen denen, die auch beim BOSS-System [Sch97] zur Bearbeitung eines Elements einer komplexen Sorte verwendet werden. Die Aktion *setRef* an einem Referenzknoten entspricht dem „Zeigerverbiegen" in prozeduralen Programmiersprachen. Hier wird sie jedoch unmittelbar durch den Benutzer interaktiv ausgeführt.

**Zustandsübergang**

Aktionen können entweder eine lokale Änderung von Werten an Knoten oder eine strukturelle Änderung des MAKs hervorrufen. Eine strukturelle Änderung des Zustands ergibt sich bei den Aktionen auf Varianten- und Listenknoten, während die Aktionen auf Referenz- und Basisknoten lediglich den Wert des durch die Aktion angesprochenen Knotens ändern.

Man beachte, dass die Listeneinfügeoperation aus Sicht des Benutzers ohne Argument abläuft. D.h. der Benutzer fügt in einem atomaren Schritt (konkret: durch Betätigen des im Formular dargestellten Buttons oder durch Klicken im Graphenditor im entsprechenden Einfügemodus) einen neuen Knoten als weiteren Nachfolger einer Liste ein, und kann anschließend die Komponenten des neuen Knotens definieren.

Eine alternative (bzw. sogar die in den meisten vergleichbaren praktischen Anwendungen angewandte) Vorgehensweise besteht darin, die Einfügeoperation in einem Dialog abzuwickeln. Dabei würde der Benutzer zuerst die Einfügeoperation anstoßen, dann die Argumente eingeben (währenddessen die Einfügeoperation auch abgebrochen werden kann) und schließlich die Einfügeoperation abschließen. Entsprechende Dialoge können im Aufgaben- und Benutzermodell definiert werden.

Der Grund dafür, dass EMU von dieser Standard-Vorgehensweise abweicht, liegt in dem damit in Verbindung stehenden, allgemeinen Konzept für eine synchrone Bearbeitung der Knoten des MAKs, die an die Funktionsweise von MVC-Architekturen angelehnt ist.

Wenn beispielsweise eine Neuaufnahme eines Mitarbeiters erfolgt, so hat der Benutzer die Möglichkeit die Komponenten des Mitarbeiters – etwa über ein Formular – einzugeben. Eine konsequente Implementierung der Funktionsweise einer MVC-Architektur bedeutet jedoch, dass ein Modell, nämlich ein neuer Teilbaum, der den neuen Mitarbeiter modelliert, bereits vorhanden sein muss.

Daher erfolgt die Konstruktion von Knoten und den dazugehörigen Teilbäumen (im Sinne von MVC das Modell) durch das EMU-System sofort beim „Beginn" einer Neuaufnahme und nicht etwa erst, wenn die Teilkomponenten eingegeben wurden. Letzteres entspräche der funktionalen Sichtweise, bei der ein Element aus der strikten Funktionsanwendung einer entsprechenden Konstruktorfunktion gewonnen wird.

Abbildung 4-4 zeigt die Konstruktion eines Mitarbeiterknotens und seines zugehörigen Teilbaums infolge einer add-Aktion auf den Mitarbeitenlistenknoten mit der Position *p*.

Diese Konstruktion von Knoten und Teilbäumen erfolgt nicht nur bei Zustandsübergängen infolge von *add*-Aktionen auf Listenknoten sondern auch infolge von *setAlt*-Aktionen auf Variantenknoten.

Bei der Konstruktion von Knoten und Teilbäumen werden gegebenenfalls Identifikatoren für Tupelknoten vergeben (vgl. den Identifikator *id* in Abbildung 4-4).

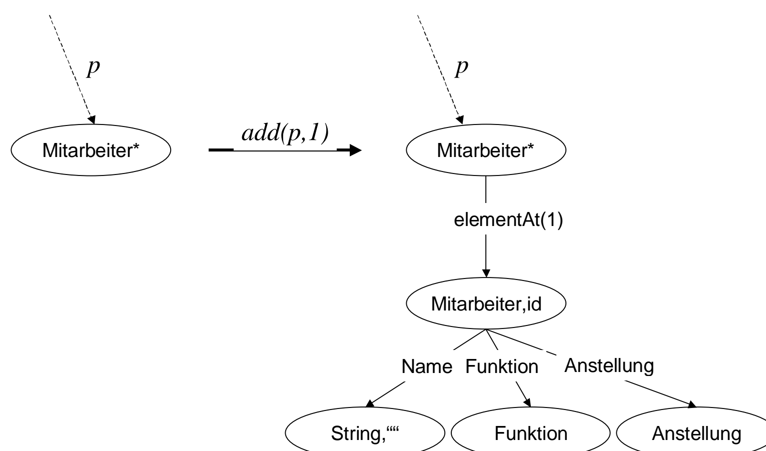


**Abbildung 4-4: Erzeugung eines neuen Listenelements**

Eine weitere Auswirkung der konsequenten Anwendung der MVC-Architektur auf das Konzept der EMU-Systeme besteht in der expliziten Verwaltung von Variantenknoten als Teilknoten des MAK. Variantenknoten können gleichzeitig in mehreren Formularen (die evtl. von verschiedenen Benutzer bearbeitet werden) dargestellt werden, wobei sich bei einer Änderung der Ausprägung einer Variante nach einer *setAlt*-Aktion auch die Darstellung in allen Formularen entsprechend ändern muss. Ferner dient der Variantenknoten als Bezugspunkt für die *setAlt*-Aktionen (vgl. das Formular in Abbildung 4-3). Als Folge dieser Vorgehensweise kann statisch jeder Position (codiert in der Dewey-Notation) im MAK ein Typ zugeordnet werden.

Bei einem Zustandsübergang infolge einer *setRef*-Aktion auf einen Referenzknoten erfolgt lediglich die Änderung der Markierung des angesprochenen Referenzknotens. Beispielsweise erfolgt beim Zustandsübergang infolge der Aktion $setRef(\langle 2\rangle,\langle 3,1\rangle)$ die Belegung des Knotens an der Position $\langle 2\rangle$ mit dem Identifikator $id_2$, welcher den Identifikator des Tupelknotens vom Typ *Mitarbeiter* an der Position $\langle 3,1\rangle$ darstellt. Diese Aktion bedeutet, dass als Chef der Abteilung (Position $\langle 2\rangle$) der Mitarbeiter an Position $\langle 3,1\rangle$ gekennzeichnet wird.

Es sollte nun ein grundlegendes Verständnis für die abstrakte Ausführung eines EMU-Systems vorhanden sein. Ferner sollte durch die Abbildung 4-2 und Abbildung 4-3 deutlich werden, wie durch ein EMU-Datenmodell ein interaktives Informationssystems definiert wird.

Die konkrete interaktive Ausführung dieser Informationssysteme am Bildschirm erfolgt durch Formulare und Grapheditoren, die dem Benutzer den abstrakten Zustand anzeigen und ihm die Möglichkeit geben, die implizit definierten Aktionen auszuführen.

### 4.2.2 Ausführung mit Formularen

Die Generierung der Formulare aus dem Datenmodell erfolgt durch die EMUGEN-Teilkomponente FORMGEN [BK99], die sich bei der Erstellung des Formularlayouts weitgehend an die im BOSS-System [Sch97] formal festgelegte Layoutrichtlinie $l_1$ hält. Wir wollen sie hier durch Standardlayoutregeln, abhängig vom Metatyp des im Formular zu bearbeitenden Knotens, informell beschreiben. Dabei wird auch besprochen, welche Anpassungen des Standardformularlayouts im Layoutmodell möglich sind.

#### Standardlayout von Formularen für Tupelknoten

Formulare zum Anzeigen und Bearbeiten von Tupelknoten bestehen aus Teilformularen für jede einzelne Komponente. Sofern genügend Platz (am Bildschirm) vorhanden ist, werden alle Teilformulare gleichzeitig dargestellt (vgl. Abbildung 4-2), ansonsten wird eine geeignete Navigationsmöglichkeit zur Verfügung gestellt, die es ermöglicht, zwischen den Teilformularen umzuschalten (vgl. Abbildung 5-9).

#### Layoutanpassung von Formularen für Tupelknoten

Der Entwickler kann im Layoutmodell die Anordnung der Teilformulare für die Tupelkomponenten festlegen. Dies kann für alle oder einzelne Tupeltypen erfolgen. Beispielsweise kann angegeben werden, ob die Komponenten bevorzugt horizontal, vertikal oder über eine Navigationsmöglichkeit dargestellt werden sollen. Bei einzelnen Tupeltypen kann ferner das Layout über Boxen (wie bei BOSS [Sch97]) angegeben werden.

#### Standardlayout von Formularen für Variantenknoten

Formulare zum Anzeigen und Bearbeiten von Variantenknoten bestehen aus einer Gruppe von Interaktionsobjekten, welche die Ausführung der zugehörigen *setAlt*-Aktionen ermöglichen, und einem Teilformular zur Anzeige der ausgewählten Alternative. Wenn genügend Platz vorhanden ist, so können die *setAlt*-Aktionen über *Radiobuttons* (vgl. Abbildung 4-6) ausgeführt werden, ansonsten kann auch dazu eine *Drop-Down-Liste* (auch als *Combobox* bekannt, vgl. Abbildung 4-5) verwendet werden.

**Abbildung 4-5: Ausführung von *setAlt*-Aktionen über eine Drop-Down-Liste**

Aufgrund der dynamischen Struktur des MAKs müssen auch die Formulare dynamisch sein. Diese Dynamik ist vor allem bei den (Teil-)Formularen für Variantenknoten von Bedeutung. Beispielsweise wird in Abbildung 4-5 ein Teilformular für einen Variantenknoten vom Typ *Anstellung* als Teil des Formulars des selektierten Mitarbeiters dargestellt (Position des Variantenknotens: $\langle 3,1,3 \rangle$ ), dessen aktuelle Ausprägung *Fest* ist. Daher wird in diesem Formularteil das Eintrittsdatum dargestellt (vgl. das Datenmodell aus Beispiel 4-1: *Eintrittsdatum* ist Komponente des Tupeltyps *Fest*). Würde nun die Aktion $setAlt(\langle 3,1,3 \rangle, Frei)$ ausgeführt, so würden im Nachfolgezustand im Formularteil des

angesprochenen Variantenknotens die Formularteile für die Komponenten des Typs *Frei*, nämlich *Vertragsbeginn* und *Vertragsende* erscheinen. Das Teilformular zu diesem Nachfolgezustand ist in Abbildung 4-6 dargestellt.

**Abbildung 4-6: Formularteil für einen Variantenknoten vom Typ *Anstellung* mit der aktuellen Ausprägung *Frei***

### Layoutanpassung von Formularen für Variantenknoten

Der Entwickler kann im Layoutmodell die relative Anordnung der beiden Teilformulare für die Gruppe der Radiobuttons bzw. der Drop-Down-Liste und der aktuellen Alternative festlegen. Ferner kann angegeben werden, ob Radiobuttons oder eine Drop-Down-Liste verwendet werden sollen.

### Standardlayout von Formularen für Listenknoten

Formulare zum Anzeigen und Bearbeiten von Listenknoten zeigen den Listeninhalt an und bieten die Möglichkeit, die zugehörige *add*- und *remove*-Aktion auszuführen. Das notwendige Argument (der Index, an dem eingefügt wird, bzw. dessen Element gelöscht werden soll) gibt der Benutzer durch Selektion des entsprechenden Elements an.

Wenn genügend Platz vorhanden ist, können als Interaktionsobjekte zum Ausführen der *add*- und *remove*-Aktion Buttons verwendet werden (vgl. Abbildung 4-3), ansonsten können die Aktionen auch über ein Menü oder über Funktionstasten dargestellt werden.

Man beachte, dass in dem Teilformular für das Personal in Abbildung 4-3 die Namen der Mitarbeiter in der Liste angezeigt werden, ohne dass dies beim Generierungsprozess explizit festgelegt wurde. Mit denselben Namen sind auch die entsprechenden graphischen Knoten des Grapheditors markiert.

Es gibt also eine Reihe von Standardlayoutregeln für die Benennung der Knoten des MAKs in den Formularen und Grapheditoren. Diese können vom Entwickler im Layoutmodell überschrieben werden, beispielsweise um in der Liste des Formulars nicht nur die Namen, sondern auch die Art der Anstellung zu visualisieren. Die implizite Festlegung, die im Zusammenhang mit der Darstellung der Listenelemente in der Liste im Formular zu tragen kommt, betrifft das Layoutattribut *FormListElementValue*, welches als synthetisches Attribut (konzeptionell) an allen MAK-Knoten vorkommt. Die implizit festgelegte Berechnung von *FormListElementValue* an einem Knoten $p$ kann semiformal[4] wie folgt definiert werden:

[4] Wir sprechen von *semiformalen Definitionen*, wenn die Definition auf informellen oder noch nicht definierten Beschreibungen basiert, wobei wir gegebenenfalls auch auf eine vollständige Indizierung zugunsten einer besseren Lesbarkeit verzichten. Hier müsste etwa die Funktion *value* korrekterweise mit dem aktuellen Zustand indiziert werden.

$$FormListLayoutValue(p) = \begin{cases} type(p) & \text{wenn } p \circ \langle 1 \rangle \text{ nicht ex.} \\ value(p \circ \langle 1 \rangle) & \text{wenn } p \circ \langle 1 \rangle \text{ Basisknoten} \\ FormListLayoutValue(p \circ \langle 1 \rangle) & \text{sonst} \end{cases}$$

Die Funktionen *type* und *value* liefern hier den Typ bzw. den Wert eines Knotens. Die Berechnung der Darstellung des Knotens der Position $\langle 3,1 \rangle$ in Abbildung 4-3 ergibt beispielsweise $FormListLayoutValue(\langle 3,1 \rangle) = value(\langle 3,1,1 \rangle) = "Huber"$.

Als generierte Standard-Navigationsmöglichkeit erhält der Benutzer die Möglichkeit, mit Hilfe eines Teilformulars das selektierte Element der Liste vollständig einzusehen und zu bearbeiten (vgl. Abbildung 4-3).

**Layoutanpassung von Formularen für Listenknoten**

Der Entwickler kann im Layoutmodell die relative Anordnung der beiden Teilformulare für die Liste und die Gruppe der Buttons für die *add*- und *remove*-Aktion und die Aktion für die implizite Navigationsmöglichkeit festlegen. Ferner können das Layout der einzelnen Listenelemente und alternative Interaktionsobjekte (etwa Menüs, Funktionstasten) für die Aktionen festgelegt werden.

**Standardlayout von Formularen für Referenzknoten**

Formulare zum Anzeigen und Bearbeiten von Referenzknoten zeigen den aktuellen Wert des entsprechenden Referenzknotens an und bieten die Möglichkeit, die *setRef*-Aktion auszuführen. Um den Argumentwert – die Position des neu referierten Knotens – einzugeben, bietet sich die Verwendung einer Drop-Down-Liste an, die jeweils genau die Referenzkandidaten im aktuellen Zustand anzeigt (vgl. Abbildung 4-7).

Links in Abbildung 4-7 wird ein Teilformular gezeigt, bei dem ein Benutzer die Referenzkandidaten für den Chef betrachtet und damit – formal gesehen – das Argument für die *setRef*-Aktion festlegt. Diese Liste der Referenzkandidaten ist in diesem Fall einelementig, weil im betrachteten Zustand (vgl. Abbildung 4-3) nur ein Mitarbeiter sowohl leitend als auch festangestellt ist.

Rechts in Abbildung 4-7 wird ein Teilformular gezeigt, bei dem ein Benutzer die Referenzkandidaten für den Verantwortlichen eines Aufgabengebiets betrachtet. Hier besteht die Liste aus beiden Mitarbeitern, weil beide eine leitende Funktion besitzen.

Der Wert des Referenzselektors (genauer: der durch den Selektor identifizierten Komponente) wird beim Formular für einen Referenzwert angezeigt. Damit sollte die Notwendigkeit eines Referenzselektors (vgl. Abschnitt 4.1.4) nun klar sein.

Ähnlich wie bei der Darstellung der Listenelemente ist dabei durch eine Standardlayoutregel festgelegt, welcher Wert des Referenzselektors in der Drop-Down-Liste angezeigt wird. Bei den bisher betrachteten Beispielen erscheint dies trivial, weil hier die Typen der Referenzselektoren jeweils Basistypen sind.

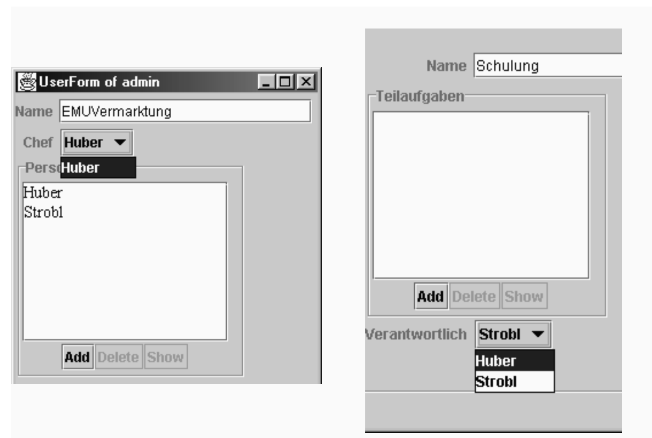


**Abbildung 4-7: Teilformulare für Referenzknoten zeigen Referenzkandidaten**

Allgemein kann jedoch ein Referenzselektor auch ein komplexer Typ sein, weshalb die Verwendung eines zusätzlichen Layoutattributs (*FormRefSelValue)* notwendig ist, dessen Wert in der Drop-Down-Liste angezeigt wird. *FormRefSelValue* wird implizit auf die gleiche Weise berechnet wie *FormListElementValue*.

**Layoutanpassung von Formularen für Referenzknoten**

Eine Anpassung der Formulare für die Referenzknoten ist derzeit nicht vorgesehen.

**Standardlayout von Formularen für Basisknoten**

Formulare zum Anzeigen und Bearbeiten von Basisknoten zeigen den aktuellen Wert an und bieten die Möglichkeit, einen neuen Wert einzugeben, was der Ausführung der *setValue*-Aktion entspricht.

**Layoutanpassung von Formularen für Basisknoten**

Die Formulare für Basisknoten sind vordefiniert und daher nicht im Layoutmodell anzupassen. Jedoch ist die Menge der Basistypen nicht beschränkt. D.h. der Entwickler kann selbst Basistypen implementieren, wozu auch die Implementierung entsprechender Formulare gehört. Ebenso kann der Entwickler die Formulare für die bereits eingebauten Basistypen in der entsprechenden Zielsprache alternativ implementieren und somit an seine Bedürfnisse anpassen.

### 4.2.3 Ausführung mit Grapheditoren

Der MAK eines EMU-Systems kann grundsätzlich sowohl mit einem Formular als auch mit einem Grapheditor bearbeitet werden. Ähnlich wie bei der Generierung von Formularen werden auch bei der Generierung von Grapheditoren eine Reihe von Standardlayoutregeln, etwa bzgl. des Layouts von graphischen Knoten, angewandt.

Wiederum kann der Entwickler diese impliziten Regeln überschreiben und somit eine Anpassung an die gewünschten Gegebenheiten erreichen. Die (derzeitigen) Möglichkeiten zum Überschreiben der Regeln sind durch die Spezifikationskonzepte für Grapheditoren von GRACE [Kle99] geprägt, weil die Generierung der Grapheditoren mit diesem Werkzeug erfolgt (vgl. Abschnitt 3.3).

Bevor diese Spezifikationskonzepte betrachtet werden, soll zunächst genau festgelegt werden, welche Teile des MAKs durch den automatisch generierten Grapheditor visualisiert werden.

**Grundlegende Funktionalität**

Der Grapheditor zu einem EMU-System visualisiert (bestimmte) Tupelknoten eines gegebenen MAKs durch graphische Knoten (*GNodes*) und (alle) Referenzknoten durch graphische Kanten. Die Teilmenge der visualisierten Tupelknoten besteht aus der disjunkten Vereinigung der *statischen* (*StaticGNodes*) und der *dynamischen* Knoten (*DynamicGNodes*). Die statischen Knoten sind diejenigen Tupelknoten, die direkte Komponenten des Wurzelknotens darstellen, sowie der Wurzelknoten selbst. Die dynamischen Knoten sind die Tupelknoten, die als Element einer Listenkomponente der Wurzelkomponente vorliegt. Die Menge der graphischen Knoten und Kanten eines Grapheditors für einen gegebenen Mehrbenutzer-Applikationsschnittstellenkern MAK kann semiformal wie folgt beschrieben werden:

$$
\begin{aligned}
&GNodes_{MAK} =_{def} StaticGNodes_{MAK} \uplus DynamicGNodes_{MAK}\\
&StaticGNodes_{MAK} =_{def} value(\langle\rangle) \uplus \{value(\langle i\rangle) \mid \langle i\rangle \text{ ist Tupelknoten}\}\\
&DynamicGNodes_{MAK} =_{def} \{value(\langle i, j\rangle) \mid \langle i, j\rangle \in \text{MAK},\\
&\qquad\qquad \langle i\rangle \text{ ist Listenknoten},\\
&\qquad\qquad \langle i, j\rangle \text{ ist Tupelknoten}\\
&\qquad \}\\
&GEdges =_{def} \{(s, p, t) \mid s, t \in GNodes, p \in \text{MAK},\\
&\qquad\qquad p \text{ ist Referenzknoten},\\
&\qquad\qquad value(p) = t,\\
&\qquad\qquad \exists q, r : p = q \circ r, value(q) = s\\
&\qquad \}
\end{aligned}
$$

Die graphischen Knoten und Kanten des in Abbildung 4-3 gezeigten Grapheditors ergeben sich zu

$$
\begin{aligned}
GNodes_1 &= \{id_1, id_2, id_3, id_4\}\\
GEdges_1 &= \{(id_1, \langle 2\rangle, id_2), (id_4, \langle 4,1,3\rangle, id_3)\}
\end{aligned}
$$

Wir nennen im Folgenden die Elemente von *GNodes* und *GEdges* auch ***Graphelemente***.

Als Aktionen auf den MAK ermöglicht ein Grapheditor das Erzeugen und Löschen dynamischer Knoten (aus dem MAK) sowie das Löschen von Referenzknoten (d.h. der entsprechende Referenzwert wird auf $\perp$ gesetzt).

Ähnlich wie ein Listenformular ermöglicht der Grapheditor mit Hilfe eines Formulars den selektierten graphischen Knoten einzusehen und zu bearbeiten.

Zur Spezifikation eines Grapheditors mit GRACE sind die darzustellenden ***graphischen Knoten- und Kantentypen*** von zentraler Bedeutung. Wir definieren sie semi-

formal, abhängig von der Menge *IDT* von Interaktionsdatentypen des Datenmodells, wie folgt:

$$GNodeType =_{def} \{X \in IDT) \mid \exists \mathrm{MAK}: \exists p \in \mathrm{MAK}: value(p) \in GNodes_{\mathrm{MAK}}, type(p) = X \}$$

$$GEdgeType =_{def} \{X \in IDT) \mid X \text{ ist Referenz}\}$$

**Standard-Dialogkontrolle von Graphelementen**

Die Benutzung eines Grapheditors mit *n* Graphelementen kann abstrakt als die nebenläufige Ausführung von *n* (meist sehr einfachen) Dialogen betrachtet werden. Damit ist beispielsweise gemeint, dass während eines Ausführungszustands einige graphische Knoten oder Kanten selektiert sind oder momentan verschoben werden.

In GRACE kann jedem graphischen Knoten- und Kantentyp eine Dialogkontrolle zugeordnet werden. Dabei ist es möglich, eine eigene Dialogkontrolle durch einen Automaten zu definieren oder einen bereits in der GRACE-Spezifikationsbibliothek vorhandenen auszuwählen. Durch die Dialogkontrolle wird beispielsweise festgelegt, dass der Benutzer ein Graphelement selektieren und damit – abstrakt gesehen – den Zustand des zugehörigen Automaten verändern kann. Durch die Dialogkontrolle wird jedoch noch nicht festgelegt, durch welche atomare Benutzerinteraktion – etwa durch einfachen oder doppelten Mausklick – eine Selektion vorgenommen wird.

Implizit wird bei der automatischen Generierung eines Grapheditors für jeden graphischen Knoten- und Kantentyp als Dialogkontrolle der Automat *Selectable* [Kle99] verwendet, der einen Bestandteil der GRACE-Spezifikationsbibliothek bildet. Dieser Automat kennt genau zwei Zustände (*not_selected* (Initialzustand), *selected*) und zwei Eingabetoken (*select, deselect*).

**Anpassung der Dialogkontrolle von Graphelementen**

Der Benutzer kann im EMU-Layoutmodell für bestimmte graphische Knoten- und Kantentypen einen alternativen Automaten angeben bzw. neu definieren. Da die GRACE-Eingabekomponenten Vererbung erlauben, ist es auch möglich, das Verhalten der vorhandenen Automaten zu verfeinern (für Details siehe [Kle99]). Obwohl es sich dabei strenggenommen um die Festlegung der Dialogkontrolle handelt, ordnen wir die Definition der Automaten für die Graphelemente dem Layoutmodell zu, weil es sich dabei um die Definition der Funktionsweise eines anwendungsspezifischen Interaktionsobjekts (vergleichbar mit der Funktionsweise eines Buttons) handelt, welche für Zustandsübergänge auf Layoutebene [Sch97, Eic98] zuständig ist.

**Standardlayout von Graphelementen**

Wie in Abbildung 4-2 ersichtlich, werden die graphischen Knoten durch geometrische Formen wie Ovale oder Rechtecke und die graphischen Kanten durch Kurven mit Pfeilen dargestellt. Nicht in der Abbildung ersichtlich ist die Art und Weise, wie etwa die Selektion eines graphischen Elements erfolgt, nämlich durch einfachen Mausklick.

Zur Spezifikation dieser Aspekte ermöglicht GRACE für jeden graphischen Knoten- und Kantentyp die Zuordnung eines Zustands des zugehörigen Automaten zu einer geometrischen Form (z.B. ein Rechteck) und einem sogenannten *Interaktor*. Durch einen Interaktor wird spezifiziert, wie durch eine Folge atomarer (quasi „physikalischer") Benutzerinteraktionen (wie etwa Mausklicks und Mausbewegungen) ein Token für den zugehörigen Automaten der Dialogkontrolle des Graphelements gebildet wird. Die Aufgabe des Interaktors ist damit mit der eines Scanners für eine Programmiersprache zu vergleichen [WM97].

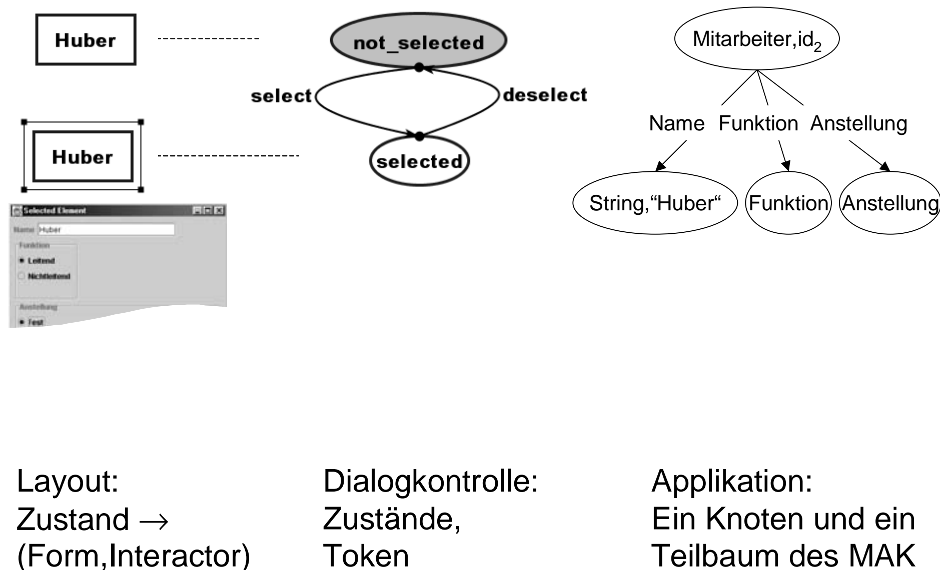


**Abbildung 4-8: Graphelemente als BOs von MAK-Knoten (vgl. Seeheim-Modell)**

Abbildung 4-8 zeigt die an das Seeheim-Modell angelehnte Struktur der Bearbeitung von MAK-Knoten durch Graphelemente eines Grapheditors. Die Applikation besteht hier aus einem Teilbaum des MAKs. Die Dialogkontrolle wird durch den bereits beschriebenen Automaten *Selectable* aus der GRACE-Spezifikationsbibliothek [Kle99]) gebildet. Der Anfangszustand ist grau markiert. Im Layout wird jedem Zustand der Dialogkontrolle eine geometrische Form und ein Interaktor zugeordnet. Im Zustand *not_selected* der Dialogkontrolle wird die geometrische Form durch ein Rechteck zusammen mit einer Beschriftung gebildet. Diese Beschriftung zeigt – wie der Inhalt einer Formularliste – den Namen des Mitarbeiters. Dafür verantwortlich ist das Attribut *GraphNodeValue*, dessen Berechnung analog zu der von *FormListElementValue* erfolgt.

Dem Zustand *selected* der Dialogkontrolle werden im Layout zwei konzentrisch gezeichnete Rechtecke und die Beschriftung zugeordnet. Ferner wird – wenn insgesamt im Grapheditor genau ein Knoten selektiert ist – das zum MAK gehörige Formular am Bildschirm angezeigt und kann bearbeitet werden. Um den graphischen Knoten für das Wurzelelement besonders zu markieren, wird dafür anstelle eines Rechtecks implizit ein Oval verwendet (vgl. Abbildung 4-2).

Im Interaktor ist definiert, dass der Benutzer den Selektionszustand durch einen einfachen Mausklick auf das Graphelement ändern kann. Abstrakt betrachtet, erstellt der Interaktor ein Token (je nach Selektionzustand *select* oder *deselect*) und sendet es der Dialogkontrolle, welche abhängig vom Token und vom aktuellen Zustand einen Zustandsübergang durchführt.

Die geometrische Form, die einer graphischen Kante $(s, p \circ i, t) \in GEdges$ zugeordnet wird (vgl. Abbildung 4-3), besteht aus einer beschrifteten Linie, welche die graphischen Knoten *s* und *t* verbindet. Dabei wird die Verbindungsstelle bei *s* mit einem Punkt und die Verbindungsstelle bei *t* mit einem Pfeil markiert. Die Beschriftung der Linie erfolgt abhängig von der an *p* angewandten Produktion $prod(p)$ durch das Attribut *GraphEdgeValue*:

$$GraphEdgeValue(p \circ i) = \begin{cases} S_i & \text{wenn } prod(p) = \mathtt{T::=T_1:S_1 \ldots T_i:S_i \ldots T_n:S_n} \\ A_i & \text{wenn } prod(p) = \mathtt{V::=A_1 | \ldots | A_i | \ldots | A_n} \\ i & \text{wenn } prod(p) = \mathtt{L::=E*} \end{cases}$$

### Layoutanpassung von Graphelementen

Die Zuordnung der geometrischen Formen und Interaktoren zu den Dialogkontrollzuständen kann entsprechend den Möglichkeiten der GRACE-Eingabespezifikationssprache angepasst werden. Für deren vollständige Betrachtung sei auf [Kle99] verwiesen. Wir wollen an dieser Stelle nur einen Eindruck vermitteln und generieren dazu einen Grapheditor für (die Diagrammsprache, vgl. Abschnitt 3.4.2) *Petrinetze*.

**Beispiel 4-2: *Petrinetze***

> *Es soll ein Grapheditor zur Eingabe und Bearbeitung einfacher, binärer Petrinetze generiert werden.*

Das zugehörige EMU-Datenmodell kann wie folgt formuliert werden.

```
PetriNetz::=Stelle*:Stellen Transition*:Transistionen
Stelle::=String:Name Marke
Marke::= Nichtvorhanden | Vorhanden
Transition::=String:Name Vorbereich* Nachbereich*
Vorbereich::=Name->Stelle
Nachbereich::=Name->Stelle
```

Ein Petrinetz besteht aus je einer Liste von Stellen und Transitionen. Jede Stelle hat einen Namen und eine Marke, die vorhanden oder nicht vorhanden sein kann. Eine Transition hat ebenfalls einen Namen und einen Vor- und einen Nachbereich, die jeweils als Liste von Referenzen auf Stellen modelliert werden.

Aus dem obigen Datenmodell wird automatisch ein Grapheditor generiert, mit dem das in Abbildung 4-9 gezeigte Petrinetz definiert werden kann. Es besteht aus zwei Stellen und einer Transition, welche die beiden Phasen miteinander verbindet. Die Darstellung der Stellen und Transitionen erfolgt gemäß den Standardlayoutregeln. Offensichtlich ist diese Darstellung (aus ergonomischen Gründen) nicht zur Bearbeitung von Petrinetzen geeignet, weil sowohl Stellen als auch Transitionen mit Rechtecken dargestellt sind und aus der Darstellung der graphischen Kanten für die Referenzen nicht hervorgeht, ob es

sich um eine Vorbereichs- oder Nachbereichsreferenz handelt. Weiterhin stört die unübliche Markierung der graphischen Kanten mit dem Index.

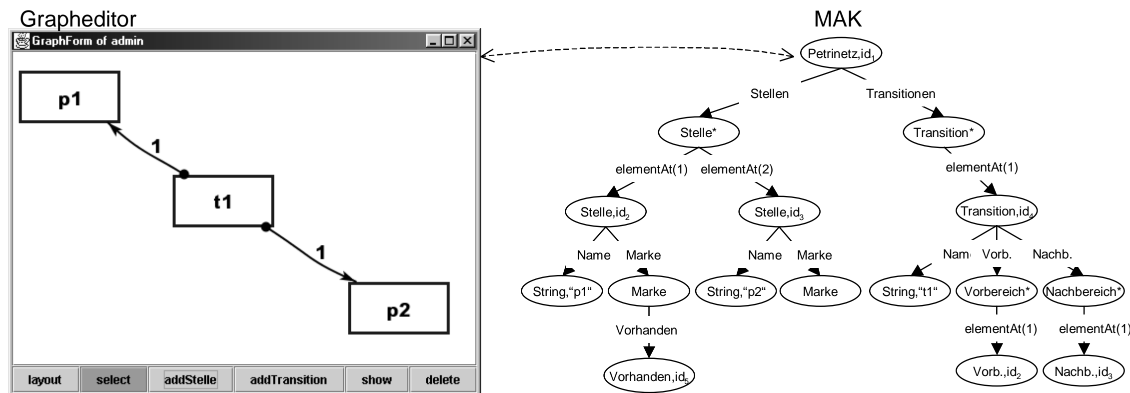


**Abbildung 4-9: Grapheditor für Petrinetze ohne Anpassung**

Jedoch würden die folgenden Layoutanpassungen der Graphelemente bereits eine wesentliche Verbesserung bedeuten:

- Phasen werden oval dargestellt
- Vorbereichskanten zeigen zur Transition hin
- Vor- und Nachbereichskanten werden nicht mit ihrem Index markiert
- Markierte Phasen werden grau markiert

Man beachte, dass es die letzte Anpassung bereits ermöglicht, die Anfangsmarkierung und einen Simulationslauf des Petrinetzes zu visualisieren.

Um die Darstellung der Phasen durch Ovale zu ermöglichen, ist eine Anpassung der Layout-Komponente zur Darstellung der graphischen Knoten vom Typ *Stelle* notwendig. Die entsprechende GRACE-Eingabekomponente heißt *StelleNodePresentation*. Sie wurde von der EMUGEN-Teilkomponente PREGRACE (vgl. Abbildung 3-5) generiert.

```
node presentation StelleNodePresentation {
  requires automaton Selectable;
  not_selected:
    (CompositeNode(TextFigure,CircleFigure),
       ClickInteractor, PerimeterConnection)
  selected:
    (SelectedNodeFigure(CompositeNode(TextFigure,
       CircleFigure)),
     ResizeInteractor, PerimeterConnection)
}
```

Eine *NodePresentation*-Komponente von GRACE ordnet den Zuständen (hier: `not_selected` und `selected`) des zugehörigen Automaten (hier: `Selectable`) ein Tripel zu, das aus einer möglicherweise zusammengesetzten geometrischen Form (hier z.B.: `CompositeNode(TextFigure,CircleFigure)`), einem Interaktor (hier z.B.: `ClickInteractor`) und einer Kantenverbindungsstrategie (hier: `PerimeterConnection`) besteht. Die Anpassung an der implizit definierten GRACE-Spezifikations-

komponente *StelleNodePresentation* besteht in der Ersetzung der geometrischen Form **BoxFigure** mit **CircleFigure** (oben unterstrichen).

Um die Richtung der Vorbereichskanten umzudrehen und ihre Beschriftung zu entfernen ist eine Anpassung der entsprechenden *EdgePresentation*-Komponente *VorbereichEdgePresentation* notwendig:

```
edge presentation VorbereichEdgePresentation {
  requires automaton Selectable;
  not_selected:
    (CurveFigure,ClickInteractor,Arrowhead,DotDecorator)
  selected:
    (CurveControl(CurveFigure),
     TextCurveInteractor,Arrowhead,DotDecorator)
}
```

Eine *EdgePresentation*-Komponente von GRACE ordnet den Zuständen (hier ebenfalls: **not_selected**, **selected**) des zugehörigen Automaten (hier ebenfalls: **Selectable**) ein Quadrupel zu, das aus einer möglicherweise zusammengesetzten geometrischen Form (hier z.B.: **CurveFigure**), einem Interaktor (hier z.B. ebenfalls: **ClickInteractor**) und den geometrischen Formen am Quell- (hier: **Arrowhead**) und Zielknoten (hier: **DotDecorator**) besteht. Zur Anpassung wurden die geometrischen Formen an Quell- und Zielknoten ausgetauscht und die geometrische Teilform für die Beschriftung (**TextEdge**) weggelassen. Gemäß der implizit verwendeten Standardlayoutregel, befindet sich **Arrowhead** am Zielknoten und die geometrische Form lautet **CompositeEdge(TextEdge,CurveFigure)** im Zustand **not_selected**.

Nun sind noch die markierten Phasen grau darzustellen. Dies kann mit folgender Regel geschehen:

```
node Stelle {:
  if (Marke.Vorhanden)
    FILL_PAINT=lightGray;
  else
    FILL_PAINT=white;
:}
```

Die Semantik applikationsspezifischer Bedingungen der Art **Marke.Vorhanden** wird in Abschnitt 5.3 allgemein definiert. Die spezielle, hier angewandte, Bedingung **Marke.Vorhanden** trifft genau dann zu, wenn beim darzustellenden (logischen) MAK-Knoten vom Typ **Stelle** der mit dem Selektor **Marke** angesprochene Variantenknoten vom Typ **Marke** die aktuelle Ausprägung **Vorhanden** besitzt.

Abbildung 4-10 zeigt den auf die beschriebene Weise angepassten Grapheditor für Petrinetze. Dabei wird derselbe MAK wie in Abbildung 4-9 visualisiert.

Interpretiert man in Anlehnung an [Obe96] ein Petrinetz als Workflow [JBS97], so hat man mit dem in Abbildung 4-10 gezeigten Grapheditor bereits eine einfache graphische Spezifikationsmöglichkeit für Workflows geschaffen. In Abschnitt 5.2.3 wird dieses Beispiel um eine Simulationsmöglichkeit erweitert.

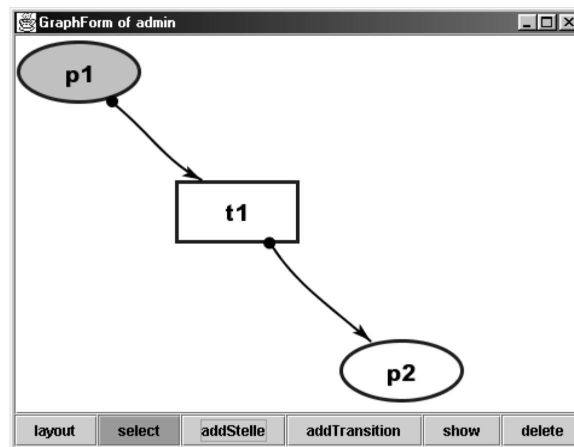


**Abbildung 4-10: Grapheditor für Petrinetze mit Layoutanpassung**

## 4.3 Semantik des EMU-Datenmodells

Die Semantik [Win93] eines EMU-Datenmodells liegt in der Festlegung eines EMU-Systems, also einem komplexen Zustandsübergangssystems, das im Allgemeinen unendlich viele Zuständen aufweist. Nun sind also die folgenden Fragen formal zu klären:

- Was ist der Zustand eines EMU-Systems?
- Wie verändert das EMU-System bei einer Interaktion seinen Zustand?
- Welches Zustandsübergangssystem wird durch ein EMU-System definiert?

Entsprechend dieser Fragen ergibt sich die Gliederung dieses Abschnitts. Wir beziehen uns nun auf die Menge der Produktionen des Datenmodells $Prod_{DM}$, eine Universalmenge $\mathcal{U}$ und eine total geordnete, unendliche Menge von Identifikatoren $ID \subset \mathcal{U}$ . Um auf den Informationsgehalt der Produktionen kompakt zugreifen zu können, verwenden wir eine Reihe von Abkürzungen, die im Weiteren als Substitutionen zu lesen sind. D.h. die linken Seiten stellen parametrisierte Abkürzungen für die rechten Seiten dar.

$$Tupel_{DM}(X,T_i,S_i) \equiv \exists p \in Prod_{DM} : p = (\texttt{X::=T}_1\texttt{:S}_1\texttt{..T}_i\texttt{:S}_i\texttt{..T}_n\texttt{:S}_n)$$

$$Tupel_{DM}(X) \equiv \exists p \in Prod_{DM} : p = (\texttt{X::=T}_1\texttt{:S}_1\texttt{..T}_n\texttt{:S}_n)$$

$$Var_{DM}(X,A_i) \equiv \exists p \in Prod_{DM} : p = (\texttt{X::=A}_1\texttt{|..|A}_i\texttt{|..A}_n)$$

$$Var_{DM}(X) \equiv \exists p \in Prod_{DM} : p = (\texttt{X::=A}_1\texttt{|..|A}_n)$$

$$List_{DM}(X,E) \equiv \exists p \in Prod_{DM} : p = (\texttt{X::=E*})$$

$$List_{DM}(X) \equiv \exists p \in Prod_{DM} : p = (\texttt{X::=E*})$$

$$Ref_{DM}(X,S,T,c) \equiv \exists p \in Prod_{DM} : p = (\texttt{X::=S->T c})$$

$$Ref_{DM}(X) \equiv \exists p \in Prod_{DM} : p = (\texttt{X::=S->T c})$$

$$Basistyp(X) \equiv \text{X ist Basistyp von EMU}$$

Damit lässt sich die Menge der Interaktionsdatentypen $\boldsymbol{IDT_{DM}}$ eines Datenmodells $DM$ wie folgt definieren:

$$IDT_{DM} =_{def} \{X \mid Tupel_{DM}(X) \vee Var_{DM}(X) \vee List_{DM}(X) \vee Ref_{DM}(X) \vee Basistyp_{DM}(X)\}$$

### 4.3.1 Zustand eines EMU-Systems

Der Zustand eines EMU-Systems wurde in der bisherigen, informellen Betrachtung als MAK bezeichnet und als ein getypter, mit Werten attibutierter Baum veranschaulicht (vgl. etwa Abbildung 4-9). Nun wird diese anschauliche Betrachtungsweise formalisiert, wozu folgende Komponenten notwendig sind:

- Formaler Baumbegriff
- Typisierung von Baumknoten
- Baumbewertungsfunktion

Zum Zugriff auf die Knoten des MAKs verwenden wir nach [Gan78] Elemente aus $\mathbb{N}*$. Die Typmarkierung der Knoten eines EMU-Systems vom Typ $X$ kann statisch mit der Funktion $type_X : \mathbb{N}* \cup \{\bot\} \to IDT_{DM} \cup \{\bot\}$ erfolgen.

$$type_X(\varepsilon) =_{def} X$$

$$type_X(p \circ \langle i \rangle) =_{def} \begin{cases} T_i & \text{wenn } Tupel_{DM}(type_X(p), T_i, S_i) \\ A_i & \text{wenn } Var_{DM}(type_X(p), A_i) \\ E & \text{wenn } List_{DM}(type_X(p), E) \\ \bot & \text{sonst} \end{cases}$$

Die statische Typbestimmung von Knoten ist möglich, weil Alternativen als innere Knoten verwaltet werden (vgl. etwa Abbildung 4-3). Die Zulassung der (Pseudo-) Knotenbezeichnung $\bot$ als möglichen Argumentwert erleichtert in Abschnitt 5.3 die Definition der Semantik von Pfadausdrücken. Wir brauchen im Folgenden die Menge der (gültigen) Positionen $Pos_X =_{def} \{p \in \mathbb{N}* \mid type_X(p) \neq \bot\}$. Damit können wir in Anlehnung an [Gan78] die Menge von MAKen eines EMU-Systems vom Typ als eine Menge getypter Bäumen betrachten.

$$Tree_X =_{def} \{\tau_X \subset Pos_X \mid$$
$$\tau_X \text{ ist endlich,}$$
$$p \circ \langle i \rangle \in \tau_X \Rightarrow \begin{cases} p \in \tau_X & \\ Var_{DM}(type_X(p), A_i) & \Rightarrow \forall j \neq i : p \circ \langle j \rangle \notin \tau_X \\ List_{DM}(type_X(p), E) & \Rightarrow \forall j < i : p \circ \langle j \rangle \in \tau_X \end{cases}$$
$$\}$$

Variantenknoten haben also höchstens einen Nachfolger (wir sprechen auch von seiner ***aktuellen Ausprägung***) und die Anzahl der Nachfolger von Listenknoten ist nicht beschränkt.

Wir bezeichnen ein Datenmodell *DM* genau dann als ***konstruktiv,*** wenn $\forall X \in IDT_{DM} : Tree_X \neq \emptyset$.

Weiterhin benötigen wir die Menge der Knotenbewertungsfunktionen:

$$NodeValue_X =_{def} \{\sigma_X \in ((Pos_X \cup \{\bot\}) \rightarrow \mathcal{U}) \mid$$

$$\sigma_X(p) \in \begin{cases} ID \cup \{\bot\} & \text{wenn } Tupel_{DM}(type_X(p)) \vee Ref_{DM}(type_X(p)) \\ Dom(type_X(p)) & \text{wenn } Basistyp(type_X(p)) \end{cases},$$

$$\forall p,q \in Tree_X : Tupel_{DM}(type_X(p)), Tupel_{DM}(type_X(q)), \sigma_X(p) = \sigma_X(q)$$

$$\Rightarrow p = q$$

$$\}$$

Elemente aus $NodeValue_X$ bezeichnen wir auch weiterhin mit $\sigma_X$. Jeder Position eines Baums mit Wurzel *X* wird mit $\sigma_X$ also ein Wert aus der Universalmenge $\mathcal{U}$ zugewiesen. Wiederum erlauben wir die (Pseudo-) Knotenbezeichnung $\bot$ als Argument.

Für jeden Basistyp *B* bezeichnet $Dom(B) \subseteq \mathcal{U}$ die zugehörige, vordefinierte Wertemenge.

Bisher blieb die Frage offen, woher die Idenfikatoren der Tupelknoten stammen. Um im Folgenden zu definieren, wie z.B. der Anfangszustand oder eine neues Listenelement erzeugt werden, gehen wir von einer statisch festgelegten, total geordneten Grundmenge von Identifikatoren *ID* aus. Für jedes Paar $(\tau_X, \sigma_X)$ definieren wir die Menge der benutzten Identifikatoren $UID_{\tau_X}^{\sigma_X} =_{def} \{i \in ID \mid \exists p \in \tau_X : Tupel_{DM}(type_X(p)), \sigma_X(p) = i\}$ sowie die Menge der unbenutzten Identifikatoren $UUID_{\tau_X}^{\sigma_X} =_{def} ID \setminus UID_{\tau_X}^{\sigma_X}$. Aufgrund der obigen Konstruktivitätsbedingung (Endlichkeit von $\tau_X$) ist $UUID_{\tau_X}^{\sigma_X}$ niemals leer und damit das kleinste Element von $UUID_{\tau_X}^{\sigma_X}$ immer vorhanden. Wir greifen darauf mit $least_{\tau_X}^{\sigma_X}$ zu.

Der Zustand eines (Einbenutzer-)EMU-Systems ergibt sich also aus dem Paar $(\tau_X, \sigma_X)$.

### 4.3.2 Interaktionen auf EMU-Systeme

Wir nennen $ImplAction_X$ die Menge der implizit definierten Interaktionen eines EMU-Systems vom Typ *X*.

$$ImplAction_X =_{def} VarAction_X \uplus AddAction_X \uplus RemoveAction_X \uplus RefAction_X \uplus BasisAction_X$$

$$VarAction_X =_{def} \{setAlt(p, A_i) \mid p \in Pos_X, A_i \in IDT_{DM}, Var_{DM}(type_X(p), A_i)\}$$

$$AddAction_X =_{def} \{add(p,i) \mid p \in Pos_X, i \in \mathbb{N}, List_{DM}(type_X(p))\}$$

$$RemoveAction_X =_{def} \{remove(p,i) \mid p \in Pos_X, i \in \mathbb{N}, List_{DM}(type_X(p))\}$$

$$RefAction_X =_{def} \{setRef(p,r) \mid p,r \in Pos_X, Ref_{DM}(type_X(p))\}$$

$$BasisAction_X =_{def} \{setValue(p,v) \mid p \in Pos_X, v \in Dom(type_X(p)), Basistyp_{DM}(type_X(p))\}$$

Konform zur Literatur [BDD93, Sch97, Eic98] verwenden wir einen statischen Interaktions-Begriff. Ein dynamischer Interaktions-Begriff, bei dem jedes Paar $(\tau_X, \sigma_X)$ eine Menge von Interaktionen indiziert, wird erst bei der formalen Betrachtung des Aufgabenmodells notwendig (Abschnitt 5.3.1), weil dort Interaktionen Vorbedingungen zugewiesen werden. Im Gegensatz dazu gelten die implizit definierten Interaktionen als immer auslösbar. Sie werden jedoch nur dann im dynamisch festgelegten Formular dem Benutzer präsentiert, wenn sie einem Knoten zugeordnet sind, der im Baum vorkommt.

Es folgt die Definition der implizit definierten Transitionsfunktion.

$$\Delta_X^I : Tree_X \times NodeValue_X \times ImplAction_X \to Tree_X \times NodeValue_X$$

$$\Delta_X^I(\tau_X, \sigma_X, setAlt(p, A_i)) =_{def} \begin{cases} \Delta_X^{Var}(\tau_X, \sigma_X, (p, A_i)) & \text{wenn } Var_{DM}(type_X(p), A_i), p \in \tau_X \\ (\tau_X, \sigma_X) & \text{sonst} \end{cases}$$

$$\Delta_X^I(\tau_X, \sigma_X, add(p,a)) =_{def} \begin{cases} \Delta_X^{Add}(\tau_X, \sigma_X, (p,i)) & \text{wenn } List_{DM}(type_X(p)), p \circ \langle i-1 \rangle, p \in \tau_X \\ (\tau_X, \sigma_X) & \text{sonst} \end{cases}$$

$$\Delta_X^I(\tau_X, \sigma_X, remove(p,a)) =_{def} \begin{cases} \Delta_X^{Remove}(\tau_X, \sigma_X, (p,i)) & \text{wenn } List_{DM}(type_X(p)), p \circ \langle i \rangle, p \in \tau_X \\ (\tau_X, \sigma_X) & \text{sonst} \end{cases}$$

$$\Delta_X^I(\tau_X, \sigma_X, setRef(p,a)) =_{def} \begin{cases} \Delta_X^{Ref}(\tau_X, \sigma_X, (p,a)) & \\ & \text{wenn } Ref_{DM}(type_X(p), S, type_X(a), c), p \in \tau_X, \mathcal{B}[\![c]\!]_{\tau_X}^{\sigma_X}\ (a, \langle\rangle) \\ (\tau_X, \sigma_X) & \text{sonst} \end{cases}$$

$$\Delta_X^I(\tau_X, \sigma_X, setValue(p,v)) =_{def} \begin{cases} \Delta_X^{Basis}(\tau_X, \sigma_X, (p,v)) & \text{wenn } Basistyp_{DM}(type_X(p)), v \in Dom(type_X(p)) \\ (\tau_X, \sigma_X) & \text{sonst} \end{cases}$$

$\Delta_X^{Basis}$ sei geeignet vordefiniert. Die Semantik boolescher Ausdrücke $\mathcal{B}[\![.]\!]_{\tau_X}^{\sigma_X}$ ist zu definieren. Diese ist von der syntaktischen Struktur der Bedingung (z.B. `[Anstellung is Fest & Funktion is Leitend]`), vom aktuellen Baum $\tau_X$, der aktuellen Knotenbewertungsfunktion $\sigma_X$ sowie der Position des entsprechenden Knotens abhängig (z.B. trifft die obige Bedingung in Abbildung 4-3 für den Mitarbeiter mit der Position $\langle 3,1 \rangle$ zu, für $\langle 3,2 \rangle$ jedoch nicht).

Bei der Definition eines Mehrbenutzersystems im nächsten Kapitel wird klar, warum im Allgemeinen sogar *zwei* Baumpositionen als Argument für $\mathcal{B}[\![.]\!]_{\tau_X}^{\sigma_X}$ notwendig sind. Die zweite Position dient dann als „Anker einer Benutzerposition“. Zum vollen Verständnis der Definition von $\mathcal{B}[\![.]\!]_{\tau_X}^{\sigma_X}$ sind daher einige Aspekte zur Benutzerdefinition notwendig, die im nächsten Kapitel eingeführt werden. Deshalb wird diese Funktion auch erst im nächsten Kapitel definiert.

Bei den hier betrachteten Einbenutzersystemen, die nur aus dem Datenmodell generiert wurden, stellt die einzige Benutzerposition die des Wurzelknotens, also $\langle\rangle$, dar. Daher ergibt sich $\langle\rangle$ oben als aktueller Parameter bei der Anwendung von $\mathcal{B}[\![c]\!]_{\tau_X}^{\sigma_X}$.

Zur Definition von $\Delta_X^{Var}$, $\Delta_X^{Add}$ und $\Delta_X^{Remove}$ benötigen wir weiterhin die folgenden beiden Hilfsfunktionen:

$$create_X, destroy_X : Tree_X \times NodeValue_X \times Pos_X \rightarrow Tree_X \times NodeValue_X$$

$create_X(\tau_X, \sigma_X, p)$ erzeugt an der Position $p$ einen neuen Knoten. Handelt es sich bei $p$ um einen Tupelknoten (was statisch festgelegt ist), so wird ein bisher unbenutzten Identifikator als Wert $p$ zugeordnet und für jede Komponente von $type_X(p)$ ein weiterer Knoten angelegt (rekursiver Aufruf von $create_X$), d.h. ein neuer Unterbaum erstellt. Handelt es sich bei $p$ nicht um einen Tupelknoten, so wird $\sigma_X(p) = \perp$ gesetzt.

$destroy_X$ ist das Gegenstück von $create_X$. $destroy_X(\tau_X, \sigma_X, p)$ entfernt den Knoten (bei Tupelknoten auch alle seine Unterknoten) aus $\tau_X$ und setzt den Wert aller nicht entfernten Referenzknoten, die auf $p$ oder einen Unterknoten von $p$ zeigen, auf $\perp$.

$$\begin{aligned}
& create_X(\tau_X, \sigma_X, p) =_{def} (\tau'_X, \sigma'_X) \text{ mit} \\
& \quad Tupel_{DM}(type_X(p), T_i, S_i) \Rightarrow \\
& \quad\quad \tau_X^0 = \tau_X \cup \{p\}, \sigma_X^0 = \sigma_X[least_{\tau_X}^{\sigma_X} / p], \\
& \quad\quad (\tau_X^i, \sigma_X^i) = create_X(\tau_X^{i-1}, \sigma_X^{i-1}, p \circ \langle i \rangle), \\
& \quad\quad \tau'_X = \tau_X^n, \sigma'_X = \sigma_X^n, \\
& \quad Var_{DM}(X) \vee List_{DM}(X) \vee Ref_{DM}(X) \vee Basistyp(X) \Rightarrow \\
& \quad\quad \tau'_X = \tau_X \cup \{p\}, \sigma'_X = \sigma_X[\perp / p]
\end{aligned}$$

$create_X$ terminiert immer bei konstruktiven EMU-Systemen, weil die obige Konstruktivitätsbedingung rekursive Tupelproduktionen ausschließt. Nochmals sei darauf hingewiesen, dass die Ausdrück zum Zugriff auf die Produktionen, wie z.B. $Tupel_{DM}(type_X(p), T_i, S_i)$, als parametrisierte Substitution zu lesen sind.

$$\begin{aligned}
& destroy_X(\tau_X, \sigma_X, p) =_{def} (\tau'_X, \sigma'_X) \text{ mit} \\
& \quad \forall q \in Tree_{type_X(p)} \setminus \{\langle\rangle\} : \tau'_X = \tau_X \setminus \{p \circ q\}, \\
& \sigma'_X(r) = \begin{cases} \perp & Ref_{DM}(type_X(r)), pos_{\tau_X}^{\sigma_X}(\sigma_X(r)) = p \circ q \\ \sigma_X(r) & sonst \end{cases}
\end{aligned}$$

Die Funktion $pos_{\tau_X}^{\sigma_X} : ID \cup \{\perp\} \rightarrow Pos_X \cup \{\perp\}$ liefert dabei die Position eines Tupelknotens mit dem übergebenen Identifikator im aktuellen Zustand $(\tau_X, \sigma_X)$. $destroy_X$ entfernt nicht den übergebenen Knoten selbst aus der Menge der Baumknoten, weil es sonst bei der Anwendung in der Transitionsdefinition schwierig wäre, die Baumbedin-

gung von $\tau_X$ (speziell bzgl. der Listennachfolger, bei denen keine „Lücken" erlaubt sind) aufrecht zu erhalten.

Wir sind nun in der Lage, die Transitionen, die an Variantenknoten durchgeführt werden können, formal darzustellen:

$$\Delta_X^{Var}(\tau_X, \sigma_X, (p, A_i)) =_{def} (\tau'_X, \sigma'_X) \quad \text{mit}$$
$$(\tau''_X, \sigma''_X) = \begin{cases} destroy_X(\tau_X, \sigma_X, p \circ \langle j \rangle) & \text{wenn } p \circ \langle j \rangle \in \tau_X \\ (\tau_X, \sigma_X) & \text{sonst} \end{cases}$$
$$(\tau'_X, \sigma'_X) = create_X(\tau''_X, \sigma''_X, p \circ \langle i \rangle)$$

Bei der informellen Beschreibung sprechen wir bei $(\tau_X, \sigma_X)$ vom ***Ausgangszustand***, bei $(\tau'_X, \sigma'_X)$ vom ***Folgezustand*** einer Transition. Eine aktuelle Ausprägung am angesprochenen Variablenknoten (formaler Parameter ***p*** in obiger Definition) im Ausgangszustand wird entfernt und ein neuer Knoten erzeugt.

$$\Delta_X^{Add}(\tau_X, \sigma_X, (p, i)) =_{def} (\tau'_X, \sigma'_X) \quad \text{mit}$$
$$\tau''_X = \tau_X \cup \{ p \circ \langle \max\{ j \in \mathbb{N} \mid p \circ \langle j \rangle \in \tau_X \} + 1 \rangle \},$$
$$\sigma''_X(r) = \begin{cases} \sigma_X(p \circ \langle j-1 \rangle \circ q) & \text{wenn } r = p \circ \langle j \rangle \circ q, i < j \\ \perp & \text{wenn } r = p \circ \langle i \rangle \\ \sigma_X(r) & \text{sonst} \end{cases},$$
$$(\tau'_X, \sigma'_X) = create_X(\tau''_X, \sigma''_X, p \circ \langle i \rangle)$$

□

$$\Delta_X^{Remove}(\tau_X, \sigma_X, (p, i)) =_{def} (\tau'_X, \sigma'_X) \quad \text{mit}$$
$$(\tau''_X, \sigma''_X) = destroy_X(\tau_X, \sigma_X, p \circ \langle i \rangle)$$
$$\sigma'_X(r) = \begin{cases} \sigma''_X(p \circ \langle j+1 \rangle \circ q) & \text{wenn } r = p \circ \langle j \rangle \circ q, i \le j \\ \sigma''_X(r) & \text{sonst} \end{cases},$$
$$\tau'_X = \tau''_X \setminus \{ p \circ \langle \max\{ j \in \mathbb{N} \mid p \circ \langle j \rangle \in \tau_X \} \rangle \}$$

Sowohl bei der Listenlöschoperation als auch beim Umschalten der aktuellen Ausprägung einer Variante kann es bei Verwendung der hier diskutierten, implizit definierten Transitionen dazu kommen, dass Referenzwerte auf $\perp$ gesetzt werden.

$$\Delta_X^{Ref}(\tau_X, \sigma_X, (p, q)) =_{def} (\tau_X, \sigma_X[\sigma_X(q)/p])$$

### 4.3.3 Zustandsübergangssystem

Nun kann das interaktive System, das durch ein EMU-Datenmodell definiert ist, durch ein Quadrupel konform zur Darstellung in [BDD93] definiert werden. In Anlehnung an [Sch97, BDD93] schreiben wir $(\tau_X, \sigma_X) \xrightarrow{a} (\tau'_X, \sigma'_X)$ um auszudrücken, dass durch Anwendung der Benutzerinteraktion *a* auf den Zustand $(\tau_X, \sigma_X)$ eines EMU-Systems der

Folgezustand $(\tau'_X, \sigma'_X)$ entsteht und $(\tau_X, \sigma_X) \xrightarrow{\langle a_1 \ldots a_n \rangle} (\tau'_X, \sigma'_X)$ für die natürliche Erweiterung auf eine Folge von Benutzerinteraktionen $\langle a_1 \ldots a_n \rangle$.

$$
\begin{aligned}
Sys_{DM} = (Actions, State, \rightarrow, Init) \quad & mit \\
& Actions = ImplActions_{start_{DM}}, \\
& State = \{Init\} \cup \{s \in (Tree_{start_{DM}} \times NodeValue_{start_{DM}}) \mid \\
& \qquad \exists a_1, \ldots, a_n \in Actions : Init \xrightarrow{\langle a_1, \ldots, a_n \rangle} s\}, \\
& s \xrightarrow{a} s' \Leftrightarrow \Delta'^{I}_{start_{DM}}(s, a) = s', \\
& Init = create_{start_{DM}}(\varnothing, \varnothing, \langle\rangle)
\end{aligned}
$$

Überall undefinierte Funktionen betrachten wir als leere Menge $\varnothing$. Betrachtet man ein EMU-System als Datenbank, so entspricht das EMU-Datenmodell *DM* einem Datenbankschema und ein einzelner Zustand $(\tau_X, \sigma_X)$ einem Inhalt (Instanz) der Datenbank.

## 4.4 Begründung und mögliche Alternativen

In diesem Abschnitt soll eine Begründung der Sprachentscheidung für das EMU-Datenmodell vorgenommen werden. Zunächst wird eine Motivation für die Verwendung eines hierarchischen Datenmodells[5] geliefert, anschließend werden mögliche Alternativen diskutiert.

### 4.4.1 Warum ein hierarchisches Datenmodell?

Der wichtigste Grund für die Verwendung eines hierarchischen Datenmodells sind die Vorteile, die sich im Hinblick auf die Generierung entsprechender Benutzungsoberflächen zur Bearbeitung der in diesem Fall baumartig organisierten Datenstrukturen ergeben. Diese Benutzungsoberflächen sind in ähnlicher Weise hierarchisch strukturiert wie die zu bearbeitenden Datenstrukturen selbst und können – beim Mehrbenutzerbetrieb entsprechend dem Benutzermodell geeignet transformiert – aus der Datenstruktur errechnet werden.

Die hierarchische Strukturierung ist jedoch auch ein grundsätzliches Prinzip, das etwa bei der Spezifikation interaktiver Systeme nicht nur auf den Datenaspekt, sondern auch auf Aufgaben und Dialoge angewandt werden könnte.

Beispielsweise könnte eine Aufgabe *Software-Entwicklung* aus den Teilaufgaben *Analyse*, *Entwurf*, *Implementierung* und *Test* bestehen, wobei die Teilaufgabe *Analyse* wiederum aus *fachlicher* und *technischer Analyse* bestehen kann.

Eine Dialogkontrolle zum *Abheben beim Geldauszahlungsautomat* (GAA) kann aus der *Anmeldung* und dem *Eingeben des Betrag* zusammengesetzt sein. Dabei kann die *Anmeldung* wiederum aus dem *Einschieben der Karte* und dem *Eingeben der Geheimzahl* bestehen.

---

[5] Genauer: eine Datenmodellierungssprache. Wenn aus dem Zusammenhang klar ist, um welche Sprachebene es sich handelt, wird hier stets der einfachere anstelle des formal korrekteren Begriffs verwendet.

Bei der allgemeinen Betrachtung solcher Sachverhalte, die zur Erstellung eines Generators notwendig ist, stellt sich die Frage, ob es notwendig ist, für die Aufgabenspezifikation und für die Dialogkontrolle Sprachelemente zum hierarchischen Aufbau bereitzustellen.

Würde man sie für jeden dieser Aspekte zur Verfügung stellen, so wäre die Semantik der Sprachelemente zur hierarchischen Komposition festzulegen. Ähnlich wie es beim EMU-Datenmodell Sprachelemente wie Tupel oder Variante gibt, wären beim Aufgabenmodell Sprachelemente zur sequentiellen oder parallelen bzw. zur obligatorischen oder optionalen Ausführung von Teilaufgaben angebracht (vgl. etwa [WJK93]).

Weiterhin wäre es bei der Spezifikation eines Generators notwendig, komplexe Verknüpfungsmöglichkeiten vorzusehen. Beispielsweise könnte der Entwickler eine Aufgabe *Analyse* mit einem Datentyp *Analysedokument* verknüpfen wollen (vgl. Abbildung 2-8), wobei – je nach Anwendung – auch die umgekehrte Sichtweise (Verknüpfung des Datentyps mit der Aufgabe) sinnvoll sein könnte.

Insgesamt gesehen ergäbe sich bei der Unterstützung des rekursiven Aufbaus bei allen Aspekten eine Vielfalt von Strukturierungs- und Verknüpfungsmöglichkeiten und damit eine sehr komplexe und schwer zu durchschauende Spezifikationssprache.

Daher sieht EMU eine hierarchische Strukturierung zunächst nur beim Datenmodell vor. Dennoch muss auf die Möglichkeiten, die etwa durch die hierarchische Strukturierung von Aufgaben gegeben sind, nicht verzichtet werden, wenn das grundlegende Konzept der Datentypen eingesetzt wird.

Es ist nämlich möglich, die vielfältigen sprachlichen Kompositionsmuster der anderen Aspekte durch ein hierarchisches Datenmodell wie das EMU-Datenmodell zu modellieren. Deshalb werden Datentypen dieser Art auch im Übersetzerbau zur Beschreibung der abstrakten Syntax verwendet [WM97]. D.h. es wird dem Entwickler – also dem Benutzer von EMUGEN – ermöglicht, ein eigenes (möglicherweise hierarchisches) Aufgabenkonzept mit Hilfe des EMU-Datenmodells zu modellieren. Man kann also auf der Metaebene zur Beschreibung von Aufgabenmodellierungssprachen mit dem EMU-Datenmodell arbeiten.

Wir wollen dies hier an einer einfachen Erweiterung des Beispiels 4-2 (*Petrinetze*) zeigen. Nach [Obe96] können Petrinetze zur Modellierung von Aufgaben verwendet werden. Wenn es nun beispielsweise notwendig sein soll, auch stellenverfeinerte (also hierarchische) Petrinetze zu verwenden, kann dies durch die folgende Anpassung von Beispiel 4-2 (genauer: des zugehörigen EMU-Datenmodells) erfolgen:

```
PetriNetz::=Stelle*:Stellen Transition*:Transistionen
Stelle::=String:Name Marke Petrinetz:Teilnetz
.../* Rest wie gehabt */
```

Damit hat man bereits in Grundzügen eine hierarchisches Beschreibungsinstrument für Aufgaben. In Kapitel 6 wird mit *Ad-hoc Workflows* ein konkreteres Beispiel für die Verwendung des EMU-Datenmodells zur Definition einer (graphischen) Aufgabenbeschreibungssprache vorgestellt.

### 4.4.2 Alternative Datenmodelle

Alternativ zur dargestellten EMU-Datenmodellierungssprache wären folgende Datenmodelle denkbar:

- Algebraische Spezifikationen (z.B. wurden in [Bau96] daraus Dialogmodelle generiert);
- ER-Modell [Che76] bzw. entsprechende Erweiterungen wie OOA-Modelle [RBP91] oder Objektdiagramme nach der UML-Notation [OMG01];
- Relationenmodell [Cod70];
- Sortendeklarationen [Bro97, Sch97]; Datentypen höherer Programmiersprachen; Data Dictionaries nach [DeM79]; XML [XML98]; ODL [Vos00];

Eine ausführliche Übersicht über weitere Datenmodelle liefert [Vos00]. Wie [Bal00] verwenden wir den Begriff ***Klassendiagramm*** als Überbegriff für OOA-Klassendiagramme und UML-Objektdiagramme. Da wir den Ansatz sehr abstrakt diskutieren, umfassen wir mit dem Begriff Klassendiagramm weitgehend auch ER-Modelle nach [Che76].

Wir werden die in den ersten drei Punkten oben aufgeführten Datenmodelle in eigenen Abschnitten diskutieren. Unterschiede des EMU-Datenmodells zu den im letzten Punkt aufgeführten Datentypbeschreibungen sind bis auf das Referenzkonzept von EMU nur in Details vorhanden. Die spezielle Entscheidung zugunsten der verwendeten BNF-artigen Notation liegt in ihrer Kompaktheit, ihrer direkten graphischen Darstellungsmöglichkeit und der Tatsache, dass sie ein anerkanntes und bewährtes Basiskonzept der Informatik darstellt.

Im Gegensatz zu einem Entwickler, der sich als Benutzer einer Modellierungssprache oft eine Vielfalt von Sprachelementen und Kompositionsmöglichkeiten erhofft, um für jeden Anwendungsfall „gewappnet zu sein“, ist bei der Konzeption eines Generators eine besondere Sorgfalt beim Sprachdesign angebracht, weil es (wie in Abschnitt 4.3 geschehen) festzulegen gilt, ob und „welches“ interaktive Informationssystem aus einem gegebenen Eingabemodell zu generieren ist.

Offensichtlich muss dabei der Generator entscheiden, ob es sich bei der Eingabe um ein gültiges Eingabemodell handelt (syntaktisches und semantisches Wortproblem). Ferner muss die Funktionalität des generierten interaktiven Systems – abhängig von den Eingabemodellen – festgelegt werden. So trivial und grundlegend dieser Sachverhalt auch erscheint, er wird an dieser Stelle deshalb erwähnt, weil er bei vielen, in der Literatur bekannten, CADUI-Werkzeugen nicht (ausreichend) behandelt wird. Als Folge kann nicht festgestellt werden, wozu die einzelnen Werkzeuge (nicht) in der Lage sind.

### 4.4.3 Algebraische Spezifikationen

Algebraische Spezifikationen werden beispielsweise in [Bau96] zur Spezifikation abstrakter Benutzungsoberflächen verwendet. Ein Vorteil dieses Ansatzes besteht darin, dass es in bestimmten Fällen möglich ist, Dialogabläufe durch Analyse der gegebenen Axiome herzuleiten. Die Nachteile bestehen jedoch in der Länge der Spezifikation bereits trivialer Beispiele und in der notwendigen formalen Schulung der Entwickler. Ferner ergibt sich beim Übergang einer algebraischen Spezifikation in ein ablauffähiges interaktives System ein Bruch in der üblichen Semantik dieser Beschreibungstechnik (vgl.

den Bruch zwischen dem *objektorientierten* ER-Modell und dem *relationalen* Modell). Dieser entsteht durch die aus technischen Gründen notwendige Verwendung von (Objekt-)Identifikatoren bei interaktiven Systemen zur Laufzeit. Identifikatoren könnte zwar durch geschickte Verwendung zusätzlicher Funktionssymbole auf Spezifikationsebene simuliert werden, der Umgang mit den ohnehin sehr formalen Beschreibungstechnik wird dabei jedoch noch schwieriger.

### 4.4.4 Klassendiagramme

Klassendiagramme werden beispielsweise bei JANUS [BHK96] als Datenmodell verwendet. UI-Generatoren, die wie BOSS [Sch97] (und daher auch das direkt daran angelehnte FORMGEN) oder ADAPTIVE FORMS [FS98] die automatische Generierung *dynamischer* Formulare ermöglichen, verwenden dagegen klassische Datentypbeschreibungen.

Zur Begründung, warum bei EMUGEN Datentypbeschreibungen anstelle von Klassendiagrammen verwendet werden, ist eine verhältnismäßig detaillierte und implementierungsnahe Betrachtung eines Beispiels notwendig. Das folgende Beispiel ist Teil von Beispiel 4-1.

**Beispiel 4-3: Personalverwaltung**

> *Ein Personal besteht aus Mitarbeitern, bei denen unterschieden werden soll, ob sie festangestellt oder frei sind. Zu Mitarbeitern soll der Name geführt werden. Bei festangestellten Mitarbeitern soll ferner das Eintrittsdatum, bei freien der Vertragsbeginn und das Vertragsende verwaltet werden.*

Abbildung 4-11 zeigt ein denkbares, naheliegendes Klassendiagramm zu diesen Anforderungen in UML-Notation [OMG01].

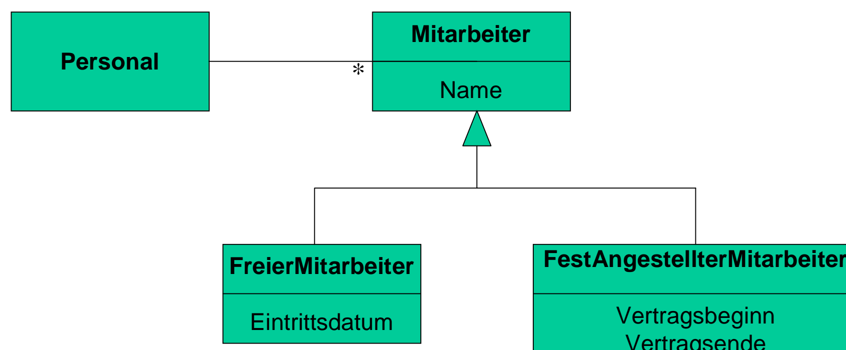


**Abbildung 4-11: Klassendiagramm zu Personal**

Die direkte Umsetzung dieses Entwurfs in eine klassische Datentypbeschreibungssprache ist in dieser Form nicht möglich, weil dort keine Vererbung im objektorientierten Sinne unterstützt wird. Beispielsweise muss bei der Angabe eines entsprechenden EMU-Datenmodells ein zusätzlicher Typ `Anstellung` verwendet werden, der als weitere Komponente von `Mitarbeiter` zu führen ist.

```
Personal::=Mitarbeiter*:Personal
Mitarbeiter::=String:Name Anstellung
Anstellung::= Fest | Frei
```

```
Fest::=String:Eintrittsdatum
Frei::=String:Vertragsbeginn String:Vertragsende
```

Nun gehen wir wieder von dem Klassendiagramm aus Abbildung 4-11 aus. Es sei angenommen, dass ein Objekt der Klasse *Personal* von einem Formular bearbeitet wird und bereits ein Objekt von *FreierMitarbeiter* als Listenelement geführt ist.

Nun wird der von diesem Objekt modellierte Mitarbeiter festangestellt. Technisch ist es dazu notwendig, ein neues Objekt anzulegen, in welches der Wert der Instanzvariablen *Name* der Oberklasse *Mitarbeiter* kopiert wird, und das Formular aufgrund des neuen dynamischen Typs des bearbeiteten *Mitarbeiter*-Objekts zu ändern. Zusätzlich müssen auch alle Referenzen, die auf das alte Objekt verwiesen haben, benachrichtigt werden. Ferner kann sich – insbesondere bei Mehrbenutzersystemen – der Fall ergeben, dass das zu verändernde Objekt zeitgleich in weiteren Benutzungsoberflächen angezeigt wird und bei Änderungen entsprechend der MVC-Architektur eine Benachrichtigung dieser Formulare notwendig wird. Dieser Vorgang ist bei einer Änderung der Objektidentität äußerst kompliziert.

Bei der OO-Modellierung sind „bessere" Lösungen (d.h. alternative Klassendiagramme) für Modellierungsprobleme dieser Art durchaus bekannt (vgl. das Entwurfsmuster *Dekorierer* [GHJ96] bzw. die Modellierungstechnik *Delegation* [Oes98]), wodurch zumindest Teile der oben angesprochenen Schwierigkeiten umgangen werden. Eine entsprechende Lösung wird in Abbildung 4-12 gezeigt.

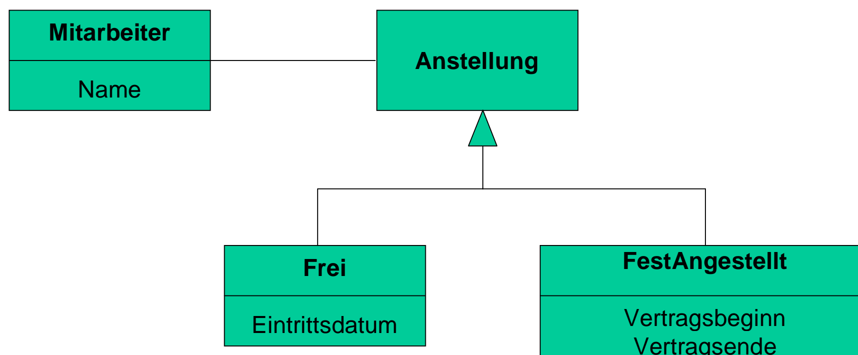


**Abbildung 4-12: Delegation des Anstellungsaspekts**

Man beachte die Isomorphie zwischen Abbildung 4-12 und den oben dargestellten Produktionen zum IDT `Mitarbeiter`. Offensichtlich sind zwar Klassendiagramme das allgemeinere Konzept als EMU-Datenmodelle (Abbildung 4-11 kann z.B. nicht direkt durch ein EMU-Datenmodell dargestellt werden), andererseits gibt es bei der Datenmanipulation im Zusammenhang mit der Vererbung Unklarheiten. Im Abschnitt 4.3 wird durch die Definition von $\Delta_X^{Var}(\ldots)$ gezeigt, wie bei EMU die oben dargestellte Änderung eines dynamischen Typs vollzogen wird.

Möglich – und für den praktischen Einsatz von EMUGEN aufgrund der Verbreitung von Klassendiagrammen durchaus angebracht – wäre es, Klassendiagramme als Eingabemodell zu verwenden und in ähnlicher Weise in ein EMU-System umzusetzen, wie es mit dem EMU-Datenmodell getan wird. D.h. man könnte aus einem – geeignet einge-

schränkten – Klassendiagramm ein EMU-Datenmodell erzeugen und somit die oben angesprochenen Probleme umgehen.

Als Vorteil für Klassendiagramme könnte die Möglichkeit der Vererbung von Methoden genannt werden. Auf der Ebene der Modellierung interaktiver Systeme werden mit Methoden auch Interaktionen modelliert, die direkt durch den Benutzer aus der Benutzungsoberfläche heraus ausführbar sind (vgl. Abbildung 2-1).

Im Formular könnte man für jede dieser Methoden (im EMU-Aufgabenmodell in Kapitel 5 werden sie *Aktionen* genannt) beispielsweise einen Button an geeigneter Stelle platzieren. Genauso wie man Klassen Methoden zuordnen kann, so kann man sie auch den Datentypen zuordnen und erreicht durch Komposition in einer Tupelproduktion die gleiche Funktionalität wie bei Vererbung – mit dem zusätzlichen Vorteil, dass der zur Methodenausführung im Formular zu platzierende Button optisch automatisch dem Datentyp zugeordnet wird, dem er in der Spezifikation zugeordnet wurde.

Der Nachteil klassischer Datentypbeschreibungssprachen im Vergleich zu Klassendiagrammen, nur kontextfreie Sachverhalte beschreiben zu können, wird im EMU-Datenmodell durch die Verwendung eines Referenzkonzepts aufgehoben.

Ein weiterer Vorteil für Datentypbeschreibungssprachen liegt in der bereits beschriebenen Ausnutzung der hierarchischen Typbeschreibung bei der Generierung der entsprechenden Benutzungsoberflächen.

Insgesamt ergaben vor allem die Probleme bzgl. der dynamischen Typänderung den Ausschlag für die Verwendung von Datentypbeschreibungssprachen anstelle von Klassendiagrammen.

### 4.4.5 Relationenmodell

Das Relationenmodell [Cod70] ermöglicht die Definition semantischer Konsistenzen, etwa die referentielle Integrität.

EMU-Systeme sind interaktiv. Daher ist es notwendig, neben der statischen Semantik von Daten (wie z.B. die referentielle Integrität) auch die Dynamik, d.h. die Veränderung von Daten aufgrund von Benutzerinteraktionen zu beschreiben. Dies beinhaltet einerseits die Semantik typischer Datenmanipulationsmöglichkeiten wie sie etwa auch die Einfüge-, Änderungs- und Löschkommandos in SQL [Vos00] darstellen, andererseits aber auch die „feingranulare“ Manipulation von Daten, etwa wenn ein einzelner Mausklick das Löschen eines Elements aus einer Liste auslöst.

Im Gegensatz zu einer Kommandosprache wie SQL muss bei der Verwendung einer formularorientierten Benutzungsoberfläche zur Datenmanipulation eine Reihe von Aspekten geklärt werden:

- Wie löst der Benutzer ein Kommando aus?
- Wie gibt der Benutzer mögliche Kommandoargumente ein?
- Wie erfährt der Benutzer mögliche Kommandoresultate (vgl. erfolgreiche vs. fehlerhafte Ausführung einer Datenbanktransaktion)?

Man beachte, dass Kommandoargumente selbst wiederum möglicherweise komplexe Daten darstellen (z.B. die Überweisungsdaten bei einem Geldausgabeautomaten, vgl. Ab-

schnitt 6.2.2), für die geeignete Benutzungsoberflächen zu generieren sind. Bei Kommandoargumenten handelt es sich meist um flüchtige Daten, die nach der Kommandoausführung verworfen werden. Diese Daten sind nicht sinnvoll durch ein Relationenmodell zu beschreiben, weil Integritätsbedingungen relationaler Modelle (z.B. NOT-NULL-Werte [Vos00]) zu scharf sind und während einer interaktiven Bearbeitung im Allgemeinen verletzt werden. Anders ausgedrückt sind die ACID-Forderungen an eine Transaktion [Vos00] nicht innerhalb einer interaktiven Bearbeitung aufrecht zu erhalten bzw. sinnvoll umzusetzen.

Demgegenüber sollten Daten, die über einen längeren Zeitraum bestehen (konzeptionell ist dies auch der funktionale Kern eines EMU-Systems) natürlich durchaus mit einem Relationenmodell spezifiziert oder mit einem geeigneten Werkzeug aus einem ER-Modell oder Klassendiagramm generiert werden.

Wie bereits in der Einleitung beschrieben, stellen EMU-Systeme eine optionale Schnittstelle zu Datenbank- oder Workflow-Management-Systemen zur Verfügung. Ein EMU-System ist daher in erster Linie als eine Verallgemeinerung der Dialogsichten [Den91] auf mehrere Benutzer zu betrachten und weniger als eigenständige Datenbank. Aus dieser Überlegung heraus liegt es näher, ein Datentypkonzept zu verwenden.

Datenmodellierung wird bei der Entwicklung interaktiver Informationssysteme an verschiedenen Stellen vorgenommen, beispielsweise bei der Modellierung der zugrunde liegenden Datenbank und bei der Modellierung der Daten, die der Benutzer zur Erledigung einer Aufgabe einzugeben hat. Aufgrund der dazu eingesetzten Werkzeuge (etwa relationale Datenbanken oder objektorientierte Programmiersprachen) werden dazu in der Praxis häufig mehrere verschiedene Datenmodellierungssprachen (etwa SQL oder Klassendiagramme) eingesetzt.

Um das Generierungskonzept von EMU darzustellen und auch zu implementieren, liegt jedoch kein konzeptioneller Grund vor, verschiedene Datenmodellierungssprachen zu berücksichtigen. D.h. sowohl der gesamte Mehrbenutzer-Applikationsschnittstellenkern (MAK) als auch die Daten, die ein Benutzer bei einer Aufgabenbearbeitung einzugeben hat (z.B. die Daten für einen Kreditantrag) werden in einer einheitlichen Datenmodellierungssprache, dem EMU-Datenmodell, definiert.

# 5 Aufgaben- und Benutzermodell

In diesem Kapitel werden das Aufgaben- und das Benutzermodell von EMU eingeführt. Wir betrachten sie gemeinsam, weil es in beiden Modellierungssprachen Sprachelemente gibt, deren Sinn und Notwendigkeit im Zusammenhang verständlicher zu vermitteln sind. Wiederum werden zunächst die Sprachelemente anhand eines Beispiels erklärt. Anschließend erfolgt die informelle Beschreibung der Ausführung generierter EMU-Systeme. Dabei handelt es sich nun im Gegensatz zum letzten Kapitel um Mehrbenutzersysteme. In Abschnitt 5.3 und 5.4 folgen die Definition der Semantik solcher EMU-Systeme und eine Begründung des Sprachdesigns. Die vollständige Eingabesyntax steht im Anhang.

## 5.1 Sprachelemente

Durch das Aufgabenmodell werden

- *Aktionen* und
- *Aktivitätsnetze*

modelliert und den Interaktionsdatentypen (IDTs) des Datenmodells zugeordnet. Durch das Benutzermodell werden

- *Benutzertypen* und deren
- *Zugriffe*

auf das EMU-System modelliert. Aktionen des Aufgabenmodells stellen Sprachelemente für den Entwickler dar und haben daher nur indirekt etwas zu tun mit dem Aktionsbegriff, der in Anlehnung an [BDD93] in den formalen Abschnitten 4.3 und 5.3 verwendet wird. Liegt die Gefahr einer Verwechslung (für den Leser) vor, so werden die Aktionen des Aufgabenmodells als *explizite Aktionen* bezeichnet.

Wir betrachten die Sprachelemente des Aufgaben- und Benutzermodells anhand der folgenden Erweiterung des Beispiels 4-1 *Abteilung*.

**Beispiel 5-1:** ***EMU-Projekte***

> *Die Abteilung beschäftigt sich mit der Software-Entwicklung mit EMUGEN. Daher ist die Ausführung von EMU-Projekten zu unterstützen. Ein EMU-Projekt besteht aus den Phasen Analyse und Entwurf. Während der Analysephase erstellt der Kunde eine Beschreibung der fachlichen Anforderungen an das zu entwickelnde System. Nachdem der Kunde die Anforderungen erstellt hat, kann er den Entwurf einleiten. Während des Entwurfs erstellt der Entwickler die EMU-Modelle und greift dabei auf die Anforderungen des Kunden lesend zu. Der Entwickler kann aus den EMU-Modellen mit Hilfe von EMUGEN den ausführbaren Prototyp generieren. Bei einer fehlerhaften Generierung ist dem Entwickler eine Fehlermeldung anzuzeigen. Der Entwickler hat zum Prototyp eine Anwendungsbeschreibung zu erstellen. Wenn der Entwickler glaubt, der Prototyp genügt den Anforderungen, so kann er erneut die Analysephase einleiten. Dazu muss sich aber der Prototyp in einem ausführbaren Zustand befinden. Wenn die Analysephase erneut eingeleitet wurde, so kann der Kunde den Prototyp ausführen und überprüfen, ob seine Anforderungen erfüllt wurden bzw. inwiefern seine Anforderungen zu ändern bzw. zu erweitern sind. Nach einer Änderung der Anforderungen kann der Kunde erneut den Entwurf einleiten.*

Die Sprachelemente des Aufgaben- und Benutzermodells ermöglichen uns eine direkte Umsetzung dieser Anforderungsbeschreibung in eine konstruktive Spezifikation eines ausführbaren Mehrbenutzersystems. Wichtig für das Verständnis dieser Sprachelemente ist die grundlegende Vorgehensweise bei EMU, die darin besteht, das Datenmodell als Grundgerüst der Eingabe zu verwenden. Um diese Vorgehensweise genauer zu erklären, benötigen wir zuerst das Datenmodell für das obige Beispiel.

Im Datenmodell wird ein EMU-Projekt durch eine Tupelproduktion modelliert. Das Tupel `EMUProjekt` besteht aus den Komponenten `Kunde`, `Entwickler`, `Anforderungen`, `EMUModelle` und `Prototyp`. Für `Anforderungen` und die einzelnen EMU-Modelle legen wir den Basistyp `Text` fest.

```
EMUProjekt::=Kunde
             Entwickler
             Text:Anforderungen
             EMUModelle
             Prototyp
Kunde::=String:Name
Entwickler::=String:Name
EMUModelle::=Text:Datenmodell
             Text:Aufgabenmodell
             Text:Benutzermodell
             Text:Layoutmodell
             Text:Schnittstelle
Prototyp::=  Text:Fehlerbeschreibung
             Text:Anwendungsbeschreibung
```

### Datenmodell als Grundgerüst der Eingabemodelle

Beim Betrachten der informellen Anforderungen und des zugehörigen Datenmodells fällt auf, dass `Kunde` und `Entwickler` der Ambivalenz unterliegen, einerseits dem Daten-

modell, andererseits auch dem Benutzermodell zuzugehören. Zum Datenmodell gehören sie, weil es interaktiv möglich sein muss, die Namen des Kunden und des Entwicklers einzugeben. Zum Benutzermodell gehören sie, weil sie gleichzeitig auch die ausführenden „Einheiten" des EMU-Systems, also die realen Benutzer, modellieren. Kompakt kann dieser Sachverhalt durch eine Markierung bestimmter IDTs des Datenmodells als Benutzer modelliert werden.

Ferner besteht ein Zusammenhang zwischen den Aufgaben und Abläufen der informellen Aufgabenbeschreibung (z.B. *Prototyp erstellen, Analysephase einleiten*) und den IDTs des Datenmodells. Dieser Zusammenhang wird bei der Beschreibung der einzelnen Sprachelemente (Aktionen und Aktivitätsnetze) in den nachfolgenden Abschnitten klar. Zur konkreten Umsetzung erfolgt eine Zuordnung von Aktionen und Abläufen zu den IDTs des Datenmodells.

Abbildung 5-1 skizziert die Vorgehensweise, das Datenmodell als Grundgerüst zu verwenden. Dabei wird nur die erste Produktion für den Starttyp **`EMUProjekt`** graphisch dargestellt. Die IDTs **`Kunde`** und **`Entwickler`** werden als Benutzer markiert. Sie erhalten außerdem ein Aktivitätsnetz, durch welches zur Laufzeit verwaltet wird, ob sie gerade am System mitarbeiten oder nicht. Ferner erhalten auch **`EMUProjekt`** und **`Prototyp`** Aktionen und Aktivitätsnetze, die in den nachfolgenden Abschnitten beschrieben werden.

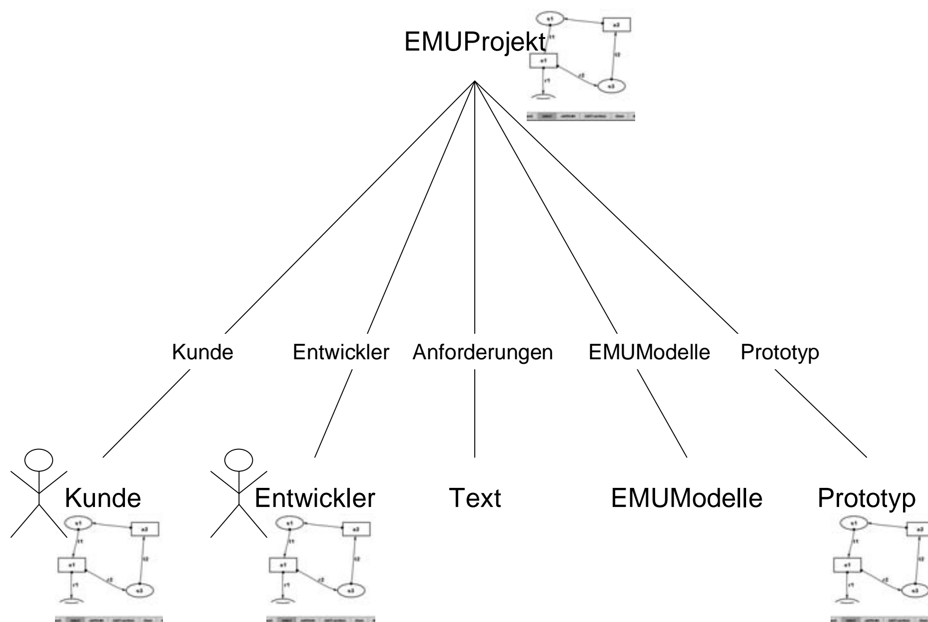


**Abbildung 5-1: Datenmodell als Grundgerüst der Eingabemodelle**

Dieser Ansatz ist mit dem Vorgehen beim Übersetzerbau zu vergleichen. Dort werden die Produktionen für den abstrakten Syntaxbaum mit Attributen und Attributierungsregeln angereichert [WM97].

### 5.1.1 Aktionen

Für Beispiel 5-1 ergeben sich folgende Aktionen, die im Folgenden genauer beschrieben werden.

```
actions of EMUProjekt {
  entwerfen      [true]          -> {::};
  analysieren    [Prototyp.Ok]   -> {::};
}
actions of Prototyp {
  generieren     [parent.Entwurf]     -> {:
    /* Kommandos zum Generieren und Installieren
       des Prototypen                               */
    try {
      EMUManager.generate(parent.EMUModelle);
      return Ok;
    } catch (EMUManager.GenerateException e)
      Fehlerbeschreibung.setValue(e.toString());
     return Fehler;
  :} Ok | Fehler;
  ausfuehren     [Ok]       -> {:
    /* Kommando zum Ausfuehren des Prototypen      */
    EMUManager.exec();
  :};
 }
```

Den Tupeltypen des Datenmodells können im Aufgabenmodell eine beliebige Anzahl von Aktionen zugeordnet werden. Aktionen des Aufgabenmodells sind ähnlich wie die implizit durch das Datenmodell definierten Aktionen direkt durch den Benutzer oder externe Systeme ausführbar. In den Formularen werden dazu geeignete Interaktionsobjekte platziert.

Jede Aktion hat einen Namen, eine Vorbedingung, ein Kommando (vergleichbar mit dem semantischen Rumpf von Scanner- oder Parser-Generatoren [Lev92, Hud99]) und eine Folge von Resultaten. Jedes dieser Resultate entspricht einer booleschen Aussagenvariable. Nach der Aktionsausführung, d.h. nach Ausführung der semantischen Aktion, ist genau eine davon gültig. Die Resultate können als Schaltbedingung für die Transitionen der Aktivitätsnetze verwendet werden, die im nachfolgenden Abschnitt beschrieben werden.

Die Vorbedingung ist ein boolescher Ausdruck, wie er bereits beim Datenmodell als Referenzbedingung verwendet wurde.

Der semantische Rumpf einer Aktion ist ein Kommando in einer Kommandosprache, die in dieser Arbeit nicht vollständig, sondern nur exemplarisch betrachtet wird. Um dennoch in Abschnitt 5.3 die Semantik von EMU-Systemen mit Aktionsausführung definieren zu können, wird dort gefordert, dass für jedes Kommando eine semantische Funktion im Sinne der denotationellen Semantik [Win93] vorhanden ist.

Warum die Kommandosprache nicht vollständig betrachtet werden kann, liegt an ihrer Komplexität. Diese entsteht durch die vielfältigen technischen Möglichkeiten[6], die sie bieten muss:

- In der Kommandosprache müssen Zustandsänderungen des MAKs programmiert werden. Beispielsweise kann während der Ausführung des Kommandos der Aktion `generieren` ein Fehler der Eingabemodelle (Komponente `EMUModelle`) festgestellt werden, welcher dem Entwickler als Fehlerbeschreibung mitgeteilt werden muss. Formal wird dazu der Wert des Basisknotens an der Position $\langle 5,1 \rangle$ (`Prototyp` ist die 5.Komponente von `EMUProjekt`, `Fehlerbeschreibung` ist die 1.Komponente von `Prototyp`) des jeweiligen MAKs vom Typ `EMUProjekt` auf den entsprechenden Wert gesetzt. Dieser Wert entspricht in diesem Fall einem String, der den Fehler beschreibt.
- Die Schnittstelle zu externen Systemen muss programmierbar sein. Beispielsweise wird im semantischen Rumpf von `generieren` das externe System *EMUManager* aufgerufen, welches neben der eigentlichen Generierung auch die Ausführung generierter EMU-Systeme verwaltet. Es verfügt daher über Funktionen zum Generieren und Ausführen von EMU-Systemen, die im Kommando von `generieren` bzw. `ausfuehren` aufgerufen werden. Soll zur Datenhaltung als externes System etwa eine relationale Datenbank verwendet werden, so wäre es notwendig, eine Schnittstelle zu SQL zu programmieren.

Aufgrund dieser Anforderungen wurde auch im Rahmen der Implementierung von EMUGEN keine eigene Kommandosprache entwickelt, sondern Java [GJS96] verwendet, weil nur mit einer gebräuchlichen Programmiersprache die Kommunikation mit beliebigen externen Systemen mit vertretbarem Aufwand erfolgen kann. Da diese Kommunikation teilweise zusammen mit der Programmierung von Zustandsänderungen des MAKs erfolgt (z. B. beim oben angesprochenen Anzeigen der Fehlerbeschreibung), liegt es nahe, für die Programmierung der Zustandsänderungen dieselbe Sprache zu verwenden.

Daher generiert EMUGEN aus dem Datenmodell eine ***API*** (*Application Programmer Interface*), die vom Entwickler dazu benutzt werden kann, die Zustandsänderungen in den Kommandos der Aktionen zu programmieren. Die API wird zusammen mit einem Beispiel in Anhang C beschrieben.

Im Wesentlichen wird dabei für jeden Datentyp des Datenmodells eine Klasse erstellt, deren Methoden es dem Entwickler erlauben, die in Kapitel 4 betrachteten, impliziten Aktionen als „programmiersprachliches Kommando" auszuführen. Wir nennen diese Klassen im Weiteren ***Knotenklassen***. Ferner können neue Teilbäume erstellt (vgl. die Funktion $create_x(...)$ aus Abschnitt 4.3.2) und an beliebige Positionen im MAK eingehängt werden. Um die Implementierung beliebiger Auswertungen mit Hilfe des *Besucher-Entwurfsmuster* [GHJ96] zu ermöglichen, wird eine Besucherklasse für die Knotenklassen generiert. Durch die Verwendung der generierten API erhält man leicht lesbare,

[6] *technisch* bedeutet hier, dass es nicht um die Mächtigkeit der Kommandosprache im Sinne der Berechenbarkeit, sondern etwa um die Kommunikation mit externen Systemen geht.

gut verständliche Kommandos, so dass wir diese bei der Darstellung der Beispielkommandos gegebenenfalls verwenden werden[7].

Im Aufgabenmodell von Beispiel 5-1 gibt es Aktionen für das Einleiten der Entwurfs- bzw. Analysephase, für das Auslösen der Generierung und für die Ausführung des Prototypen. Ferner müssen in einem (in Abschnitt 5.1.2 betrachteten) Aktivitätsnetz die Phasen **`Analyse`** und **`Entwurf`** mit den zugehörigen Transitionen definiert werden.

Die Aktionen **`entwerfen`** und **`analysieren`** werden dem Tupeltyp **`EMUProjekt`** zugeordnet und besitzen jeweils kein Kommando. Die Vorbedingung von **`ausfuehren`** nimmt Bezug auf die Phase **`Ok`**, die im Aktivitätsnetz von **`Prototyp`** (im nächsten Abschnitt) definiert ist. Informell bedeutet diese Vorbedingung, dass nur wenn der Prototyp „in Ordnung“ ist, soll der Entwickler bzw. der Kunde ihn ausführen können.

Mit der Aktion **`generieren`** kann der Entwickler die Generierung des Prototyp auslösen. Durch die Vorbedingung von **`generieren`** wird festgelegt, dass diese Aktion nur ausgeführt werden kann, wenn im Vorgängerknoten (**`parent`**) die Bedingung **`Entwurf`** erfüllt ist. **`Entwurf`** ist keine Komponente von **`EMUProjekt`**, sondern eine Phase des zugehörigen Aktivitätsnetzes, das im nächsten Abschnitt definiert wird.

Bedingungen können sich also auch auf die Phasen eines IDTs und damit zur Laufzeit auf die aktuell aktiven Phasen eines Aktivitätsnetzes beziehen. Dieser Sachverhalt ist bei der Definition der Semantik boolescher Ausdrücke zu berücksichtigen (Abschnitt 5.3.2).

Das Kommando der Aktion **`generieren`** erstellt mit dem Aufruf der statischen Methode **`generate`** der Klasse **`EMUManager`** den Prototyp. Die Klasse **`EMUManager`** kapselt das externe System *EMUManager*, welches die Verwaltung generierter EMU-Systeme und den Zugriff darauf über den Projektnamen ermöglicht.

Bei der Generierung kann es zu Fehlern kommen, wenn beispielsweise der Benutzer (in diesem Fall ist der Benutzer der Entwickler des EMU-Projekts) kein korrektes Datenmodell eingegeben hat. Dieser Sachverhalt wird mit zwei Resultaten der Aktion **`generieren`** modelliert. Eine fehlerhafte Ausführung führt dazu, dass dem Benutzer eine Beschreibung des Fehlers angezeigt wird.

Dazu sind zwei Dinge notwendig:

- **`Fehlerbeschreibung`** (also die Komponente von **`Prototyp`**) erhält als neuen Wert die durch das Ausnahmeobjekt [GJS96] übergebene Zeichenreihe. Dazu dient das Kommando **`Fehlerbeschreibung.setValue(e.toString())`**.
- **`Fehlerbeschreibung`** wird für den Entwickler sichtbar geschaltet.

Das Sichtbarschalten der Fehlerbeschreibung wird in den nächsten Abschnitten durch das Aktivitätsnetz und die Zugriffe für **`Prototyp`** definiert.

---

[7] Man könnte die Kommandos natürlich auch in einer Pseudo-Notation angeben. Die dadurch suggerierte Sprachunabhängigkeit ist jedoch effektiv nicht vorhanden, wenn man davon ausgeht, dass die Implementierung der Schnittstelle zu externen Systemen in einer konkret vorliegenden Sprache (es könnte natürlich auch eine andere als Java sein) erfolgen muss.

Im Kommando der Aktion `generieren` wird das jeweilige Resultat durch eine entsprechende `return`-Anweisung erzeugt. Hier zeigt sich ein weiterer Vorteil der Verwendung einer ausgereiften Hochsprache wie Java, nämlich ein Ausnahme-Mechanismus, der insbesondere bei der Kommunikation mit externen Systemen unerlässlich ist.

Das Kommando der Aktion `ausfuehren` führt den Prototyp aus. Es benutzt dazu die Methode `exec` der bereits angesprochen Klasse `EMUManager`. Hierbei geht die vorliegende Implementierung davon aus, dass die Ausführung des Prototyps (die asynchron in einem neuen Prozess abläuft) funktioniert. D.h. es ist kein Fehlerfall vorgesehen, der etwa aufgrund technischer Probleme eintreten könnte. Wollte man solche Fehler abfangen, so könnte ein ähnlicher Ausnahme-Mechanismus wie bei `generate` und ein entsprechendes Fehlerresultat verwendet werden.

### 5.1.2 Aktivitätsnetze

Aktivitätsnetze dienen dazu, verschiedene Ausführungszustände und ihre Übergänge auf den Objekten des Datenmodells zu definieren. Abhängig von den Ausführungszuständen können Vorbedingungen von Aktionen und Zugriffe der Benutzer definiert werden. Wir werden Aktivitätsnetze ebenfalls zunächst textuell beschreiben und später eine mögliche graphische Visualisierung diskutieren (vgl. Abbildung 5-5).

Aktivitätsnetze können als Variante von Petrinetzen [Esp95] mit Ähnlichkeiten zu Bedingungs-/Ereignisnetzen (B/E-Netzen) [Bal00, Obe96] und Dialognetzen [GFJ96] betrachtet werden. Genau wie ein B/E-Netz besteht ein Aktivitätsnetz aus Bedingungen (wir sprechen aufgrund unserer Anwendungssituation auch von *Phasen*, die *aktiv*, d.h. markiert, sein können) und Transitionen mit Vor- und Nachbereich (wir sprechen von *Vor-* und *Nachphasen*).

Anders als bei B/E-Netzen sind Transitionen nicht durch ein einzelnes Ereignis markiert, sondern durch eine Aktion und optional durch ein Resultat dieser Aktion. Eine Transition schaltet genau dann, wenn jede Phase des Vorbereichs aktiv ist (also auch, wenn der Vorbereich leer ist), die zugehörige Aktion ausgeführt wurde und – im Fall, dass ein Resultat angegeben wurde – die Ausführung des entsprechenden semantischen Rumpfes das in der Markierung angegebene Resultat liefert. Es können auch mehrere Transitionen mit demselben Aktion/Resultat-Paar markiert sein und daher auch gleichzeit schalten.

Wenn eine Transition schaltet, so werden alle Phasen des Vorbereichs deaktiviert (in Petrinetz-Terminologie: die Markierung wird entfernt) und alle Phasen des Nachbereichs aktiviert. Phasen im Schnitt von Vor- und Nachbereich sind nach dem Schalten der Transition aktiv, d.h. die Aktivierung überwiegt die Deaktivierung.

Bei der Definition eines Aktivitätsnetz erfolgt die Festlegung einer Menge von ***Startphasen***, was der Initialbelegung bei Petrinetzen entspricht. Einem IDT kann höchstens ein Aktivitätsnetz zugeordnet werden.

Im Beispiel ergeben sich folgende Aktivitätsnetze.

```
actnet of EMUProjekt {
  -> Analyse
  Analyse->modellieren ->Entwurf
  Entwurf->analysieren ->Analyse
```

```
    }
    actnet of Prototyp {
      -> Ok
      Ok->generieren.Fehler->Fehler
      Fehler ->generieren.Ok->Ok
    }
```

Man beachte, dass die Phasen nicht explizit definiert werden, sondern sich aus ihrer Verwendung bei der Definition als Startphase oder als Vor- oder Nachphase einer Transition ergeben.

Im Aktivitätsnetz von `EMUProjekt` werden `Analyse` als Startphase und zwei Transitionen definiert. Aus der Phase `Analyse` gelangt man mit der Aktion `modellieren` in die Phase `Entwurf`. Mit der Aktion `analysieren` gelangt man von `Entwurf` wieder zurück zur Phase `Analyse`. In beiden Fällen ist keine Fallunterscheidung bzgl. eines Resultats der jeweiligen Aktion notwendig.

Beim Aktivitätsnetz von `Prototyp` sind die beiden Transitionen nicht nur mit der Aktion `generieren`, sondern auch mit den zugehörigen Resultaten `Fehler` und `Ok` markiert.

Die informelle Bedeutung des Aktivitätsnetzes von Prototyp besteht darin, dass nach einer fehlerhaften Ausführung der Aktion `generieren` die Phase `Fehler` eingenommen wird und nach einer erfolgreichen die Phase `Ok`. Die Aktion `ausfuehren` kommt im Aktivitätsnetz von Prototyp nicht vor, weil ihre Ausführung den logischen Ablauf nicht beeinflusst.

Obwohl die beiden Aktivitätsnetze von `EMUProjekt` und `Prototyp` eine ähnliche Struktur aufweisen, beschreiben sie aus abstrakter Sicht Sachverhalte, die bei CADUI-Werkzeugen meist in getrennten Spezifikationsteilen modelliert werden.

Das Aktivitätsnetz von `EMUProjekt` beschreibt einen *Workflow*, weil Kunde und Entwickler gemeinsam an einem Vorgang, d.h. an der Abwicklung eines EMUProjekts, arbeiten und in beiden Phasen unterschiedlich involviert sind, d.h. jeweils unterschiedliche Aktionen ausführen und Komponenten bearbeiten.

Demgegenüber beschreibt das Aktivitätsnetz von `Prototyp` eine *Dialogkontrolle* zur Erstellung des Prototypen durch den Entwickler.

Aktivitätsnetze werden also benutzt zum Modellieren von

- Vorgängen (Workflows), die von mehreren Benutzern gemeinsam ausgeführt werden,
- Dialog(kontroll)en und
- Aktivitäten eines einzelnen Benutzers (vgl. das Aktivitätsnetz in Abschnitt 5.1.5 mit dem *Sachbearbeiter*–Konzept bei [Den91]).

Obwohl in der Literatur [Den91, Lar92, Sch97, Eic98] im Zusammenhang mit der Modellierung von Dialogen von *Zuständen* gesprochen wird, verwenden wir im Weiteren den Begriff *Phase*, um einheitlich zu bleiben. Die hier nur informell beschriebene Ausführungssemantik von Aktivitätsnetzen wird in Abschnitt 5.3 vollständig definiert.

### 5.1.3 Benutzertypen

Benutzertypen werden durch Tupelproduktionen des Datenmodells modelliert. Aus anderer Sicht kann man sagen, dass bestimmte Tupeltypen des Datenmodells als Benutzertyp markiert werden. Zur Laufzeit modelliert dann jeder MAK-Knoten, dessen IDT als Benutzer markiert wurde, einen realen Benutzer.

Die Eigenschaften des realen Benutzers können somit als Bedingungen formuliert werden. Damit ist es möglich, die bereits angesprochene Ambivalenz von Benutzerdaten, die einerseits als Teil des Problembereichs bzw. Datenmodells, andererseits aber auch als Teil des Benutzermodells betrachtet werden können, kompakt zu modellieren.

Konkret erfolgt bei Beispiel 5-1 die Markierung von `Kunde` und `Entwickler` als Benutzer eines EMU-Projekts wie folgt:

```
User::= Entwickler (Name) | Kunde (Name)
```

Die Festlegung der Benutzertypen erfolgt durch eine besondere Art einer Variantenproduktion mit dem Bezeichner (Schlüsselwort) `User` und einer Markierung, die angibt, welche Komponenten der Benutzertypen bei der Systemanmeldung durch den realen Benutzer zur Identifikation eingegeben werden müssen (in diesem Fall jeweils die Komponente `Name`).

#### Benutzereigenschaften

Bei der Definition der Zugriffe (ein Beispiel kommt im nächsten Abschnitt) kann es notwendig sein, Fallunterscheidungen bzgl. Benutzereigenschaften vorzunehmen.

Die Abfrage von Benutzereigenschaften erfolgt über einen – jedoch nur imaginär vorhandenen – Variantenknoten vom Typ `User`. Abbildung 5-2 soll dies veranschaulichen. Sie zeigt einen MAK zu `EMUProjekt`, wobei nur die erste Baumebene eingezeichnet wurde.

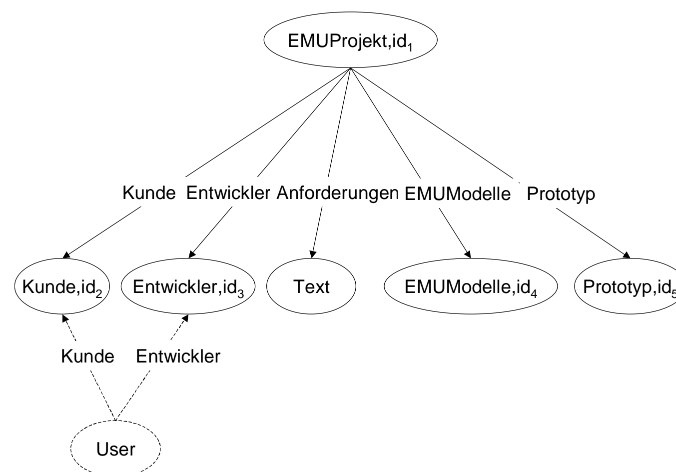


**Abbildung 5-2: Der imaginäre User-Knoten**

Zur Auswertung eines entsprechenden booleschen Ausdrucks ist als Kontext immer die Position des jeweiligen Benutzers zu beachten. Z.B. wird der Ausdruck „`Entwickler`“ in bestimmten Teilen der Eingabespezifikation (z.B. als Anwendbarkeitsbedingung einer

Zugriffsdefinition, ein Beispiel kommt im nächsten Abschnitt) als boolescher Ausdruck interpretiert, der etwa bei der Auswertung in einem EMU-System vom Typ `EMUProjekt` im Kontext des Benutzers an der Position $\langle 1 \rangle$ `false`, an der Position $\langle 2 \rangle$ jedoch `true` liefert.

Anders ausgedrückt hat der Benutzer mit der Position $\langle 1 \rangle$ die Eigenschaft, Entwickler zu sein, während der Benutzer mit der Position $\langle 2 \rangle$ diese nicht hat.

Formal wird in Abschnitt 5.3 festgelegt, dass an einem Variantenknoten *p* zu einer Produktion $V::=A_1|\ldots|A_i|\ldots|A_n$ der Ausdruck $A_i$ genau dann `true` ergibt, wenn dort $A_i$ aktuell ausgeprägt ist (formal: $p \circ \langle i \rangle \in \tau_X$, wenn $\tau_X$ der aktuelle Baum ist). Entsprechendes funktioniert auch am imaginären Variantenknoten vom Typ *User*. Beispiele für die Anwendung von Benutzereigenschaften ergeben sich bei den im Folgenden betrachteten Definitionen der Zugriffe der Benutzer auf ein EMU-System.

### 5.1.4 Zugriffe

Pro Tupeltyp des Datenmodells kann der Zugriff der verschiedenen Benutzer auf die zugehörigen Komponenten definiert werden. Ein Zugriff kann dabei abhängig vom Zustand des EMU-Systems, d.h. beispielsweise auch von den aktuell aktiven Phasen des zugeordneten Aktivitätsnetzes und den Benutzereigenschaften, formuliert werden.

Die Definition eines Zugriffs besteht aus einer Folge von Zugriffsregeln. Jede Zugriffsregel besteht aus einer ***Anwendbarkeitsbedingung*** (ein boolescher Ausdruck) und einer Folge von Paaren, wobei jedes dieser Paare aus einem Zugriffsoperator (`read, write` oder `exec`) und einer Komponente oder einer Aktion gebildet wird.

Wird für einen Tupeltyp `T` kein Zugriff definiert, so kann implizit jeder Benutzer auf alle Komponenten und Aktionen von `T` zugreifen.

Im Beispiel 5-1 ergeben sich folgende Zugriffsdefinition auf den Starttyp `EMUProjekt`:

```
access on EMUProjekt {
  [User]                    read Anforderungen,
                            read Prototyp;
  [Analyse & Kunde]         write Anforderungen,
                            exec  entwerfen;
  [Entwurf & Entwickler]    write EMUModelle,
                            exec  analysieren;
}
```

Mit der ersten Regel wird festgelegt, dass jeder Benutzer (`User`) – egal ob `Kunde` oder `Entwickler` – zu jeder Phase – egal ob `Analyse` oder `Entwurf` – auf `Anforderungen` und `Prototyp` lesend zugreifen kann. Der boolesche Ausdruck `User` ist in den EMU-Eingabemodellen gleichbedeutend mit `true`.

Der lesende Zugriff auf eine Komponenten mit Basistyp bedeutet, dass dem Benutzer der Wert lesend visualisiert wird. Der lesende Zugriff auf Komponenten mit *komplexen* Typ bedeutet, dass die implizit definierten Interaktionen (vgl. Abschnitt 4.3) nicht ausführbar sind. Jedoch können die explizit definierten Interaktionen (also die in Abschnitt 5.1.1 definierten Aktionen) von Komponenten, auf die lesend zugegriffen werden kann, ausgeführt werden, wenn der Zugriff auf den entsprechenden Tupeltyp für den jeweili-

gen Benutzer auf ausführbar (**exec**) gesetzt wurde und die Vorbedingung der jeweiligen Aktion gilt.

Der lesende Zugriff auf einen Knoten eines EMU-Systems ist also eine notwendige, aber keine hinreichende Voraussetzung für die Ausführbarkeit der im Aufgabenmodell für den entsprechenden Knotentyp explizit definierten Aktionen.

Der Kunde kann während der Analysephase auf die Komponente **Anforderungen** schreibend zugreifen sowie die Aktion **entwerfen** ausführen (und, wie oben erklärt, **Prototyp** lesen).

Während der Entwurfsphase wird der Zugriff des Kunden nur durch die erste Regel festgelegt, weil nur deren Anwendbarkeitsbedingung zutrifft. Während der Analysephase trifft Ähnliches für den Entwickler zu, d.h. er kann lediglich gemäß der ersten Zugriffsregel auf die Komponenten **Anforderungen** und **Prototyp** lesend zugreifen.

Während der Entwurfsphase kann der Entwickler gemäß der dritten Zugriffsregel die Komponente **EMUModelle** schreibend bearbeiten und die Aktion **analysieren** ausführen, womit – gemäß des Aktivitätsnetzes – erneut die Analysephase eingeleitet wird.

Die Auswertung der Anwendbarkeitsbedingung einer Zugriffsregel ist genauer zu betrachten. Hier liegt ein Beispiel für den in Abschnitt 5.1.3 angesprochenen Zugriff auf die Benutzereigenschaften über den imaginären *User*-Knoten vor. Dabei handelt es sich um die Eigenschaft eines Benutzers, „der Kunde zu sein".

Informell betrachtet, ist der boolesche Ausdruck **[Analyse & Kunde]** genau dann wahr, wenn sich das EMU-Projekt in der Phase **Analyse** befindet und der aktuelle Benutzer der Kunde ist. Formal betrachtet, erfolgt die Auswertung des Ausdrucks an zwei Positionen. Die erste Position ist diejenige, die mit dem IDT markiert ist, für den das Zugriffsrecht definiert wird. In diesem Fall handelt es sich dabei um die Wurzelposition $\langle\rangle$, weil der IDT **EMUProjekt**, für den das Zugriffsrecht definiert wird, der Starttyp des Datenmodells ist. Wir sprechen in diesem Zusammenhang von der ***Datenposition***. Die zweite Position, an welcher der Ausdruck ausgewertet wird, ist die Position des Benutzers, die wir deshalb ***Benutzerposition*** nennen. Im Beispiel gibt es die Benutzerpositionen $\langle 1\rangle$ und $\langle 2\rangle$ (**Kunde** ist 1.Komponente vom Starttyp **EMUProjekt** und **Entwickler** ist 2.Komponente von **EMUProjekt**).

Die Semantik von booleschen Ausdrücken wird daher in Abschnitt 5.3.2 als eine Funktion mit den Argumenten Daten- und Benutzerposition definiert.

Zu beachten ist dabei eine mögliche Mehrdeutigkeit, die beispielsweise dann entsteht, wenn der IDT, für den die Zugriffsrechte definiert wurden, eine zu einem Benutzertyp gleichnamige Komponente besitzt. Solche Mehrdeutigkeiten werden entweder durch EMUGEN implizit zugunsten der Datenposition aufgelöst oder explizit vom Entwickler durch Voranstellung des Schlüsselworts **User** zugunsten der Benutzerposition.

Zu Beispiel 5-1 gehört noch die Zugriffsdefinition auf **Prototyp**:

```
access on Prototyp {
   [Kunde]                  exec ausfuehren,
                            read Anwendungsbeschreibung;
   [Entwickler]             exec ausfuehren,
                            exec generieren;
```

```
                                write Anwendungsbeschreibung;
      [Fehler & Entwickler]     read Fehlerbeschreibung;
   }
```

Der Kunde ist also berechtigt, die Aktion `ausfuehren` auszuführen und auf die Komponente `Anwendungsbeschreibung` lesend zuzugreifen. Der Entwickler kann die Aktionen `ausfuehren` und `generieren` ausführen und die Komponente `Anwendungsbeschreibung` beschreiben. Wenn ein Fehler aufgetreten ist (also wenn die Phase `Fehler` aktiv ist), dann kann der Entwickler zusätzlich die Komponente `Fehlerbeschreibung` lesen.

### 5.1.5 Implizite Ergänzungen

Im Aufgaben- und Benutzermodell gibt es für jeden Benutzertyp implizit definierte Teile. Erfolgte im Benutzermodell keine Zugriffsdefinition für ein Tupel, so erfolgt von EMUGEN eine implizite Ergänzung des Benutzermodells, wodurch definiert wird, dass für jeden Benutzer alle Komponenten und Aktionen auf den zugehörigen Tupelknoten schreib- bzw. ausführbar sind.

Ferner wird für jede Tupelproduktion eines Benutzertyps des Benutzermodells die zusätzliche Komponente `Login` hinzugefügt, die selbst durch eine Tupelproduktion definiert wird. Zu `Login` werden implizit die beiden Aktionen `anmelden` und `abmelden`, ein implizit definiertes Aktivitätsnetz (wir sprechen vom ***Benutzeraktivitätsnetz***) und eine Zugriffsdefinition festgelegt.

Damit wird definiert, wie ein Benutzer sich am EMU-System anmeldet und damit als aktiver Benutzer den Zustand des EMU-Systems gemäß seinen Zugriffsmöglichkeiten bearbeiten kann. Für Beispiel 5-1 wurde in Abschnitt 5.1.3 festgelegt, dass beide Benutzer sich mit ihrem Namen anmelden sollen. Daher ergeben sich folgende, von EMUGEN implizit definierten (also für systeminterne Zwecke erzeugten) Produktionen.

```
Login::=Anmeldedaten Fehlerbeschreibung
Anmeldedaten::= Anmeldedaten.Kunde | Anmeldedaten.Entwickler
Anmeldedaten.Kunde::=String:Name
Anmeldedaten.Entwickler::=String:Name

actions of Login {
  anmelden  [true] -> {:
       /*Kommandos zur Impl. des Anmeldevorgangs */
  :} Ok | Fehler;
  abmelden  [true] -> {::};
}
actnet of Login {
  -> Passiv
  Passiv    ->anmelden.Ok     ->Aktiv
  Passiv    ->anmelden.Fehler ->Fehler
  Fehler    ->anmelden.Ok     ->Aktiv
  Aktiv     ->abmelden        ->Passiv
}
access on Login {
  [Passiv]        write Anmeldedaten,
                  exec  anmelden;
```

```
    [Fehler]        read  Fehlerbeschreibung;
    [Aktiv]         exec  abmelden;
  }
  access on Kunde {
   /* evtl. explizite Definitionen */
    [Kunde==this]       write Login;
  }
  access on Entwickler {
   /* evtl. explizite Definitionen */
    [Entwickler==this]  write Login;
  }
```

Der hier implizit dargestellte IDT **Anmeldedaten** definiert die bei der Anmeldung einzugebenden Daten (vgl. Abbildung 5-8). Im Aktivitätsnetz zu **Login** (Benutzeraktivitätsnetz) wird festgelegt, dass sich ein Benutzer zunächst „in der Phase" **Passiv** „befindet"[8] und mit **anmelden** in die Phase **Aktiv** wechseln kann, wenn das Kommando von **anmelden** das Resultat **Ok** liefert. Von **Aktiv** kommt man mit **abmelden** zurück nach **Passiv**. Im Fehlerfall wird dem Benutzer eine Fehlerbeschreibung (z.B. „falscher Name") mitgeteilt.

Der „eigene Knoten" ist der einzige Knoten, auf den auch ein passiver Benutzer Zugriff hat. Der hier auftretende Pfadausdruck **this** steht für die aktuelle Datenposition $\langle\rangle$. Er ist an den Bezeichner für die Selbstreferenz in Java [GJS96] oder C++ [Str92] angelehnt.

Der Entwickler kann für die einzelnen Benutzertypen auch explizit ein Aktivitätsnetz definieren, welches orthogonal (im Sinne von unabhängig) vom implizit definierten Benutzeraktivitätsnetz existiert. Aus technischer Sicht bedeutet das lediglich, dass der Entwickler den reservierten Bezeichner **Login** nicht verwenden darf.

### 5.1.6 Graphische Darstellung durch TUMs

Es ist auf verschiedene Weise möglich, den bisher textuell betrachteten Informationsgehalt des Aufgaben- und Benutzermodells – zumindest teilweise – graphisch darzustellen. Beispielsweise könnte man dazu *Anwendungsfall-* und *Aktivitätsdiagramme* von UML [OMG01] verwenden. Das Anwendungsfalldiagramm in Abbildung 5-3 visualisiert die Benutzung auf eine sehr abstrakte Weise. Es zeigt lediglich, dass der Kunde beim Anwendungsfall (UML-Terminologie [OMG01]) **Analyse** und der Entwickler beim Anwendungsfall **Entwurf** beteiligt ist.

Das Aktivitätsdiagramm in Abbildung 5-4 visualisiert die Ablauflogik des EMU-Projekts. Der Ablauf startet mit der *Aktivität* [OMG01] **Analyse** und geht anschließend in die Aktivität **Entwurf** über. Ein Endzustand wurde der Vollständigkeit halber hinzugefügt. Im Aktivitätsdiagramm wird nicht dargestellt, wie die Benutzer in den Ablauf eingreifen können und auf welche Weise das Schalten der Aktivitäten vonstatten geht.

---

[8] Bei Benutzeraktivitätsnetzen oder bei Aktivitätsnetze, die Dialoge beschreiben, wäre es üblicher, anstelle von *Phasen* von *Zuständen* zu sprechen. Ein solches Homonym würde jedoch die weitere Erklärung der Modellierungssprache verkomplizieren.

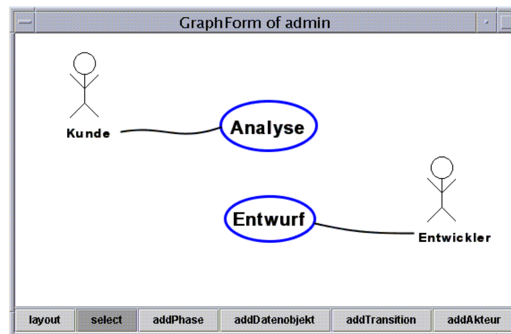


**Abbildung 5-3: Anwendungsfalldiagramm zu *EMUProjekt***

Zur graphischen Visualisierung der Aspekte des Aufgaben- und Benutzermodells, wir sprechen daher von einem ***TUM***, einem ***Task-User-Model***, können die graphischen Sprachelemente des Anwendungsfall- und Aktivitätsdiagramms vereinigt werden, wobei zusätzlich Aktionen als eigene Knoten visualisiert werden.

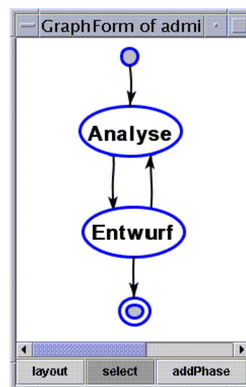


**Abbildung 5-4: Aktivitätsdiagramm zu *EMUProjekt***

Das TUM[9] von Abbildung 5-5 zeigt den gleichen Informationsgehalt, wie das Aktivitätsnetz und die Zugriffsdefinition zu `EMUProjekt`, die in den Abschnitten 5.1.2 und 5.1.4 textuell angegeben wurden.

Die Komponenten und Aktionen werden in einem TUM durch Rechtecke, die Phasen durch Ovale (wobei die Startphasen grau markiert sind) und die Benutzereigenschaften durch Strichmännchen dargestellt. Bei den Benutzereigenschaften handelt es sich hier um die Zugehörigkeit zu einem Benutzertyp (vgl. den Zugriff über den User-Knoten in Abbildung 5-2). In manchen CADUI-Werkzeugen [PCO99] ist dies sogar die einzige Möglichkeit, Benutzereigenschaften zu modellieren. Wir werden in Anlehnung an die Anwendungsfallsdiagramme bei der graphischen Präsentation von Benutzereigenschaften von ***Akteuren*** sprechen.

Im Gegensatz zu Anwendungsfall- oder Aktivitätsdiagrammen zeigt ein TUM ein konstruktives (d.h. ausführbares) Modell eines interaktiven (Teil-)Systems mit mehreren

[9] Das dargestellte TUM wurde mit einem Grapheditor erstellt, der mit EMUGEN selbst generiert wurde. Die Spezifikation des TUM-Grapheditors wird in Abschnitt 6.2.1 beschrieben.

Benutzern, wenn dazu passende IDTs (Datenmodell), Aktionen (Aufgabenmodell) und Benutzertypen (Benutzermodell) vorhanden sind. Im Allgemeinen handelt es sich nur um einen Teil des Gesamtsystems, weil ein TUM genau *einem* IDT des Datenmodells zugeordnet wird. Das TUM von Abbildung 5-5 ist beispielsweise dem IDT `EMUProjekt` zugeordnet.

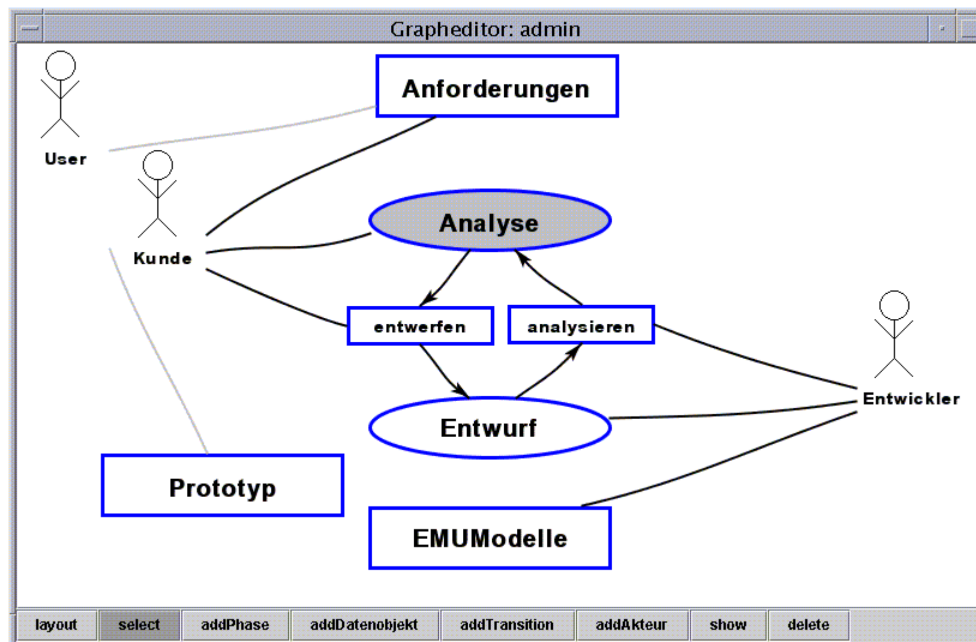


**Abbildung 5-5: Ein TUM zu `EMUProjekt`**

Das TUM von Abbildung 5-5 ist folgendermaßen zu lesen. Jeder Benutzer (`User`) hat lesenden Zugriff (graue Kante) auf die Komponenten `Anforderungen` und `Prototyp`. Während der Phase `Analyse` hat ein Benutzer mit der Eigenschaft, ein Kunde zu sein, schreibenden Zugriff (schwarze Kante) auf die Komponente `Anforderungen` und kann die Aktion `entwerfen` ausführen. Während der Phase `Entwurf` hat ein Benutzer mit der Eigenschaft, ein Entwickler zu sein, schreibenden Zugriff auf die Komponente `EMU-Modelle` und kann die Aktion `analysieren` ausführen.

Die in der textuellen Darstellung als Konjunktion von Phase und Benutzertyp dargestellten Anwendbarkeitsbedingungen (z.B. `[Analyse & Kunde]`) von Zugriffsregeln werden also durch Kanten zwischen den entsprechenden graphischen Knoten der Phase und dem Akteur dargestellt.

Besteht eine Anwendbarkeitsbedingung nur aus einer Benutzereigenschaft (wie z.B. `[User]`), so wird der Akteur ohne Verbindung zu einer Phase dargestellt. Kommt in der gesamten Zugriffsdefinition keine Benutzereigenschaft vor (wie z.B. im implizit definierten Benutzeraktivitätsnetz in Abschnitt 5.1.5), so kann auf den User-Akteur verzichtet und die Phase direkt mit der Komponente bzw. Aktion verbunden werden (vgl. Abbildung 5-7).

Ohne Informationsverlust ist diese Art der Visualisierung einer Anwendbarkeitsbedingung offensichtlich nur dann möglich, wenn die Anwendbarkeitsbedingung aus einer Konjunktion eines Phasenbezeichners und einer Benutzereigenschaft oder einer dieser

beiden Teile besteht, während in der textuellen Darstellung ein beliebiger boolescher Ausdruck möglich ist.

Bei der Diskussion verschiedener Beispiele im nächsten Kapitel wird sich zeigen, dass diese Einschränkung bzgl. der Modellierung der Zugriffe nicht weiter von Bedeutung ist, sondern vielmehr zu übersichtlicheren Aktivitätsnetzen führt.

Umgekehrt ist es immer möglich, ein TUM auf ein Aktivitätsnetz und eine Zugriffsdefinition abzubilden.

Soll dennoch eine komplexe Anwendbarkeitsbedingung verwendet werden, so kann diese im TUM entweder durch eine sogenannte *virtuelle Phase* oder einen *virtuellen Akteur* dargestellt werden, je nachdem, ob sie mehr eine Phase oder einen Akteur beschreibt. Diese virtuellen Elementen bestehen jeweils aus einem Bezeichner und einer booleschen Bedingung. Virtuelle Phasen dürfen sich in keinem Vorbereich einer Transition befinden, da sie nicht deaktiviert werden können.

Abbildung 5-6 zeigt das TUM für `Prototyp` und visualisiert damit den Zugriff des Kunden und des Entwicklers auf diese Teilkomponente. Es zeigt, dass der Kunde immer lesenden Zugriff auf die Komponente `Anwendungsbeschreibung` hat und die Aktion `ausfuehren` immer ausführen kann.

Dabei ist zu beachten, dass weder durch ein TUM noch durch die textuelle Zugriffsdefinition die Vorbedingungen von Aktionen berücksichtigt werden, sondern lediglich der Zugriff der Benutzer auf die Komponenten und Aktionen modelliert wird. Genauer müsste man also sagen, dass der Kunde immer auf die Aktion `ausfuehren` zugreifen und sie damit potentiell – je nach der Gültigkeit der Vorbedingung im aktuellem Zustand – ausführen kann.

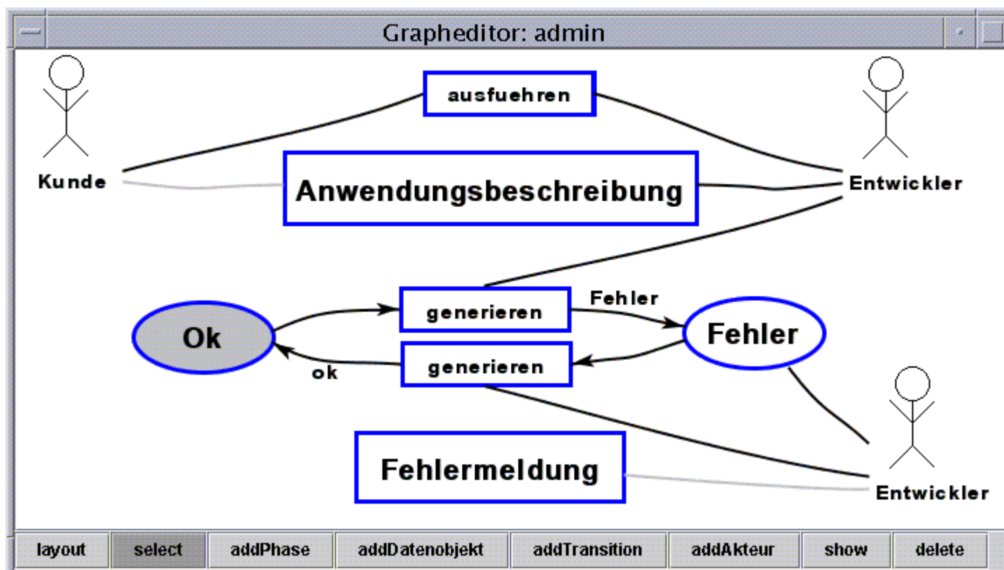


**Abbildung 5-6: Ein TUM zu `Prototyp`**

Der Entwickler kann immer die Komponente `Anwendungsbeschreibung` beschreiben und auf die Aktionen `generieren` und `ausfuehren` zugreifen.

Abbildung 5-7 zeigt das TUM für das implizit definierte Benutzeraktivitätsnetz. Da an einem Benutzeraktivitätsnetz nur genau ein Benutzer arbeitet, wird kein Akteur explizit eingezeichnet. Initial ist jeder Benutzer passiv, daher ist `Passiv` als Startphase grau markiert. Ein passiver Benutzer kann konzeptionell immer die Anmeldedaten ausfüllen (d.h. auf `Anmeldedaten` schreibend zugreifen) und die Aktion `anmelden` ausführen. Entsteht ein Fehler bei der Ausführung des Kommandos von `anmelden`, etwa weil die Anmeldedaten nicht mit den Daten des Benutzers (in Beispiel 5-1 handelt es sich nur um den Benutzernamen) übereinstimmen, wird sowohl die Phase `Passiv` (zur Neueingabe der Anmeldedaten) als auch die Phase `Fehler` (zur Darstellung der Fehlerbeschreibung) eingenommen. War das Anmeldekommando erfolgreich, so wird (ausschließlich) die Phase `Aktiv` eingenommen.

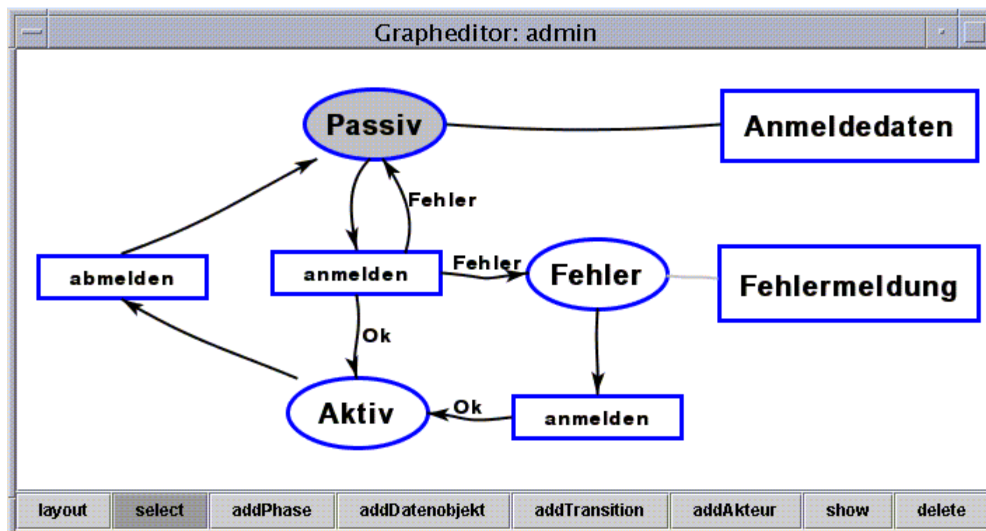


**Abbildung 5-7: Ein TUM zum impliziten Benutzeraktivitätsnetz**

Man sieht am Benutzeraktivitätsnetz die Möglichkeiten des EMU-Aufgabenmodells, einfache Dialoge in der üblichen graphischen Art und Weise [Den91, Eic98] zu modellieren. Dadurch, dass mehrere Phasen gleichzeitig aktiv sein können (hier etwa die Phasen `Passiv` und `Fehler`), erhält man eine sehr kompakte Darstellung. Dies liegt an der Petrinetz-ähnlichen Semantik der Aktivitätsnetze.

## 5.2 Beispielausführungen generierter EMU-Systeme

Wie in Abschnitt 4.2 erfolgt nun eine informelle Beschreibung einer Beispielausführung eines generierten EMU-Systems, wobei es sich nun um ein Mehrbenutzersystem handelt. Ein EMU-System vom Typ `EMUProjekt` ist ein Mehrbenutzersystem mit Aufgaben, an dem neben den beiden explizit modellierten Benutzern `Kunde` und `Entwickler` auch der Administrator, der als implizit immer vorhandener Benutzer betrachtet werden kann, arbeitet. Als konkrete Benutzungsoberfläche zeigen wir bei diesem Beispiel aus Übersichtlichkeitsgründen nur die Formulare und nicht die Grapheditoren der einzelnen Benutzer.

Der Beispielablauf startet mit dem Initalzustand des EMU-Systems vom Typ `EMUProjekt` (Zustand (1) in Abbildung 5-8). Hier ist das Formular des Administrators eingezeichnet. In den nachfolgenden Zuständen wird es zur besseren Übersicht weggelassen.

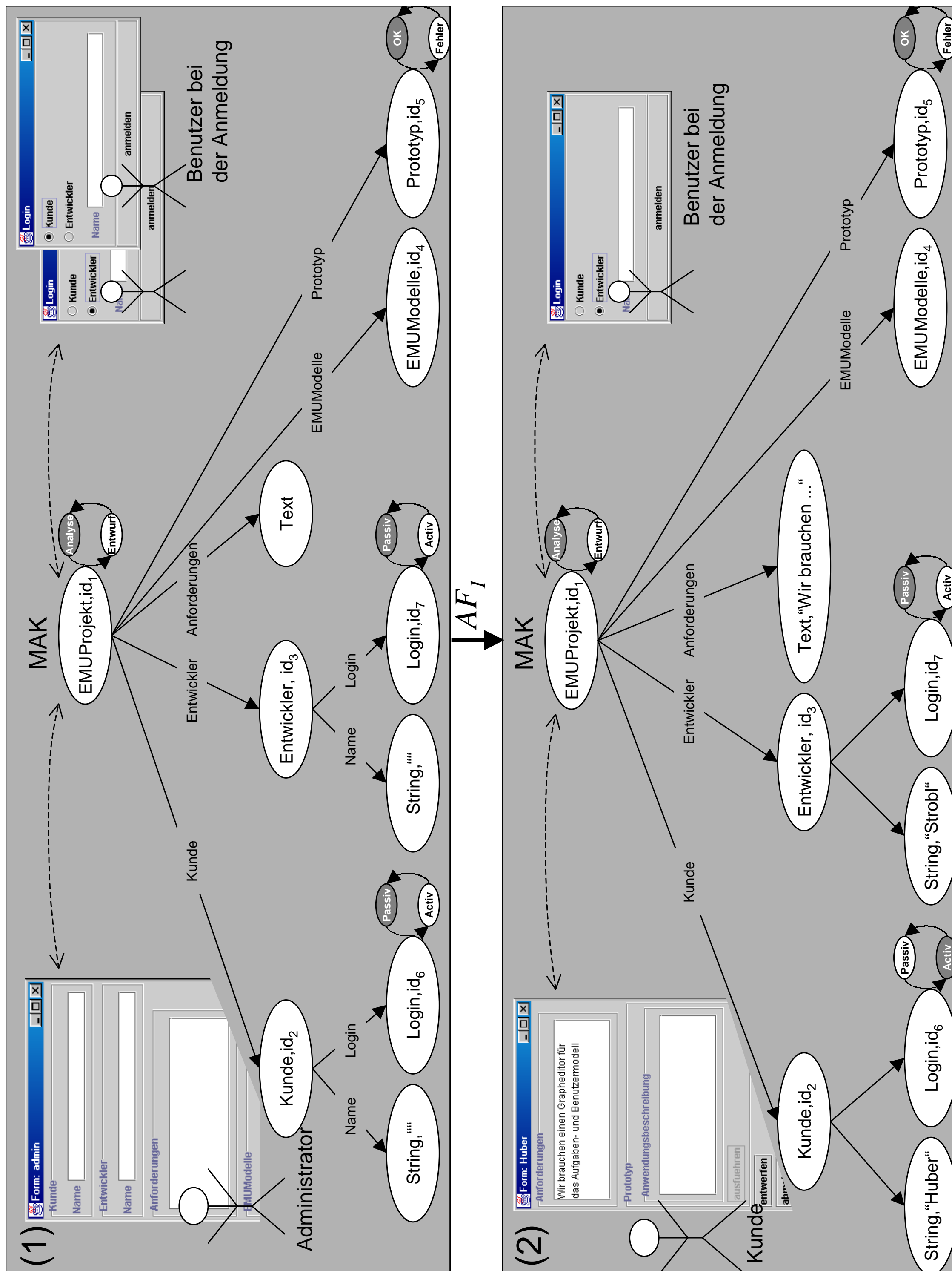


**Abbildung 5-8: Ausführung eines Mehrbenutzer-EMUSystems (1)**

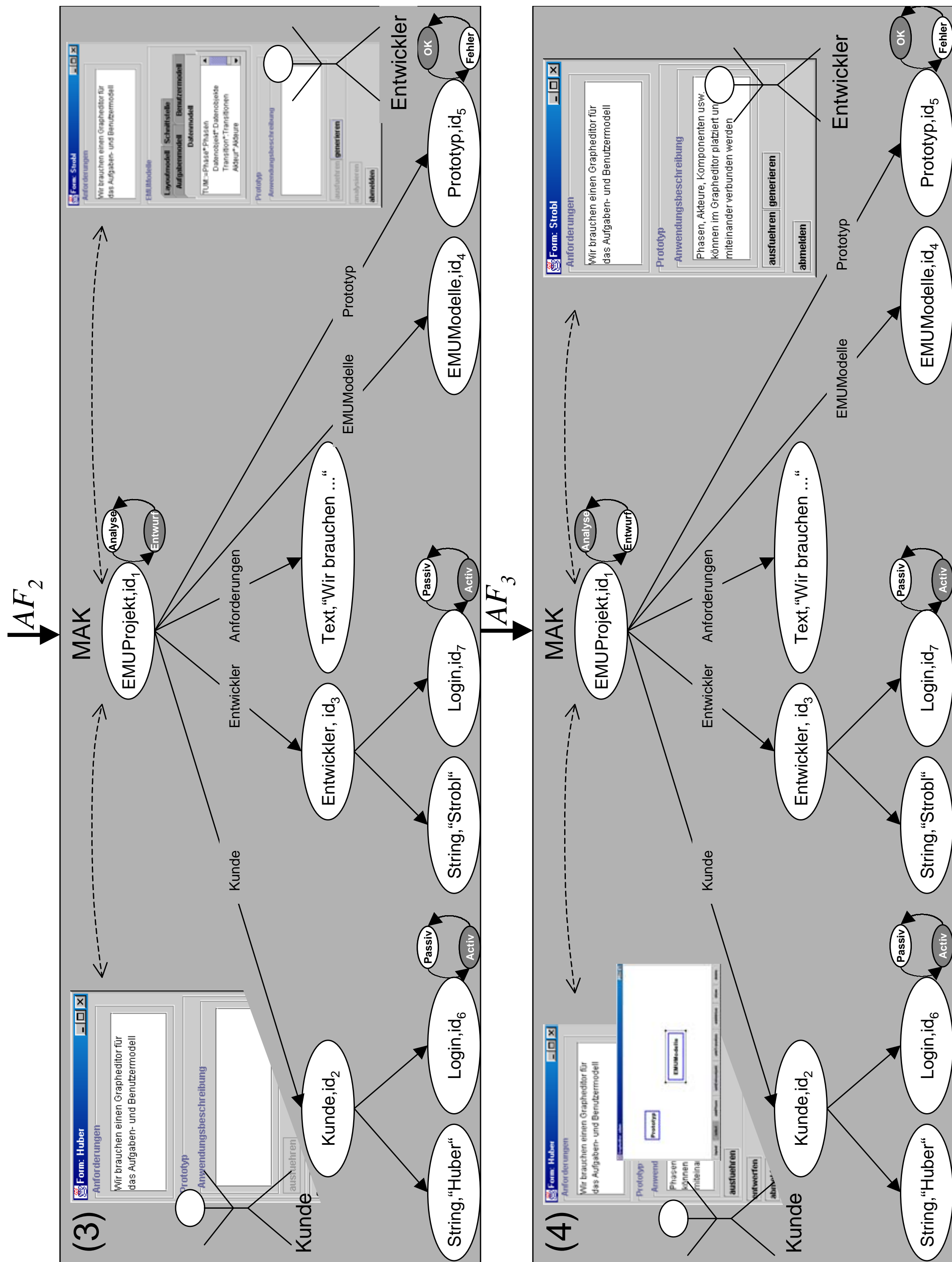


**Abbildung 5-9: Ausführung eines Mehrbenutzer-EMUSystems (2)**

Bei der Skizzierung des MAKs werden an den entsprechenden Knoten die Aktivitätsnetze mit ihren aktuellen Markierungen eingezeichnet. Dadurch soll gezeigt werden, dass zur Laufzeit an bestimmten Knoten intern Aktivitätsnetze mit ihrer aktuellen Markierung verwaltet werden. Aus Platzgründen wurden bei den Benutzeraktivitätsnetzen nur die Phasen **`Passiv`** und **`Aktiv`** eingezeichnet.

Vom Initialzustand gelangt das EMU-System durch die Aktionsfolge $AF_1$ in den Zustand (2). Die Bedeutung der nun (bei Mehrbenutzersystemen) erweiterten, systeminternen Aktionscodierung wird in Abschnitt 5.2.1 erklärt. Die wichtigste Erweiterung besteht aus der Position des Benutzers, der die Aktion ausführt (dritte Komponente, $\langle\rangle$ beim Administrator).

$$\begin{aligned} AF_1 = \ & \langle \widetilde{setValue}(\langle 1,1\rangle, "Huber", \langle\rangle), \\ & \widetilde{setValue}(\langle 2,1\rangle, "Strobl", \langle\rangle), \\ & \widetilde{exec}(\langle 1,2\rangle, anmelden, \langle 1\rangle), \\ & \widetilde{setValue}(\langle 3\rangle, "\text{Wir brauchen ...}", \langle 1\rangle)\rangle \end{aligned}$$

In $AF_1$ werden die Namen des Kunden und des Entwicklers vom Administrator eingegeben. Der Kunde meldet sich am EMU-System an und schreibt seine Anforderungen auf. D.h. der Wert der Komponente **`EMUProjekt.Anforderungen`** (Position $\langle 3\rangle$) wird vom Benutzer mit der Position $\langle 1\rangle$ auf einen neuen Wert gesetzt.

Im Zustand (2) in Abbildung 5-8 ist nur der Kunde aktiv. Man beachte die Markierung des Aktivitätsnetzes der **`Login`**-Komponente (vgl. Abschnitt 5.1.5) des Kunden im MAK (Position $\langle 1,2\rangle$). Da nur der Kunde aktiv ist, wird auch nur ein benutzerspezifisches Formular konstruiert. Man vergleiche das im Zustand (2) gezeigte Kunden-Formular mit dem TUM für **`EMUProjekt`** in Abbildung 5-5 und **`Prototyp`** in Abbildung 5-6. Es wird lediglich das Teilformular für **`Anforderungen`** und **`Prototyp`** dargestellt. Der Kunde kann die Aktionen **`entwerfen`** und **`abmelden`** ausführen.

Vom Zustand (2) wird durch die Aktionsfolge $AF_2$ in den Zustand (3) (Abbildung 5-9) gewechselt.

$$\begin{aligned} AF_2 = \ & \langle \widetilde{exec}(\langle 2,2\rangle, anmelden, \langle 2\rangle), \\ & \widetilde{exec}(\langle\rangle, entwerfen, \langle 1\rangle), \\ & \widetilde{setValue}(\langle 4,1\rangle, "\text{TUM::=...}", \langle 2\rangle)\rangle \end{aligned}$$

In $AF_2$ meldet sich der Entwickler an und der Kunde startet durch Ausführung der Aktion **`entwerfen`** die Phase **`Entwurf`**. Ferner gibt der Entwickler ein Datenmodell ein. Genauer: der Entwickler gibt an der Position $\langle 4,1\rangle$, die zur besseren Übersichtlichkeit in Abbildung 5-8 und Abbildung 5-9 nicht dargestellt ist, einen neuen Wert ein.

Da in Zustand (3) die Phase **`Entwurf`** aktiv ist, wird dem Entwickler (gemäß der Zugriffsdefinition) ein Formular präsentiert, bei dem die Anforderungen gelesen und die EMU-Modelle beschrieben werden können.

Von Zustand (3) wird durch die Aktionsfolge $AF_3$ in den Zustand (4) von Abbildung 5-9 gewechselt.

$$\begin{aligned} AF_3 = \; & \langle \widetilde{exec}(\langle 5\rangle, generieren, \langle 2\rangle), \\ & \widetilde{setValue}(\langle 5,2\rangle, \text{"Phasen,..."}, \langle 2\rangle), \\ & \widetilde{exec}(\langle 5\rangle, analysieren, \langle 2\rangle), \\ & \widetilde{exec}(\langle 5\rangle, ausfuehren, \langle 1\rangle)\rangle \end{aligned}$$

In $AF_3$ erstellt der Entwickler durch die Aktion `generieren` den Prototyp. Ferner wird vom Entwickler die Anwendungsbeschreibung (Position $\langle 5,2\rangle$) erstellt und durch Ausführung der Aktion `analysieren` erneut die Analysephase eingeleitet. Der Kunde führt den Prototyp durch Ausführung der Aktion `ausfuehren` aus (vgl. Zustand (4)).

Man beachte die Veränderung der Markierung der Aktivitätsnetze. Die Markierung eines Benutzeraktivitätsnetzes ändert sich beim Anmelden des entsprechenden Benutzers. Das Aktivitätsnetz von `EMUProjekt` an der Wurzel ändert sich bei Ausführung der Aktionen `entwerfen` und `analysieren` (vgl. das TUM von `EMUProjekt` in Abbildung 5-5), also beim Übergang von Zustand (2) nach (3) und von (3) nach (4).

In Zustand (4) ist die Phase `Ok` am Prototyp (Position $\langle 5\rangle$) markiert. Daraus kann man erkennen, dass die Ausführung des Kommandos der Aktion `generieren` (erstes Element von $AF_3$) als Resultat `Ok` lieferte, womit auch die Vorbedingung für die Aktion `ausfuehren` erfüllt ist.

### 5.2.1 Abstrakte Ausführung

Es folgt nun eine informelle Beschreibung der Erweiterungen, die zur Ausführung eines Mehrbenutzersystems mit Aktionen und Aktivitätsnetzen, also zusätzlich zu den in Abschnitt 4.2 (zur Ausführung eines Einbenutzersystems) besprochenen Konzepten, notwendig sind.

#### Zustände

Beim Zustandsbegriff sind folgende Erweiterungen notwendig:

- An jedem Knoten, für dessen Typ ein Aktivitätsnetz explizit oder implizit (Benutzeraktivitätsnetz, vgl. Abschnitt 5.1.5) definiert wurde, ist ein Aktivitätsnetz mit einer aktuellen Markierung zu verwalten.
- An jedem Knoten, dessen Typ Aktionen zugeordnet wurden, ist eine Menge von Aktionen und die Gültigkeit ihrer Vorbedingungen zu verwalten.
- Für jeden Benutzer ist ein Zugriffsadapter (vgl. Abschnitt 3.2.2) zu verwalten. Formal besteht ein Zugriffsadapter eines Benutzer aus genau der Teilmenge aller Knoten des MAKs (zum Mengencharakter des MAKs siehe Abschnitt 4.3), auf die der jeweilige Benutzer lesend zugreifen darf und der Menge der Aktionen des MAKs, die der Benutzer ausführen darf. Der Zugriffsadapter bildet die Applikationsschnittstelle der konkreten Benutzungsoberflächen der einzelnen Benutzer im Sinne des Seeheim-Modells (vgl. Abbildung 3-4).

#### Initialzustand

Beim Initialzustand sind die Erweiterungen durch die Erweiterungen des Zustandsbegriffs vorgegeben. D.h. an den entsprechenden Knoten werden Aktionen und die Initial-

markierung des zugehörigen Aktivitätsnetz (d.h. die als Startphasen definierten Phasen werden aktiviert) initialisiert.

**Aktionen**

Die in Abbildung 5-8 und Abbildung 5-9 dargestellten Aktionen zeigen zwei Erweiterungen, die am Beispiel der Aktion $\widetilde{exec}(\langle 5\rangle, generieren, \langle 2\rangle)$ diskutiert werden können.

- Es gibt den zusätzlichen Aktionsbezeichner *exec*, der für die Ausführung einer explizit definierten Aktion steht (hier: *generieren* an der Position $\langle 5\rangle$ ).
- Bei einer Mehrbenutzeranwendung wird die Position des Benutzers, der die Aktion ausführt, als dritte Komponente mitgeliefert (hier: die Position $\langle 2\rangle$). Handelt es sich um den implizit definierten Benutzer *Administrator*, so wird die Position $\langle\rangle$ verwendet. Die Benutzerposition kann im Kommando der Aktion verwendet werden, was in einigen Fällen nützlich sein kann.

Aufgrund dieser Erweiterungen werden die Aktionsbezeichner (die wieder in gleicher Weise auch im formalen Abschnitt verwendet werden) beim Mehrbenutzersystem mit $\widetilde{\quad\quad}$ markiert.

**Zustandsübergang**

Beim Zustandsübergang ist bei der Ausführung einer explizit definierten Aktion die Auswirkung der Kommandoausführung zu berücksichtigen. Die Kommandoausführung kann zu einer Zustandsänderung des gesamten MAKs führen. Nach einer Kommandoausführung kann es ferner zum Schalten einer Transition in einem Aktivitätsnetz kommen.

Aufgrund der Erweiterungen beim Zustandsbegriff, wo durch die Verwaltung der Zugriffadapter eine komplexe Komponente hinzukommt, erscheint es notwendig, auch beim Zustandsübergang entsprechend weitere Elemente hinzuzufügen, etwa um den „Zustandsübergang der Zugriffsadapter“ zu definieren. Dies vermeiden wir, indem wir die Zugriffsadapter ausschließlich durch die Zustände und nicht durch die Übergänge definieren. Man kann dieses Vorgehen mit der zustands- und nicht zustandsübergangsorientierten Spezifikation des Layouts bzw. der Layout-Richtlinien bei BOSS [Sch97] vergleichen.

### 5.2.2 Ausführung mit Formularen

Ein Benutzer kann über ein eigenes Formular auf genau den Teil des MAKs zugreifen, der ihm durch den zugehörigen Zugriffsadapter gewährt wird. Aufgrund der Verwendung dynamischer Formulare, deren Layoutberechnung mit (Standard-)Layoutregeln erfolgt, die den einzelnen IDTs bzw. den Metatypen zugeordnet werden (vgl. Abschnitt 4.2.2), ergibt sich in vielen Fällen ein zufriedenstellendes Layout, auch wenn einzelne Teilkomponenten nicht angezeigt werden, weil aktuell kein Zugriffsrecht dafür vorhanden ist. Beispielsweise wird im Zustand (2) in Abbildung 5-8 im Kunden-Formular kein Teilformular für die Komponente `EMUModelle` angezeigt.

### 5.2.3 Ausführung mit Grapheditoren

Die Verwendung von Grapheditoren ist beim Beispiel `EMUProjekt` zwar prinzipiell möglich, jedoch nicht sonderlich hilfreich. Es würden lediglich – gemäß den in Abschnitt 4.2.3 formulierten Regeln – die Knoten mit den Positionen $\langle\rangle, \langle 1\rangle, \langle 2\rangle, \langle 4\rangle$ und $\langle 5\rangle$ als graphische Knoten visualisiert. Es könnten keine graphischen Knoten neu hinzugefügt werden, weil keine Listen verwendet werden. Da es bei dem Beispiel auch keine Referenzen gibt, wären auch keine graphischen Kanten im Grapheditor darzustellen.

Eine sinnvolle Integration von Grapheditoren in eine Mehrbenutzeranwendung wird in Kapitel 6 gezeigt, wenn es im Beispiel *Ad-hoc Workflows* (Abschnitt 6.2.5) darum geht, ein EMU-System zu erstellen, welches die Definition von Aufgaben und ihren kausalen Abhängigkeiten zur Laufzeit ermöglicht.

In Vorbereitung dafür soll zunächst das Beispiel 4-2 *Petrinetze* um eine Simulationsmöglichkeit erweitert werden. Dieses, noch nicht mehrbenutzerfähige, Beispiel soll auch einen Einblick in die technische Umsetzung einer konkreten Problemstellung bei der Entwicklung eines Diagrammspracheditors geben. Daher erfolgt auch die Darstellung des notwendigen Java-Quellcodes für die Aktionskommandos.

**Beispiel 5-2: *Simulationswerkzeug für Petrinetze***

> *Das bereits vorhandene EMU-System für binäre Petrinetze soll um eine Simulationsmöglichkeit erweitert werden. Dabei ist wie üblich eine Transition genau dann schaltbereit, wenn alle Stellen im Vorbereich markiert sind. Es soll eine automatische Simulation möglich sein, bei der in jedem Schritt zufällig eine Transition gewählt wird. Die zufällig gewählte Transition kann nur schalten, wenn sie schaltbereit ist. Ferner soll eine Simulation per Hand möglich sein. Dabei wählt der Benutzer eine schaltbereite Transition durch Selektion im Grapheditor aus und schaltet sie an der Benutzungsoberfläche. Daher sollen schaltbereite Transitionen von nichtschaltbereiten im Grapheditor zu unterscheiden sein.*

Es wäre möglich, bei einem automatischen Simulationsschritt auch eine Menge nebenläufig ausführbarer [Esp95] Transitionen (und nicht nur, wie oben gefordert, eine einzelne) schalten zu lassen, um eine nebenläufige Ausführung zu simulieren. Dann wäre jedoch das entsprechende Kommando zu komplex, um es an dieser Stelle vollständig zu betrachten.

Im Datenmodell sind keine Erweiterungen zu Beispiel 4-2 notwendig. Zur Wiederholung wird es in Abbildung 5-10 in der graphischen Notation gezeigt.

Das Aufgabenmodell zu *Simulationswerkzeug für Petrinetze* besteht aus folgenden Elementen:

- Eine Aktion zum Schalten der Transitionen, die dem IDT `Transition` zugeordnet wird;
- Aktionen zum Starten und Stoppen der automatischen Simulation, die dem Starttyp `Petrinetz` zugeordnet werden;
- ein Aktivitätsnetz zur Steuerung der Simulation durch den Benutzer, welches ebenfalls dem Starttyp zugeordnet wird;

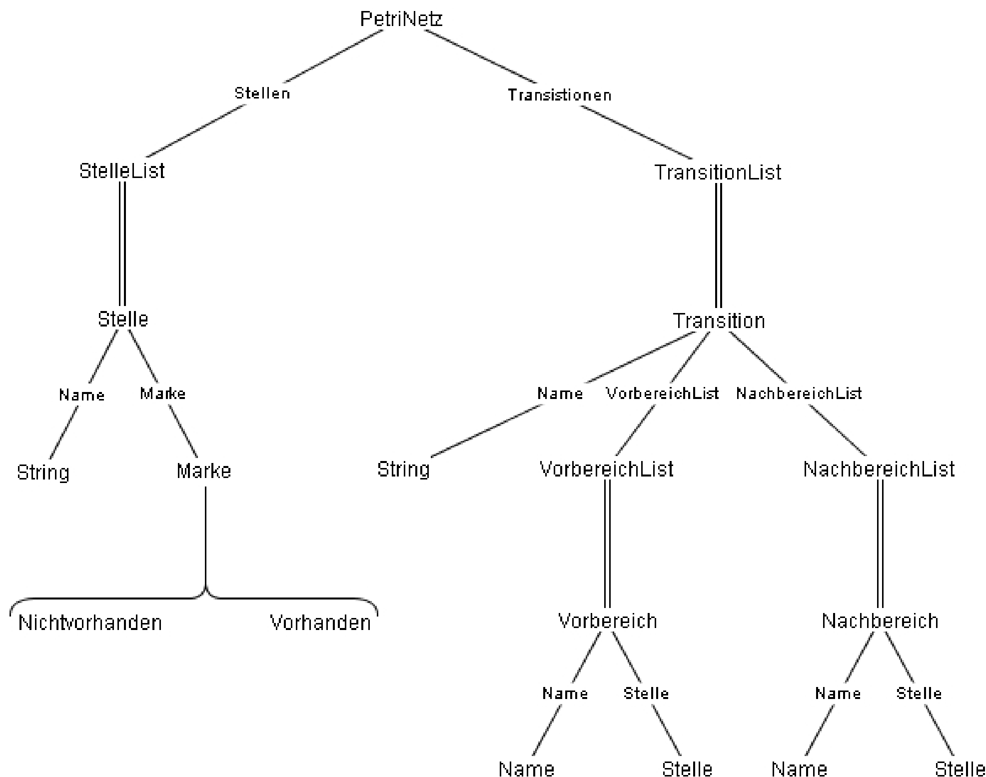


**Abbildung 5-10: Datenmodell zum Beispiel *Petrinetz***

Es ergibt sich folgendes Aufgabenmodell:

```
actions of Transition {
  schalten [schaltbereit()] () -> {:
    schalten();
  :};
}
actions of Petrinetz {
  start [Bearbeiten] () -> {:
    simulate();
  :};
  stop [Simulation] () -> {::};
}
actnet of Petrinetz {
  ->Bearbeiten
  Bearbeiten->start ->Simulation
  Simulation->stop  ->Bearbeiten
}
```

Die Vorbedingung der Aktion `schalten` stellt einen Aufruf der booleschen Instanzmethode `schaltbereit()` dar, die als Erweiterung zur automatisch erstellten API der Knotenklasse `Transition` implementiert wird. Sie liefert genau dann `true`, wenn alle Stellen des Vorbereichs markiert sind. `schalten()` ist eine weitere Instanzmethode von `Transition`, in deren Rumpf der Zustandsübergang des jeweiligen MAKs (vom Typ `Petrinetz`) beim Schalten einer Transition implementiert wird.

Die Implementierung dieser Methoden erfolgt unter Verwendung der automatisch erstellten Java-API für die Knotenklassen des EMU-Datenmodells.

```
methods of Transition {:
  public boolean schaltbereit() {
    for (int j=0;j<VorbereichList.size;j++) {
      Vorbereich v=VorbereichList.elementAt(j);
       if (!(v.value.Marke.Vorhanden))
        return false;
    }
    return true;
  }
  public void schalten() {
    for (int j=0;j<VorbereichList.size(); j++) {
      Vorbereich v=VorbereichList.elementAt(j);
      if (!v.value==null)
        v.value.Marke.setNichtvorhanden();
    }
    for (int j=0;j<NachbereichList.size;j++) {
      Nachbereich n=NachbereichList.elementAt(j);
      if (!n.value==null)
        n.value.Marke.setVorhanden();
    }
  }
:}
```

Die API für Listenklassen (vgl. Anhang C)) ermöglicht also das einfache Durchwandern von Listen und das Umschalten von Alternativen (z.B. zum Entfernen der Marke aus einer Stelle des Vorbereichs: `v.value.Marke.setNichtvorhanden()`). Bei der Implementierung einer Zustandsänderung unter Zugriff auf die Referenzwerte ist der Fall zu beachten, dass die Referenz im aktuellen Zustand nicht definiert ist.

Zum Starten der Simulation im Kommando der Aktion `start` erfolgt ein Aufruf der Methode `simulate()`, einer Instanzmethode der Knotenklasse `Petrinetz`. Dort wird die automatische Simulation implementiert, die nebenläufig zur Bearbeitung des jeweiligen EMU-Systems durch den Benutzer stattzufinden hat. Diese Nebenläufigkeit ist notwendig, weil es dem Benutzer ermöglicht werden muss, die Simulation über die Benutzungsoberfläche auch zu stoppen. Die dazu notwendige Kommunikation der Benutzungsoberfläche mit der nebenläufigen Ausführung von `simulate()` findet über die boolesche Variable `Simulation` statt, die genau dann `true` ist, wenn die entsprechende Phase aktiv ist. Wird die Phase gestoppt, was durch Ausführung der Aktion `stop` über die Benutzungsoberfläche stattfinden kann (vgl. oben das Aktivitätsnetz von `Petrinetz`), so endet die Simulation. Zur Implementierung der Simulation wird vom Entwickler ein Objekt der Klasse `java.lang.Thread` verwendet und die Methode `run` überschrieben [GJS96].

```
methods of Petrinetz {:
  private void simulate() {
    Thread simulationThread = new Thread() {
      public void run() {
        while (Simulation) {
           try {
```

```
          sleep(1000);
        } catch  (InterruptedException e) {}
         int zufallsindex= ((int)
              (Math.random()*Transitionen.size()));
         Transition t = Transitionen.elementAt(zufallsindex);
         if (t.schaltbereit())
           t.schalten();
      }
    }
  }; //Ende der Thread-Definition
  simulationThread.start();
 }
:}
```

Im Benutzermodell ist vorerst nichts weiter zu tun, weil in diesem Beispiel von einem Einbenutzersystem ausgegangen wird.

Im Layoutmodell sind die bereits bei Beispiel 4-2 erfolgten Anpassungen für das Layout des Grapheditors um die Visualisierung schaltbereiter Transitionen zu erweitern:

```
node Transition {:
  if (schaltbereit)
    FILL_PAINT=yellow;
  else
    FILL_PAINT=white;
:}
```

Damit ist ein funktionsfähiges Simulationswerkzeug für Petrinetze mit einem zugehörigen Grapheditor vollständig spezifiziert und kann mit EMUGEN generiert werden. Abbildung 5-11 zeigt zwei Zustände bei einem Beispielablauf einer Simulation. Im ersten dargestellten Zustand ist nur die Stelle *S1* markiert. Da *S1* die einzige Stelle im Vorbereich der Transition *T1* ist, ist *T1* und damit auch die zugehörige Aktion `schalten` schaltbereit. In Abbildung 5-11 wird in beiden Zuständen neben dem Formular und dem Grapheditor für den gesamten MAK auch das Formular zum Anzeigen des im Grapheditor selektierten Teilbaums dargestellt. Dieses Formular kann zur Ausführung der Aktion `schalten` verwendet werden, was in Abbildung 5-11 skizziert wird. Im zweiten dargestellten Zustand wird gezeigt, dass durch das Schalten von *T1* deren Nachstelle *S2* markiert wurde und daher nun die beiden Transitionen *T2* und *T3* schaltbereit sind, während die Transition *T1* nun nicht mehr schaltbereit ist (vgl. die entsprechende Aktion in Zustand (2) von Abbildung 5-11)

Wenn man ein Petrinetz in Anlehnung an [Obe96] als einen Workflow (bzw. ein *Workflow-Schema* [JBS97]) interpretiert, so kann die in Abbildung 5-11 skizzierte Ausführung der Aktion `schalten` abstrakt als die Ausführung eines Vorgangs bzw. (abhängig von der verwendeten Terminologie) einer Teilaufgabe interpretiert werden. Im nächsten Kapitel wird dieser Gedanke im Beispiel *Ad-hoc Workflows* (Abschnitt 6.2.5) weiterentwickelt, welches die petrinetzbasierte Definition und Ausführung von Workflows mit mehreren Benutzern ermöglicht.

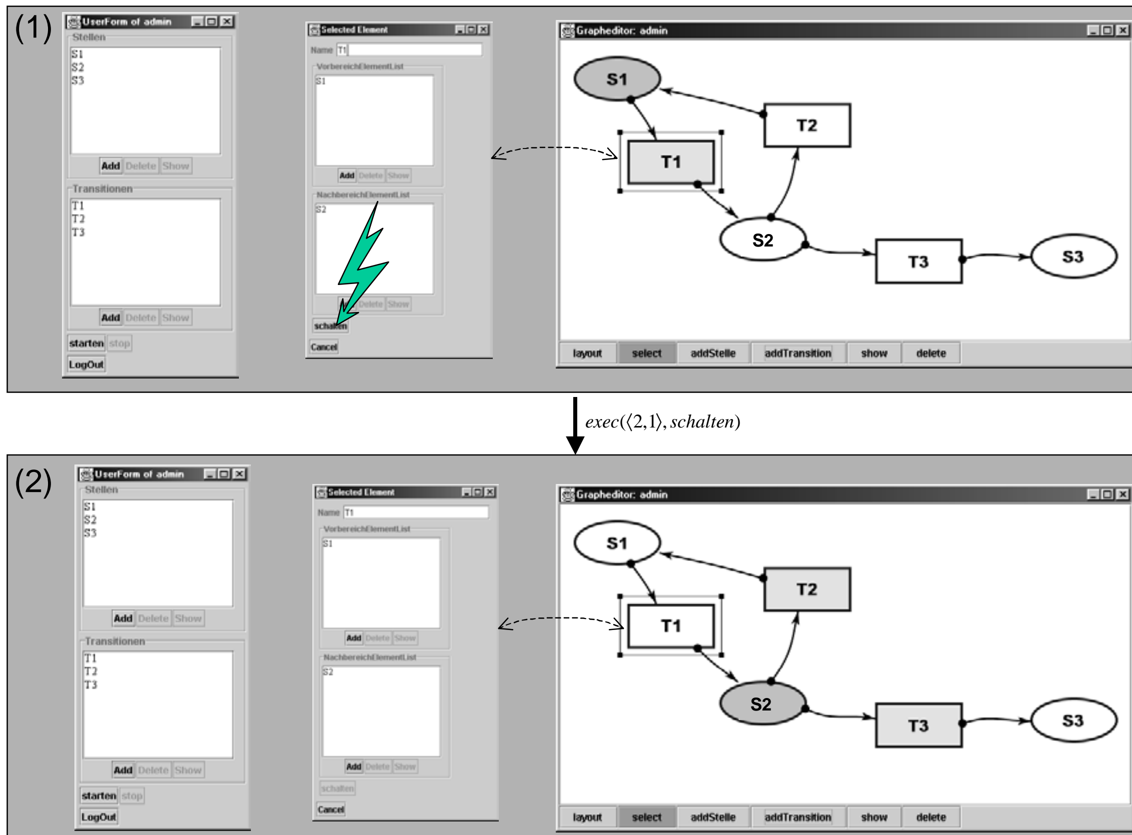


**Abbildung 5-11: Ausführung des Simulationswerkzeugs für Petrinetze**

## 5.3 Semantik der EMU-Eingabemodelle

Die Semantik der EMU-Eingabemodelle besteht aus der Definition eines Mehrbenutzer-EMU-Systems mit Aktionen und Aktivitätsnetzen. Wir erweitern daher in diesem Abschnitt die in Abschnitt 4.3 definierte Semantik von EMU-Systemen, die nur aus einem EMU-Datenmodell generiert wurden, um die folgenden Aspekte:

- Pro Zustand müssen an den Knoten die Aktivitätsnetze mit ihren aktuellen Markierungen verwaltet werden.
- Pro Benutzer muss ein Zugriffsadapter verwaltet bzw. berechnet werden.
- Bei den Interaktionen müssen nun die im Aufgabenmodell explizit definierten Aktionen berücksichtigt werden.
- Bei der Zustandsübergangsfunktion muß die Auswirkung der Ausführung der explizit definierten Aktionen berücksichtigt werden, wozu auch eine Veränderung der Markierung eines Aktivitätsnetzes gehören kann.

Dazu benötigen wir in diesem Abschnitt neben der bereits im letzten Kapitel betrachteten Produktionenmenge $Prod_{DM}$, des Datenmodells *DM* Informationen über das Aufgaben- und Benutzermodell *TM* bzw. *UM*. Erneut verwenden wir eine Reihe von Substitutionen, um eine verkürzte und damit übersichtlichere Darstellung zu erhalten:

$ExplAction_{TM}(X,a,p,c,i) \equiv$ a ist (explizite) Aktion von $X$,
mit der Vorbedingung p,
dem Kommando c,
$i = 0$ oder es gibt ein i-tes Resultat von a

$Phase_{TM}(X,p) \equiv p$ ist Phase von $X$

$StartPhase_{TM}(X,p) \equiv p$ ist Startphase von $X$

$Trans_{TM}(X,p,a,0,q) \equiv$

`...p...->a->...q...` ist Transition im Aktivitätsnetz von X,
$ExplAction(X,a,*,*,0)$

$Trans_{TM}(X,p,a,i,q) \equiv$

`...p...->a.`$r_i$`->...q...` ist Transition im Aktivitätsnetz von X,
$ExplAction(X,a,*,*,i)$

$UserTypes_{UM} \subseteq \{T \in IDT_{DM} \mid Tupel_{DM}(T)\}$

$Read_{UM}(X,b,S_i) \equiv$

`[b] … read` $S_i$ … ist Zugriffsregel von X,
$Tupel_{DM}(X,*,S_i)$

$Write_{UM}(X,b,S_i) \equiv$

`[b] … write` $S_i$ … ist Zugriffsregel von X,
$Tupel_{DM}(X,*,S_i)$

$Exec_{UM}(X,b,a) \equiv$

`[b] … exec a…` ist Zugriffsregel von X,
$ExplAction(X,a,*,*,*)$

### 5.3.1 Zustandserweiterungen

**Markierung der Aktivitätsnetze**

Bei den vereinfachten EMU-Systemen, die im vorherigen Kapitel betrachtet wurden, bestand der Zustand eines EMU-Systems vom Typ $X$ aus einem Paar $(\tau_X, \sigma_X)$. Zusätzlich ist nun die aktuelle Markierung der Phasen der Aktivitätsnetze an den einzelnen Knoten des MAKs zu verwalten. Dazu dient die Funktion $\alpha_X : Pos_X \cup \{\perp\} \to \wp(ID)$ mit $\alpha_X(p) \subseteq \{q \mid Phase_{TM}(type_X(p),q)\}$. Der erweiterte Zustand besteht damit aus dem Tripel $(\tau_X, \sigma_X, \alpha_X)$.

**Zugriffsadapter**

Nun ist zu klären, wie die einzelnen Benutzer auf den MAK zugreifen und damit auch Zustandsänderungen hervorrufen können. Dazu wird zunächst die aktuelle Menge der Benutzer festgelegt:

$$users_{\tau_X} = \{ p \in \tau_X \mid type_X(p) \in UserTypes_{UM} \}$$

Der bereits mehrfach erwähnte Zugriffsadapter eines Benutzers mit der Position *u* besteht formal aus dem Paar $(read_{\tau_X}^{\sigma_X,\alpha_X,u}, action_{\tau_X}^{\sigma_X,\alpha_X,u})$. Die erste Komponente beschreibt die Knoten, auf die der Benutzer lesend zugreifen kann, die zweite Komponente die Aktionen, die der Benutzer ausführen kann.

Bei der Diskussion des Beispiels 5-1 *EMUProjekt* wurde den beiden Benutzern der Zugriff auf den MAK „über" den Wurzelknoten ermöglicht. D.h. beide Benutzer sehen in ihrem Formular gemäß ihrer Zugriffsrechte Teile der „MAK-Spitze" (vgl. Abbildung 5-8). Bei der Diskussion weiterer Beispiele im nächsten Kapitel wird sich zeigen, dass es ferner nützlich ist, wenn einem Benutzer mit der Benutzerposition *u* zusätzlich (möglicherweise auch *ausschließlich*, vgl. das Beispiel *Ad-hoc Workflows*, Abschnitt 6.2.5) der Teilbaum an der Position *u* über das Formular präsentiert wird. Dieser Zugriff ist insbesondere für die passiven Benutzer notwendig, um auf „ihr" Login-Teilformular zugreifen zu können.

Daher erfolgt die Definition von $read_{\tau_X}^{\sigma_X,\alpha_X,u}$ ausgehend von der (globalen) Wurzelposition $\langle\rangle$ *und* von der (lokalen) Benutzerposition $u$. Formal handelt es sich dabei um eine Vereinigung zweier Knotenmengen:

$$read_{\tau_X}^{\sigma_X,\alpha_X,u} =_{def} global_{\tau_X}^{\sigma_X,\alpha_X,u} \cup local_{\tau_X}^{\sigma_X,\alpha_X,u}$$

$$\text{mit } read_{\tau_X}^{\sigma_X,\alpha_X,u}, global_{\tau_X}^{\sigma_X,\alpha_X,u}, local_{\tau_X}^{\sigma_X,\alpha_X,u} \subseteq \tau_X$$

$global_{\tau_X}^{\sigma_X,\alpha_X,u}$ ist dabei die größte Menge mit den folgenden Eigenschaften

$$\begin{aligned}
&\langle\rangle \in global_{\tau_X}^{\sigma_X,\alpha_X,u},\\
&p \circ \langle i \rangle \in global_{\tau_X}^{\sigma_X,\alpha_X,u} \Rightarrow\\
&\quad p \in global_{\tau_X}^{\sigma_X,\alpha_X,u},\\
&\quad Tupel_{DM}(type_X(p), T_i, S_i) \Rightarrow\\
&\qquad Read_{UM}(type_X(p), b, S_i) \vee Write_{UM}(type_X(p), b, S_i),\\
&\qquad \mathcal{B}[\![b]\!]_{\tau_X}^{\sigma_X,\alpha_X}(p,u)
\end{aligned}$$

Der Zugriff auf die Wurzelkomponente ist immer möglich (evtl. aber nicht der Zugriff auf die Komponenten der Wurzel!) und auf alle Vorgängerknoten eines zugreifbaren Knotens kann ebenfalls zugegriffen werden. Wenn ein Knoten in $global_{\tau_X}^{\sigma_X,\alpha_X,u}$ enthalten ist und eine Komponente eines Tupelknotens darstellt, so ist er gemäß einer Zugriffsregel des Aufgabenmodells zugreifbar und deren Anwendbarkeitsbedingung ist erfüllt.

$local_{\tau_X}^{\sigma_X,\alpha_X,u}$ ist die größte Menge mit den folgenden Eigenschaften

$$
\begin{aligned}
& u \in local_{\tau_X}^{\sigma_X,\alpha_X,u}, \\
& \forall p \in local_{\tau_X}^{\sigma_X,\alpha_X,u} \exists q : p = u \circ q, \\
& p \circ \langle i \rangle \in local_{\tau_X}^{\sigma_X,\alpha_X,u} \Rightarrow \\
& \qquad p \in local_{\tau_X}^{\sigma_X,\alpha_X,u}, \\
& \qquad Tupel_{DM}(type_X(p), T_i, S_i) \Rightarrow \\
& \qquad\qquad Read_{UM}(type_X(p), b, S_i) \vee Write_{UM}(type_X(p), b, S_i), \\
& \qquad\qquad \mathcal{B}[\![b]\!]_{\tau_X}^{\sigma_X,\alpha_X}(p, u)
\end{aligned}
$$

In Abbildung 5-12 werden mögliche Konstellationen von $global_{\tau_X}^{\sigma_X,\alpha_X,u}$ und $local_{\tau_X}^{\sigma_X,\alpha_X,u}$ , die sich abhängig vom Benutzermodell ergeben, schematisch dargestellt. In Teilskizze (A) hat der durch den Knoten $u$ modellierte Benutzer sowohl auf Knoten des Teilbaums von $u$ ($local_{\tau_X}^{\sigma_X,\alpha_X,u}$) als auch auf Teile des Gesamtbaums ($global_{\tau_X}^{\sigma_X,\alpha_X,u}$) Zugriff und es gilt: $global_{\tau_X}^{\sigma_X,\alpha_X,u} \cap local_{\tau_X}^{\sigma_X,\alpha_X,u} \neq \emptyset$. In Teilskizze (B) ist dieser Schnitt leer.

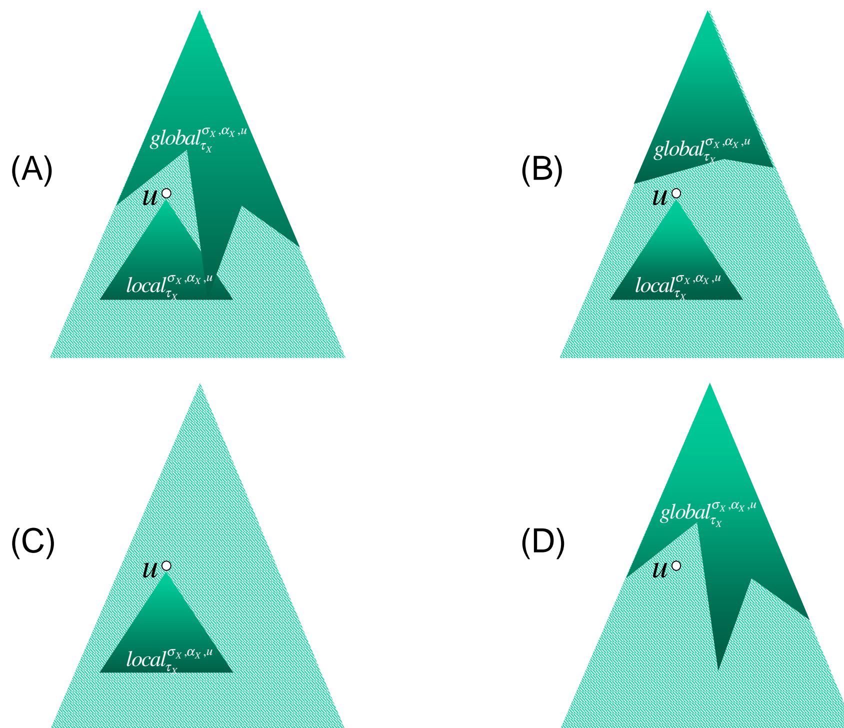


**Abbildung 5-12: Schematische Skizze von** $read_{\tau_X}^{\sigma_X,\alpha_X,u}$

In Teilskizze (C) hat der Benutzer nur Zugriff auf Knoten im Teilbaum von $u$ ($global_{\tau_X}^{\sigma_X,\alpha_X,u} = \{\langle\rangle\}$) und in (D) nur auf Knoten aus $global_{\tau_X}^{\sigma_X,\alpha_X,u}$ ($local_{\tau_X}^{\sigma_X,\alpha_X,u} = \{u\}$).

Man beachte, dass durch das implizit definierte Benutzeraktivitätsnetz (vgl. Abschnitt 5.1.5) der Zugriff passiver Benutzer festgelegt wird und daher bei der Definition von $local_{\tau_X}^{\sigma_X,\alpha_X,u}$ keine entsprechende Fallunterscheidung notwendig ist.

Die zweite Komponente des Zugriffsadapters $action_{\tau_X}^{\sigma_X,\alpha_X,u}$ ist abhängig von der ersten Komponente $read_{\tau_X}^{\sigma_X,\alpha_X,u}$, weil nur Aktionen von Knoten ausführbar sind, auf denen ein lesender Zugriff gewährt wird.

$$
\begin{aligned}
action_{\tau_X}^{\sigma_X,\alpha_X,u} = \{&\widetilde{actIdent}(p\circ\langle i\rangle,a,u) \mid \\
&\quad p\circ\langle i\rangle \in read_{\tau_X}^{\sigma_X,\alpha_X,u}, actIdent(p\circ\langle i\rangle,a) \in ImplAction_X, \\
&\quad Tupel_{DM}(type_X(p),T_i,S_i) \Rightarrow \\
&\qquad Write_{UM}(type_X(p),b,S_i), \\
&\qquad \mathcal{B}[\![b]\!]_{\tau_X}^{\sigma_X,\alpha_X}(p,u) \\
\} \cup \{&\widetilde{exec}(p,a,u) \mid \\
&\quad p \in read_{\tau_X}^{\sigma_X,\alpha_X,u}, \\
&\quad Exec_{UM}(type_X(p),b,a), \\
&\quad \mathcal{B}[\![b]\!]_{\tau_X}^{\sigma_X,\alpha_X}(p,u) \\
\}&
\end{aligned}
$$

Dabei gilt: $actIdent = setAlt \mid add \mid remove \mid setRef \mid setValue$

Die implizit definierten Aktionen (vgl. Abschnitt 4.3) sind höchstens dann eingeschränkt, wenn der Vorgängerknoten (`parent`) ein Tupeltyp ist. Bevor nun die Aktionen in der Definition der Zustandsübergangsfunktion verwendet werden, ist die bisher schon öfter verwendete Semantik von Bedingungen zu definieren.

### 5.3.2 Semantik von Bedingungen und Pfadausdrücken

In Bedingungen können sowohl von der Datenposition als auch von der Benutzerposition beliebig lange Pfade zu Vorgänger-, Nachfolger- und Nachbarknoten vorkommen. Ferner können Pfade über Referenzkanten gebildet werden (vgl. Syntax in Anhang A).

Die Semantik wird rekursiv über die beiden Argumentpositionen definiert. Da dabei sowohl der Baum als auch die Knotenbewertungsfunktion und die Phasenmarkierung konstant bleiben, stellen wir die Semantik von Bedingungen in der folgenden Curry-Schreibweise dar:

$$\mathcal{B}[\![.]\!]_{\tau_X}^{\sigma_X,\alpha_X} : Pos_X \times Pos_X \to \{false, true\}$$

Zur Definition von $\mathcal{B}[\![.]\!]_{\tau_X}^{\sigma_X,\alpha_X}$ benötigen wir die Semantik der syntaktischen Pfadausdrücke, die von $\tau_X$ und $\sigma_X$ (wegen den Referenzkanten) abhängt:

$$\mathcal{P}[\![.]\!]_{\tau_X}^{\sigma_X} : Pos_X \times Pos_X \to Pos_X \cup \{\bot\}$$

Im EMU-System aus Abbildung 5-9 gilt etwa

$$\mathcal{P}[\![\texttt{parent.Anforderungen}]\!]_{\tau_X}^{\sigma_X}(\langle 1\rangle,\langle 2\rangle)=\langle 3\rangle,\ \mathcal{P}[\![\texttt{User}]\!]_{\tau_X}^{\sigma_X}(\langle 1\rangle,\langle 2\rangle)=\langle 2\rangle$$

Ein Pfadausdruck legt also genau einen Knoten von einem gegebenen Paar aus Daten- und Benutzerposition (vgl. Abschnitt 5.1.4) fest.

$$\mathcal{P}[\![\varepsilon]\!]_{\tau_X}^{\sigma_X}(d,*)=_{def}\mathcal{P}[\![\texttt{this}]\!]_{\tau_X}^{\sigma_X}(d,*)=_{def}d,$$

$$\mathcal{P}[\![\texttt{root}]\!]_{\tau_X}^{\sigma_X}(*,*)=_{def}\langle\rangle,$$

$$\mathcal{P}[\![\texttt{parent}]\!]_{\tau_X}^{\sigma_X}(\langle\rangle,*)=_{def}\perp,$$

$$\mathcal{P}[\![\texttt{User}.REST]\!]_{\tau_X}^{\sigma_X}(d,u)=_{def}\mathcal{P}[\![REST]\!]_{\tau_X}^{\sigma_X}(u,u),$$

$$\mathcal{P}[\![\texttt{parent}.REST]\!]_{\tau_X}^{\sigma_X}(d\circ\langle i\rangle,u)=_{def}\mathcal{P}[\![REST]\!]_{\tau_X}^{\sigma_X}(d,u)$$

$$\mathcal{P}[\![S_i.REST]\!]_{\tau_X}^{\sigma_X}(d,u)=_{def}\begin{cases}\mathcal{P}[\![REST]\!]_{\tau_X}^{\sigma_X}(d\circ\langle i\rangle,u) \text{wenn } Tupel_{DM}(type_X(d),T_i,S_i)\\ \perp \quad \text{sonst}\end{cases},$$

$$\mathcal{P}[\![A_i.REST]\!]_{\tau_X}^{\sigma_X}(d,u)=_{def}\begin{cases}\mathcal{P}[\![REST]\!]_{\tau_X}^{\sigma_X}(p\circ\langle i\rangle,u) \text{wenn } Var_{DM}(type_X(d),A_i),d\circ\langle i\rangle\in\sigma_X\\ \perp \quad \text{sonst}\end{cases},$$

$$\mathcal{P}[\![elementAt(i).REST]\!]_{\tau_X}^{\sigma_X}(d,u)=_{def}\begin{cases}\mathcal{P}[\![REST]\!]_{\tau_X}^{\sigma_X}(d\circ\langle i\rangle,u) \text{ wenn } List_{DM}(type_X(d),E),d\circ\langle i\rangle\in\sigma_X\\ \perp \quad \text{sonst}\end{cases},$$

$$\mathcal{P}[\![\texttt{value}.REST]\!]_{\tau_X}^{\sigma_X}(d,u)=_{def}\begin{cases}\mathcal{P}[\![REST]\!]_{\tau_X}^{\sigma_X}(q) \quad \text{wenn } Ref_{DM}(type_X(d),S,T,c),\sigma_X(d)=\sigma_X(q)\\ \perp \quad \text{sonst}\end{cases}$$

In der ersten Zeile werden die terminalen Fälle behandelt. Durch die ausgezeichneten Pfadausdrücke **`root`** und **`User`** kann man auf die Wurzel des MAK bzw. die aktuelle Benutzerposition zugreifen. Die letzte Zeile definiert Pfade über Referenzen.

Unter Verwendung von $\mathcal{P}[\![.]\!]_{\tau_X}^{\sigma_X}$ kann nun $\mathcal{B}[\![.]\!]_{\tau_X}^{\sigma_X,\alpha_X}$ definiert werden. Die Variablen $p_1$, $p_2$ stehen dabei für (syntaktische) Pfadausdrücke.

$$\mathcal{B}[\![p_1 == p_2]\!]_{\tau_X}^{\sigma_X,\alpha_X}(d,u)\Leftrightarrow_{def}\begin{cases}false \quad \text{wenn } \mathcal{P}[\![p_i]\!]_{\tau_X}^{\sigma_X}(d,u)=\perp, i=1 \text{ oder } i=2\\ v_1=v_2 \quad \text{sonst, mit } \sigma_X(\mathcal{P}[\![p_i]\!]_{\tau_X}^{\sigma_X}(d,u))=v_i, i=1,2\end{cases}$$

$$\mathcal{B}[\![p_1]\!]_{\tau_X}^{\sigma_X,\alpha_X}(d,u)\Leftrightarrow_{def}p_1\in\alpha_X(p_1)$$

$$\mathcal{B}[\![p_1.A_i]\!]_{\tau_X}^{\sigma_X,\alpha_X}(d,u)\Leftrightarrow_{def}Var_{DM}(type_X(\mathcal{P}[\![p_1]\!]_{\tau_X}^{\sigma_X}(d,u),A_i)),(\mathcal{P}[\![p_1]\!]_{\tau_X}^{\sigma_X}(d,u)\circ\langle i\rangle)\in\tau_X$$

$$\mathcal{B}[\![ p_1 \texttt{ in } p_2 ]\!]_{\tau_X}^{\sigma_X,\alpha_X} (d,u) \Leftrightarrow_{def} \sigma_X (\mathcal{P}[\![ p_1 ]\!]_{\tau_X}^{\sigma_X} (d,u)) \in ValueSet_{\tau_X}^{\sigma_X} (\mathcal{P}[\![ p_2 ]\!]_{\tau_X}^{\sigma_X} (d,u))$$

$$\mathcal{B}[\![ p( p_1,\ldots, p_n ) ]\!]_{\tau_X}^{\sigma_X,\alpha_X} (d,u) \Leftrightarrow_{def} [\![ p ]\!](\sigma_X (\mathcal{P}[\![ p_1 ]\!]_{\tau_X}^{\sigma_X} (d,u)),\ldots,\sigma_X (\mathcal{P}[\![ p_n ]\!]_{\tau_X}^{\sigma_X} (d,u)))$$

Die Semantik der logischen Operatoren ( **`!, &, |, ->, <->`**) sei wie üblich definiert.

Das Symbol für den Gleichheitsoperator „**`==`**" überprüft die Gleichheit von Knotenwerten durch Anwendung der Knotenbewertungsfunktion $\sigma_X$ . Aufgrund der Definition von $\sigma_X$ führt dies bei Tupelknoten zum Vergleich der Identität. Es kann jedoch auch ein Referenzknoten mit einem Tupelknoten (oder auch mit einem weiteren Referenzknoten) verglichen werden. Anschaulich gesprochen liefert dieser Vergleich genau dann *true*, wenn der Referenzknoten auf den Tupelknoten selber (oder eben zwei Referenzknoten auf denselben) verweist. Dabei ergibt der Vergleich zweier undefinierter Werte stets *false*.

Der Ausdruck „**`p.A`**" bzw. nur „**`A`**" prüft, ob der durch p bzw. $\varepsilon$ angesprochene Variantenknoten die aktuelle Ausprägung **`A`** hat. Wenn ein Ausdruck eine aktuell aktive Phase beschreibt, so ist er gültig.

Der Ausdruck „$p_1$ **`in`** $p_2$" prüft, ob in der durch einen Knoten $p_2$ (bei gegebenem Ankerknoten *pos*) festgelegten Menge $ValueSet_{\tau_X}^{\sigma_X} (\mathcal{P}[\![ p_2 ]\!]_{\tau_X}^{\sigma_X} (d,u))$ der Wert von $p_1$ enthalten ist.

$$\begin{aligned} ValueSet_{\tau_X}^{\sigma_X} (p) =_{def} \{ \sigma_X (q) \mid\ & \sigma_X (q) = \sigma_X (p) \vee \\ & Tupel_{DM} (type_X (p), T_i, S_i), q \in ValueSet_{\tau_X}^{\sigma_X} (p \circ \langle i \rangle) \vee \\ & Var_{DM} (type_X (p), A_i), p \circ \langle i \rangle \in \tau_X , q \in ValueSet_{\tau_X}^{\sigma_X} (p \circ \langle i \rangle) \vee \\ & List_{DM} (type_X (p), E), p \circ \langle i \rangle \in \tau_X , q \in ValueSet_{\tau_X}^{\sigma_X} (p \circ \langle i \rangle) \vee \\ & Ref_{DM} (type_X (p), S, T, c), p \circ \langle i \rangle \in \tau_X , q \in ValueSet_{\tau_X}^{\sigma_X} (p \circ \langle i \rangle) \\ & \} \setminus \{\bot\} \end{aligned}$$

Die Semantik eines n-stelligen Prädikatsymbols *p* ergibt sich durch seine Zuordnung zu einer n-stelligen booleschen Abbildung $[\![ p ]\!]$. Wie das Beispiel 5-2 *Simulationswerkzeug für Petrinetze* zeigt, kann dies technisch durch die Implementierung einer booleschen Methode zur Knotenklasse erfolgen.

### 5.3.3 Erweiterte Interaktionen

Sei $ActValue_X =_{def} \{\alpha_X\}$ und $Action_X =_{def} \bigcup_{\tau \in Tree_X, \sigma \in NodeValue_X, \alpha \in ActValue_X, u \in users_\tau} action_\tau^{\sigma,\alpha,u}$ .

Dann ergibt sich die folgende Zustandsübergangsfunktion eines Mehrbenutzersystems mit Aufgaben vom Typ *X*:

$$\Delta_X : Tree_X \times NodeValue_X \times ActValue_X \times Action_X \rightarrow Tree_X \times NodeValue_X \times ActValue_X$$

$$\Delta_X(\tau_X,\sigma_X,\alpha_X,\widetilde{actIdent}(p,a,u)) =_{def} (\Delta^I_X(\tau_X,\sigma_X),\alpha'_X) \qquad \text{mit}$$
$$\widetilde{actIdent}(p,a,u) \in action^{\sigma_X,\alpha_X,u}_{\tau_X},$$
$$(\tau'_X,\sigma'_X) = \Delta^I_X(\tau_X,\sigma_X),$$
$$\alpha'_X(q) = \begin{cases} \{s \mid StartPhase_{TM}(type_X(p),s)\} & \text{wenn } q \in (\tau'_X \setminus \tau_X), \\ \alpha_X(q) & \text{sonst} \end{cases},$$

$$\Delta_X(\tau_X,\sigma_X,\alpha_X,\widetilde{exec}(p,a,u)) =_{def} (\Delta^I_X(\tau_X,\sigma_X),\alpha'_X) \quad \text{mit}$$
$$\widetilde{exec}(p,a,u) \in action^{\sigma_X,\alpha_X,u}_{\tau_X},$$
$$ExplAction_{TM}(type_X(p),a,pre,c,i),$$
$$(\tau'_X,\sigma'_X,result) = Com[\![c]\!](\tau_X,\sigma_X,\alpha_X,p,u)$$
$$V = \{v \mid Trans_{TM}(X,v,a,result,*)\},$$
$$N = \{n \mid Trans_{TM}(X,*,a,result,n)\},$$
$$\alpha'_X(q) = \begin{cases} (\alpha_X(q) \setminus V) \cup N & \text{wenn } q = p \\ \alpha_X(q) & \text{sonst} \end{cases}$$

Die semantischen Funktion eines Kommandos hat die folgende Funktionalität:

$$Com[\![.]\!]: \quad Tree_X \times NodeValue_X \times ActValue_X \times Pos_X \times Pos_X \to Tree_X \times NodeValue_X \times \mathbb{N}$$

Ein Kommando kann die Baumstruktur und die Knotenbewertungsfunktion, jedoch nicht die Markierung der Aktivitätsnetze an den Knoten (also die mit $\alpha_X$ bezeichneten Funktionen) ändern. Technisch erfolgt dies über die von EMUGEN erstellte API, wie es bereits bei den beiden Beispielen in diesem Kapitel gezeigt wurde. Der weitere Rückgabewert von $Com[\![.]\!]$ ist der Index des Resultats (vgl. Abschnitt 5.1.1).

Wir definieren an dieser Stelle aus den in Abschnitt 5.1.1 dargelegten Gründen nicht die Semantik der Kommandosprache und daher auch nicht $Com[\![.]\!]$.

### 5.3.4 Erweitertes Zustandsübergangssystem

Das aus einem EMU-Datenmodell *DM*, EMU-Aufgabenmodell TM und einem EMU-Benutzermodell *UM* konstruierte Zustandsübergangssystem ist wie folgt definiert:

$$Sys_{DM,TM,UM} = (Actions, State, \to, Init) \qquad \text{mit}$$
$$Actions = Actions_{start_{DM}},$$
$$State = \{Init\} \cup \{s \in (Tree_{start_{DM}} \times NodeValue_{start_{DM}} \times ActValue_{start_{DM}}) \mid \exists a_1,\ldots,a_n \in Actions : Init \xrightarrow{\langle a_1,\ldots,a_n \rangle} s\},$$
$$s \xrightarrow{a} s' \Leftrightarrow \Delta_{start_{DM}}(s,a) = s',$$

$$Init = (\tau_{start_{DM}}, \sigma_{start_{DM}}, \alpha_{start_{DM}}),$$
$$(\tau_{start_{DM}}, \sigma_{start_{DM}}) = create_{start_{DM}}(\emptyset, \emptyset, \langle\rangle),$$
$$\alpha_{start_{DM}}(p) = \{s \mid p \in \tau_{start_{DM}}, StartPhase_{TM}(type_{start_{DM}}(p), s)\}$$

## 5.4 Begründung und mögliche Alternativen

In diesem Abschnitt soll eine Begründung der Sprachentscheidung vorgenommen werden. Zunächst wird motiviert, warum EMU kein explizites Dialogmodell als Eingabe verwendet. Anschließend werden mögliche Alternativen für das EMU-Aufgabenmodell betrachtet. Da die Benutzermodellierung weitgehend durch die Datenmodellierungssprache geprägt ist (Benutzer werden durch Tupeltypen beschrieben, Benutzereigenschaften sind Bedingungen des Datenmodells), wird durch Abschnitt 4.4 bereits die EMU-Benutzermodellierungssprache begründet. Man kann diesen Sachverhalt mit dem Vorgehen bei relationalen Datenbanken vergleichen, wo die Benutzer durch bestimmte Tabellen – also in der gleichen relationalen Form wie die Daten selbst – verwaltet werden und damit auch das Benutzermodell durch das Datenmodell bestimmt wird.

### 5.4.1 Warum kein Dialogmodell?

CADUI-Werkzeuge und Software-Entwicklungsmethoden (z.B. die nach Denert [Den91]) benutzen bzw. propagieren meist sowohl eine Aufgaben- als auch eine Dialogmodellierungssprache. Um zu motivieren, warum im EMU-Generierungskonzept kein explizites Dialogmodell verwendet wird, ist der Begriff *Dialog* bzw. *Dialogkontrolle*/*Dialogmodell* genauer zu klären.

Bei der Spezifikation von Grapheditoren im vorherigen Kapitel zeigte sich die Notwendigkeit einer Dialogkontrolle auf sehr niedrigem Abstraktionsniveau, etwa zur Festlegung, wie ein Graphelement selektiert oder deselektiert wird. Dialoge dieser Art werden also im Layoutmodell von EMU definiert.

Zur Definition einer Dialogkontrolle auf höherem Abstraktionsniveau – man denke etwa an das Standardbeispiel *Geldausgabeautomat* [Den91, Sch97, Eic98] – benötigt man genau die Sprachelemente, die bereits im Aufgabenmodell vorhanden sind. Im EMU-Aufgabenmodell gibt es Sprachelemente zur Modellierung von

- Zuständen,
- Benutzerinteraktionen mit Vorbedingungen,
- Aktionsausführungen mit Nachbedingungen bzw. Resultaten,
- Transitionen, die Zustände beim Eintreten einer Benutzerinteraktion, einer anschliessenden Aktionsausführung und dem Eintreten einer bestimmten Nachbedingung in einen Nachfolgezustand überführen und
- zustandsabhängige Zugriffsdefinitionen (vgl. *Datensicht* in [Den91]).

Wie wir in Abschnitt 6.2.2 sehen werden, sind diese Sprachelemente gut geeignet, um damit auch Dialoge zu modellieren. Daher ist kein explizites Dialogmodell notwendig.

### 5.4.2 Alternative Aufgabenmodelle

Wie im Einleitungskapitel fassen wir hier unter dem Aufgabenaspekt alle dynamischen Aspekte eines interaktiven Informationssystems zusammen. Dynamische Aspekte lassen sich durch folgende Basiskonzepte der Software-Technik *konstruktiv* beschreiben (*nicht konstruktiv* wäre etwa eine temporal-logische Formel [CS99] oder ein UML-Sequenzdiagramm [OMG01]):

- Deterministischer endlicher Automat mit Variationen wie etwa Interaktionsdiagramme [Den91], Transitionsdiagramme [Eic98], Statecharts [Har87];
- Petrinetze mit Variationen wie etwa NF/T-Netze [Obe96] zur Workflow-Modellierung, Anwendungsfalls- und Aktivitätsdiagramme von UML [OMG01];
- Algebraische Spezifikationen, Prozessalgebren (CCS [Mil89], LOTOS [BB87]); objektorientierte Spezifikationssprachen (Object-Z [Hus99, HC99]);
- Attributierte Grammatiken [Eic98]; Syntaxdiagramme zur Beschreibung der Sprache der Interaktionen [Lar92, Eic98],
- Modelle der kognitiven Psychologie (GOMS [CMN83], TKS [Joh92]).

Die unter den ersten beiden Punkten genannten Basiskonzepte haben den Vorteil, dass sich die entsprechenden Modelle auch unmittelbar graphisch darstellen lassen und kommen deshalb bei der Software-Entwicklung [Den91, Bal00] oft zum Einsatz. Diese Art der Modellierung wird deshalb auch von EMUGEN unterstützt (vgl. die graphische TUM-Spezifikationstechnik in Abschnitt 5.1.6 und die *Ad-hoc Workflows* in Abschnitt 6.2.5).

Anwendungsfall- und Aktivitätsdiagramme wurden bereits in Zusammenhang mit den TUMs exemplarisch angesprochen. Sie haben sich bei der Entwicklung interaktiver Systeme in der Praxis gut bewährt, obwohl sie von keinem entsprechenden Werkzeug als *konstruktive* Spezifikation verwendet werden. D.h. es ist beispielsweise nicht möglich, aus einem Anwendungsfalldiagramm eine Benutzungsoberfläche zu erzeugen, weil die dargestellten Informationen zu vage und der Bezug zu weiteren Modellen der Spezifikation (z. B. Klassendiagramm), die zur Erzeugung einer Benutzungsoberfläche notwendig wären, nicht ausreichend festgelegt sind.

Bei der Überlegung, ob als Modellierungssprache für den dynamischen Aspekt eine Variation eines endlichen Automaten oder eines Petrinetzes verwendet werden soll, ist zu beachten, dass ein deterministischer endlicher Automat, wenn er zur Modellierung der Dynamik eines interaktiven Systems verwendet wird, leicht durch ein Petrinetz simuliert werden kann. Jedem Interaktionstoken kann dazu eine Transition mit leerem Vorbereich zugeordnet werden. Umgekehrt ist dies natürlich auch möglich, aber praktisch aufgrund der dazu notwendigen (im Extremfall exponentiellen) Anzahl von Zustände im endlichen Automaten nicht sinnvoll. Daher ist es naheliegend, zur Aufgabenmodellierung Petrinetze zu bevorzugen.

Wenn es sich bei dem zu modellierenden dynamischen Aspekt um gemeinsame Aufgaben (Workflows) handelt, sind Petrinetze vor allem deshalb eine adäquate Modellierungssprache, weil es häufig vorkommt, dass mehrere Teilaufgaben (modelliert durch Stellen) gleichzeitig aktiv bzw. markiert sind. Die Verwendung von Petrinetzen zur Aufgabenmodellierung wurde daher bereits häufig (z.B. in [Obe96, Aal98, Pod00]) praktiziert.

Demgegenüber erfolgt die Modellierung von Dialogen meist durch Variationen endlicher Automaten, weil dabei jedem Zustand ein Layout der Benutzungsoberfläche zugeordnet werden muss. [Den91] spricht dabei von *Masken* bzw. von einer *Maskentabelle* für die Zuordnung von Zuständen zu Masken. Arbeitet man mit einem Petrinetz zur Dialogmodellierung, so liegt es nahe, jeder Stelle des Petrinetzes ein Layout zuzuweisen. Dabei muss jedoch festgelegt werden, was passiert, wenn mehrere Stellen des Petrinetzes markiert sind. Dieses Problem wird bei EMU dadurch gelöst, dass die Zuordnung auf abstrakter Ebene erfolgt. D.h. es wird pro Phase eines Aktivitätsnetzes im EMU-Benutzermodell festgelegt, welche Zugriffsrechte auf die zugehörigen Tupelkomponenten bestehen, nicht aber die Anordnung der zugehörigen Teilformulare im Layout. Unabhängig davon wird das Layout, etwa die relative Anordnung der Teilformulare für die Komponenten und Aktionen bei Tupeltypen, implizit durch die Standardlayoutregeln oder explizit durch den Entwickler festgelegt.

Die Verwendung algebraischer Spezifikationen wurde bereits als Alternative zum Datenmodell in Abschnitt 4.4.3 diskutiert. Prinzipiell ist ihre Verwendung zwar auch zur Beschreibung dynamischer Aspekte möglich [Bau96], aber neben den in Abschnitt 4.4.3 bereits genannten Nachteile ist bei der Aufgabenmodellierung insbesondere auch die fehlende diagrammsprachliche Darstellungsmöglichkeit solcher Spezifikationen problematisch.

Ähnliche Probleme liegen bei Prozessalgebren bzw. -kalküle wie CCS vor. Obwohl etwa durch LOTOS auch diagrammsprachliche Darstellungen möglich sind, wären vollständig formal ausgearbeitete Spezifikationen (und solche wären für eine Generierung eines ausführbaren Systems bzw. eines Prototypen notwendig) – etwa für die in Kapitel 6 betrachteten Beispiele – nicht weniger kompakt als der notwendige Quellcode bei der Verwendung eines geeigneten objektorientierten Rahmenwerks. Diesen Ansätzen fehlt noch eine handhabbare, kompakte Syntax. Eine wesentliche Anforderung an die Modellierungssprachen dieser Arbeit besteht aber darin, dass sowohl durch ihre Kompaktheit als auch durch ihre konkrete Ausführbarkeit der Vorteil des generativen Ansatzes deutlich wird. Die praktische Anwendbarkeit von EMUGEN zeigt sich etwa in der mit weniger als 200 Zeilen sehr kompakten Eingabe für das Beispiel *Ad-hoc Workflows* (Abschnitt 6.2.5).

Die Verwendung formaler Sprachbeschreibungstechniken wie Grammatiken oder Syntaxdiagramme wurden bereits früh zur Beschreibung interaktiver Systeme aus Sicht der kognitiven Psychologie verwendet (etwa durch *Command Language Grammar, CLG* [Mor81]). Dabei war es jedoch meist nicht das Ziel, eine Eingabesprache für ein Generierungswerkzeug zu entwerfen. Eine Ausnahme bildet dabei *TKS* (*Task Knowledge Structure*) [Joh92], eine an LOTOS angelehnte Modellierungssprache, die als Eingabe für das Adept-System [JMJ92], einem frühen, nicht vollständig automatisierten CADUI-Werkzeug, verwendet wird. Jedoch fehlt auch TKS eine mit den EMU-Modellen vergleichbaren Kompaktheit.

# 6 Entwicklung von EMU-Systemen

In diesem Kapitel wird gezeigt, wie EMU-Systeme entwickelt werden. Eine entsprechende Vorgehensweise wird im ersten Abschnitt beschrieben. Ihre konkrete Umsetzung wird anhand einiger Beispiele im zweiten Abschnitt gezeigt.

Wir betrachten dabei die folgenden Beispiele:

- Die Entwicklung eines Grapheditors für eine Diagrammsprache zur graphischen Spezifikation von Aufgaben- und Benutzermodellen (Beispiel *TUM*). Diese Diagrammsprache wird für die weiteren Beispiele als Spezifikationsmittel benutzt.
- Die Entwicklung eines Geldausgabeautomaten (Beispiel *Geldausgabeautomat*) dient dem direkten Vergleich mit bereits vorhandenen Dialogspezifikationstechniken, weil dieses Beispiel in der einschlägigen Literatur häufig verwendet wird (z.B. in [Den91, Sch97, Eic98]).
- Die Entwicklung eines einfachen Talk-Programms (Beispiel *Talk*) zeigt die Spezifikation einer einfachen, interaktiven Zugriffskontrolle.
- Die Entwicklung eines Systems zur Abwicklung eines Praktikums (Beispiel *Praktikum*) wird als Beispiel einer relativ komplexen Anwendung betrachtet, bei der sämtliche Sprachelemente von EMU zum Einsatz kommen.
- Bei der Entwicklung eines einfachen Workflow-Mangagement-Systems wird die Verwendung eines Grapheditors als Werkzeug zur Definition- und Administration von Workflows gemäß Abbildung 2-2 (Beispiel *Ad-hoc Workflows*) gezeigt.

Die Diskussion einer ganzen Reihe von Beispielen dient dem Beleg der allgemeinen und komfortablen Anwendbarkeit von EMUGEN zur Generierung interaktiver Informationssysteme. Dies ist notwendig, weil für Sprachen zur Beschreibung interaktiver Systeme im Gegensatz zur Theorie funktionaler Berechnungen kein Berechenbarkeitsbegriff existiert. Ferner gibt es eine Reihe praxisrelevanter Anforderungen an CADUI-Werkzeugen, wie beispielsweise die Allgemeinheit sowie die

Werkzeugen, wie beispielsweise die Allgemeinheit sowie die Kompaktheit und Einfachheit der Eingabemodelle, die nur durch die Betrachtung verschiedenartiger Anwendungen belegt werden können.

## 6.1 Vorgehensweise

Eine Vorgehensweise zur Software-Entwicklung wird üblicherweise durch Phasen, Entwicklungsdokumente (Modelle) und die beteiligten Personen (bzw. Rollen wie z.B. *Kunde* und *Entwickler*) dargestellt [Bal00].

Die Vorgehensweise zur Entwicklung von EMU-Systemen wird im Gegensatz zu detailliert ausgearbeiteten Software-Entwicklungsprozessen hier nur rudimentär angegeben. Dennoch kann die partizipative Entwicklung von EMU-Systemen durch Kunde und Entwickler durch ein interaktives System unterstützt werden. In Beispiel 5-1 wurde ein entsprechendes EMU-System bereits entwickelt.

Der Kunde erstellt während der Analysephase eine Beschreibung der fachlichen (nicht der technischen) Anforderungen an das zu entwickelnde System (vgl. auch Abbildung 5-5). Dabei folgt er möglicherweise einem bestimmten Vorgehensmodell zur Anforderungsanalyse oder greift auf bereits bestehende formale Spezifikationen, z.B. ER-Modelle [Che76], Arbeitsablaufdiagramme wie EPK [JBS97] oder Aktivitätsdiagramme [OMG01] zu. Wir abstrahieren in der Darstellung von diesen Aspekten (obwohl sie natürlich praxisrelevant sind) und betrachten die Anforderungen als Black-Box, bzw. als ein Dokument, das während der Analyse vom Kunden bearbeitet wird.

Manche CADUI-Werkzeuge [Pat99, TMP98] bieten etwa die Möglichkeit, aus einer natürlichsprachlichen Beschreibung konstruktive Modellelemente (teil-)automatisiert zu erzeugen. Da die EMU-Modellierungssprachen in einer formalen Beschreibung vorliegen, sollte es möglich sein, entsprechende Transformationen aus einer gegebenen, externen Beschreibungsform zu implementieren. EMUGEN würde damit in einem umfassenderen Werkzeug genauso verwendet, wie etwa GRACE als Teilkomponente von EMUGEN (vgl. Abbildung 3-5).

Bei der Entwicklung interaktiver Informationssysteme mit einer graphischen Benutzungsoberfläche ist ein iteratives Vorgehen sinnvoll, bei dem die Analyse der Problemstellung immer wieder durchlaufen wird, bis das gewünschte interaktive System vorliegt [Bal00]. Daher kann die Analysephase durchaus auch zu einem Zeitpunkt vorläufig beendet werden, wenn noch nicht alle Anforderungen vollständig geklärt wurden, und zum Entwurf übergegangen werden.

Während der Entwurfsphase erstellt der Entwickler die EMU-Modelle und generiert daraus ablauffähige Prototypen des interaktiven Systems. Wichtig ist dabei auch eine Anwendungsdokumentation für den Kunden. Wenn der Entwickler glaubt, die aktuellen Anforderungen in ausreichender Weise berücksichtigt zu haben und den Prototyp dem Kunden zeigen will, so kann erneut die Analysephase eingeleitet werden. Nun führt der Kunde den Prototyp aus und benutzt dabei die Anwendungsdokumentation.

Teil der Anwendungsdokumentation sollten das graphisch dargestellte EMU-Datenmodell (vgl. Abbildung 4-1) und die notwendigen TUMs (vgl. Abbildung 5-5) sein, die in ihrer einfachen Darstellung auch für Nichtinformatikern verständlich sind.

## 6.2 Beispiele

Bei den folgenden Beispielen kann das oben beschriebene iterative Vorgehen natürlich nicht direkt vermittelt werden. Anstelle dessen werden sie wie folgt dokumentiert:

- Natürlichsprachliche Anforderungsbeschreibung
- Gegebenenfalls graphische Darstellung des Aufgaben- und Benutzermodells durch ein TUM
- Darstellung des Datenmodells in graphischer oder textueller Notation
- Darstellung relevanter Teile der übrigen Eingabemodelle mit einer Beschreibung der Kommandos
- Gegebenenfalls Beschreibung der Ausführung der generierten EMU-Systeme

Bei den einzelnen Beispielen sind also nicht immer alle Beschreibungsformen notwendig bzw. sinnvoll. Insbesondere wird das Layoutmodell nur bei den beiden Beispielen benötigt, bei denen ein Grapheditor erstellt wird (*TUM* und *Ad-hoc Workflows*). D.h. die Screenshots der Formulare zeigen hier stets das automatisch generierte Layout entsprechend den Standardlayoutregeln von EMUGEN (vgl. Abschnitte 3.3, 4.2.2).

### 6.2.1 EMU-System *TUM*

#### Anforderungsbeschreibung

Das Task-User-Modell (*TUM*) soll der graphischen Darstellung des EMU-Aufgaben- und Benutzermodells dienen. Die Anforderungen und eine Reihe von Beispielen sind in Abschnitt 5.1.6 beschrieben.

#### Datenmodell

Folgenden Produktionen bilden das EMU-Datenmodell zum Beispiel *TUM*:

```
TUM::=Phase*:Phasen Datenobjekt*:Datenobjekte
      Transition*:Transitionen  Akteur*:Akteure
Phase::=String:Name Aktivierungszustand
Aktivierungszustand::=Aktiv|Nichtaktiv
Datenobjekt::=String:Name Schreiber* Leser*
Schreiber::= Name->Akteur
Leser::= Name->Akteur
Transition::=Vorbereich* Nachbereich* Schreiber*
Vorbereich::=Name->Phase
Nachbereich::=String:Resultat Nachphase
Nachphase::=Name->Phase
Akteur::=String:Name Aktivphase* Expr:Aktivierungsbedingung
Aktivphase::= Name->Phase
```

Modelliert im EMU-Datenmodell, stellt `TUM` ein Tupel dar, welches aus Listen von Phasen, Datenobjekten, Transitionen und Akteuren besteht. Auch `Phase` ist ein Tupel und setzt sich aus den Komponenten `Name` und `Aktivierungszustand` zusammen. `Aktivierungszustand` ist eine Variante mit den beiden nicht weiter definierten Alternativen `Aktiv` und `Nichtaktiv`. Bei `Datenobjekt` werden als Komponenten der Name so-

wie die Liste der schreibenden und lesenden Akteure (`Schreiber` bzw. `Leser`) verwaltet. Sowohl `Schreiber` als auch `Leser` sind Referenztypen, die auf `Akteur` verweisen. Sie werden als Typ unterschieden, damit in der Präsentation die Darstellung der Kanten (schwarz bei `Schreiber`, grau bei `Leser`, vgl. Abbildung 5-5) unterschiedlich definiert werden kann. `Transition` besteht aus `Vorbereich*` (eine Liste von Referenzen auf die Phasen, die nach dem Schalten der Transition deaktiviert werden), `Nachbereich*` (die Phasen, die nach dem Schalten aktiviert werden) und `Schreiber*` (die Akteure, welche die Transition bzw. die zugehörige Aktion ausführen dürfen). Elemente des Nachbereichs (Typ `Nachbereich`) bestehen aus einer Referenz auf die zu aktivierende Phase (`Nachphase`) und dem Namen des entsprechenden Resultats. Ein Akteur hat einen Namen, eine Referenz auf die Phasen, während denen er aktiv ist (`Aktivphase*`) und eine Aktivierungsbedingung. Der Typ `Expr` wird im Anhang A definiert.

### Aufgabenmodell

Will man aus einem TUM ein von EMUGEN weiter zu verarbeitendes Aufgaben- und Benutzermodell erstellen, so ist dazu die Implementierung einer entsprechenden textuellen Ausgabe notwendig. Dies kann im Rumpf einer Aktion erfolgen, die dem IDT `TUM` zugeordnet wird.

```
actions of TUM{
  generieren [true] -> {:
   /* Kommandos zum Erzeugen
      des Aufgaben- und Benutzermodells  */
  :};
}
```

### Layoutmodell

Im Layoutmodell von TUM wird die graphische Repräsentation der abstrakten Sprachelemente in ähnlicher Weise festgelegt, wie dies in Kapitel 4 bei der Erstellung eines Grapheditors für Petrinetze bereits beschrieben wurde. Man beachte, dass sich der Begriff *Sprachelemente* nun auf die im Datenmodell des Beispiels spezifizierten IDTs, wie z.B. `Phase` und `Akteur`, bezieht und nicht etwa auf die EMU-Sprachelemente.

Zusätzlich zum Layoutmodell vom Beispiel *Petrinetze* muss festgelegt werden, dass die graphischen Knoten zur Darstellung der Akteure geeignet gezeichnet werden (vgl. Abbildung 5-5). Ferner ist im Layoutmodell zu definieren, dass ein schreibender Zugriff mit schwarzen und ein lesender Zugriff mit grauen Kanten dargestellt werden soll.

### Ausführung

Die Ausführung bzw. Verwendung eines EMU-Systems vom Typ *TUM* zur Modellierung und anschließenden Generierung eines weiteren EMU-Systems entspricht einem Bootstrapping-Schritt, der in Abbildung 6-1 skizziert und im Folgenden genauer beschrieben wird.

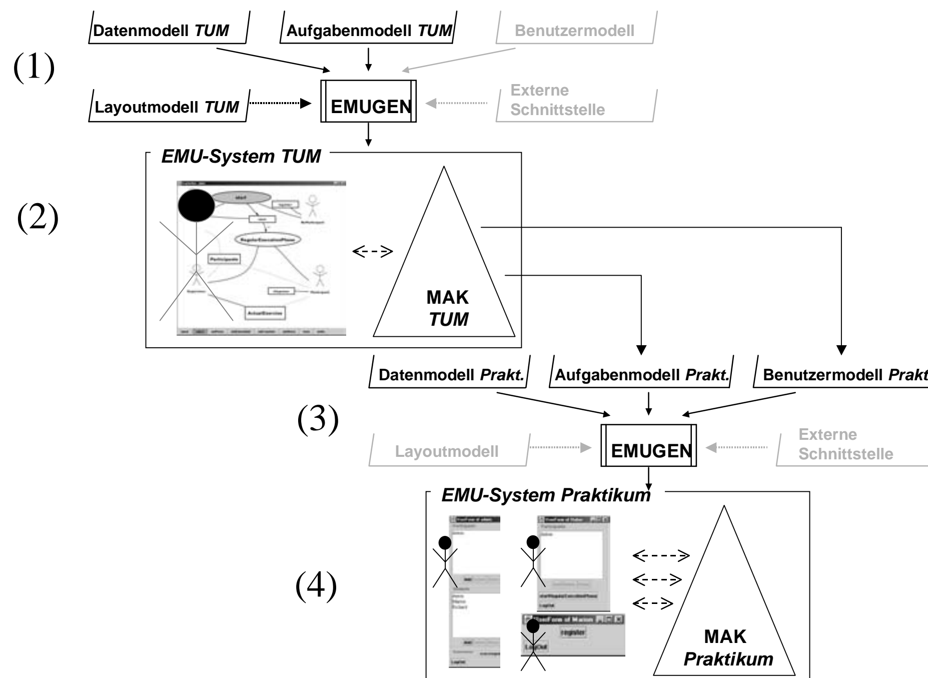


**Abbildung 6-1: Bootstrapping-Schritt mit EMUGEN**

Im Schritt (1) von Abbildung 6-1 erfolgt die Spezifikation und Generierung des EMU-Systems *TUM*. Im darauffolgenden Schritt (2) erstellt ein Benutzer des EMU-Systems *TUM* in der – im ersten Schritt spezifizierten – Diagrammsprache das Aufgaben- und Benutzermodell eines weiteren EMU-Systems, nämlich das in Abschnitt 6.2.4 besprochene Beispiel *Praktikum*. Bei der Erstellung der textuellen Aufgaben- und Benutzermodelle, die als Eingabe von EMUGEN im Schritt (3) erforderlich sind, kann der MAK des EMU-Systems *TUM* als Syntaxbaum wie bei einer herkömmlichen Übersetzung verwendet werden. EMUGEN unterstützt Anwendungen dieser Art dadurch, dass die Knotenklassen das Besuchermuster [GHJ96] implementieren (z.B. Methoden für Standardtraversierungen des MAKs).

Im Schritt (3) muss jedoch noch ein zugehöriges Datenmodell definiert werden, weil dies nicht durch ein TUM spezifiziert werden kann. Anschließend kann im Schritt (4) das nun diagrammsprachlich spezifizierte Mehrbenutzersystem ausgeführt werden.

### 6.2.2 EMU-System *Geldausgabeautomat*

#### Anforderungsbeschreibung

> *Der Geldausgabeautomat soll die übliche Funktionalität aufweisen. Nach der Anmeldung, die durch Karten- und Geheimnummerangabe erfolgt, wird der Kontostand angezeigt. Nun kann eine Überweisung oder eine Bargeldauszahlung erfolgen. Bei der Überweisung sind die üblichen Daten wie Empfängername, Empfängerkontonummer, der Betrag und der Verwendungszweck anzugeben. Von jedem Zustand aus soll es möglich sein, die Bearbeitung abzubrechen. Nach einer fehlerhaften Bedienung (z.B. nach falscher Geheimnummer) soll eine geeignete Fehlerbeschreibung angezeigt werden.*

**Datenmodell**

Folgenden Produktionen bilden das EMU-Datenmodell zum Beispiel *Geldausgabeautomat*:

```
Geldausgabeautomat::= Integer:Geheimnummer
                       Integer:Kontostand
                       Überweisungsdaten
                       Integer:Auszahlungsbetrag
                       String:Fehlerbeschreibung
                       Bankkunde
Überweisungsdaten::=   String:Empfängername
                       Integer:Empfängerkonto
                       Integer:Betrag
                       Text:Verwendungszweck
```

Im Datenmodell sind alle Daten, die beim Dialog zwischen Kunde und Geldautomat ein- und auszugeben sind, zu modellieren. Da dieses Beispiel eine eher ablauf- als datenorientierte Struktur aufweist, ist das Datenmodell verhältnismäßig einfach. Es besteht aus zwei Tupelproduktionen für die Tupel `Geldausgabeautomat` und `Überweisungsdaten`. Ferner ist der eigentliche Bankkunde als Teil des Datenmodells definiert. Da der Anmeldevorgang in diesem Beispiel auf eine spezielle Art und Weise mit Karte und Geheimnummer erfolgen soll, werden keine zusätzlichen Informationen zur Anmeldung benötigt. `Bankkunde` ist daher ein Nulltupel, das nicht explizit definiert werden muss. Es dient im Benutzermodell als einziger Benutzertyp.

**Aufgaben- und Benutzermodell als TUM**

In diesem Beispiel bietet es sich an, das Aufgaben- und Benutzermodell durch ein TUM zu beschreiben. Das zum Starttyp `Geldausgabeautomat` zugeordnete TUM ist in Abbildung 6-3 dargestellt. Da es sich in diesem Beispiel um ein Einbenutzersystem handelt, werden keine Akteure eingezeichnet. Wenn eine Phase eingenommen ist (z.B. der Initialzustand `Bereit`), so werden alle Datenobjekte, die mit der jeweiligen Phase verbunden sind (z.B. ist das Datenobjekt `Geheimnummer` mit der Phase `Geheimnummereingabe` verbunden) dem Benutzer angezeigt. Ferner wird dem Benutzer eine Aktion visualisiert, wenn es eine Transition *t* gibt, die mit dem Namen der Aktion markiert ist und eine der Phasen, die sich im Vorbereich von *t* befinden (vgl. Abschnitt 6.2.1) aktiv ist. Beispielsweise wird im Initialzustand die Aktion `eingebenKarte` visualisiert, weil die Phase `Bereit` im Vorbereich einer Transition liegt, die mit `eingebenKarte` markiert ist.

Man beachte ferner, dass einige Transitionen mit `abbrechen` markiert sind und (mit einer Ausnahme) einen leeren Nachbereich besitzen. Gemäß der üblichen Petrinetz-Semantik (vgl. Abschnitt 5.3, [Esp95]) beenden Transitionen mit leerem Nachbereich lediglich die Phasen (Stellen) des Vorbereichs.

Die Phase `Bereit` ist die einzige Initialphase des TUMs aus Abbildung 6-3. Die Aktion `eingebenKarte` beendet die Phase `Bereit` und aktiviert je nach Resultat (vgl. Abschnitt 5.1.1) weitere Phasen. Liefert `eingebenKarte` das Resultat `Ok` (eine entsprechende Überprüfung der Karte kann im Aktionskommando erfolgen), dann werden die Phasen `Geheimnummereingabe` und `Fehlerfrei` aktiviert. Anderenfalls liefert `einge-`

**benKarte** das Resultat **Fehler**, wodurch die Phasen **Fehler** und **Bereit** aktiviert werden (identische Namen für Aktionsresultate und Phasen sind unproblematisch). Während der Phase **Geheimnummereingabe** wird dem Benutzer ein schreibender Zugriff auf das Datenobjekt **Geheimnummer** gewährt (schwarze Kante).

Während der Phase **Fehlerfrei** erfolgt keine zusätzliche Visualisierung eines Datenobjekts. **Fehlerfrei** bildet zusammen mit der Phase **Fehler** einen orthogonal zum übrigen Dialog ablaufenden *Fehlerdialog*. D.h. während der Ausführung (genauer: nach Ausführung der Aktion **eingebenKarte**) ist immer genau eine dieser beiden Phasen aktiv. Ist die Phase **Fehler** aktiv, so erhält der Benutzer einen lesenden Zugriff (graue Kante) auf das Datenobjekt **Fehlerbeschreibung**. Dies erfolgt zusätzlich zu der Visualisierung weiterer Datenobjekte aufgrund der jeweils aktuellen Phase des eigentlichen *Fachdialogs*.

Wurde beispielsweise eine falsche Geheimnummer angegeben, so bleibt einerseits die Phase **Geheimnummereingabe** aktiv, andererseits wird auch die Phase **Fehler** aktiviert. Daher werden die Datenobjekte **Geheimnummer** und **Fehlerbeschreibung** gemeinsam dargestellt. Der Wert von **Fehlerbeschreibung** wird dabei abhängig vom jeweils eingetretenen Fehler im Aktionskommando gesetzt (z.B. „Falsche Geheimnummer“). Das Benutzerformular in diesem Zustand wird in Abbildung 6-2 dargestellt.

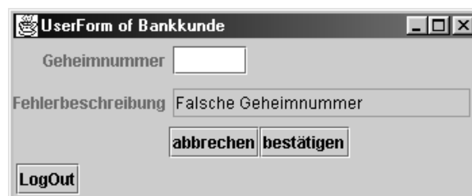


**Abbildung 6-2: Geldausgabeautomat nach fehlerhaften Geheimnummereingabe**

Wurde die richtige Geheimnummer eingegeben (die Aktion **bestätigen** liefert in diesem Fall das Resultat **Ok**), so wird die Phase **Auswahl** aktiviert und die Phase **Fehlerfrei** bleibt aktiviert. Während der Phase **Auswahl** kann der Benutzer zu den Phasen **Auszahlung** oder **Überweisung** wechseln. Dort kann der Kunde das jeweils zugehörige Datenobjekt (**Betrag** bzw. **Überweisungsdaten**) eintragen und den Vorgang mit der Aktion **bestätigen** abschließen.

Liefert **bestätigen** das Resultat **Fehler**, so bleibt die jeweilige Phase (**Auszahlung** bzw. **Überweisung)** aktiv und im Fehlerdialog erfolgt ein Übergang in die Phase **Fehler**. Beispielsweise zeigt Abbildung 6-4 einen Fehler bei der Überweisung. Die Aktion **bestätigen** kann also nicht nur zu einem Übergang im Fachdialog (z.B. von der Phase Auszahlung zur Phase Auswahl), sondern auch im Fehlerdialog (z.B. von der Phase **Fehler** zur Phase **Fehlerfrei**) führen, d.h. zwei Transitionen gleichzeitig schalten (vgl. die Definition der Zustandsübergangsfunktion in Abschnitt 5.3).

Man beachte, dass die Unterscheidung bzw. Modellierung des Fach- und Fehlerdialogs eine (in diesem Beispiel angewandte) Technik zur Beschreibung von Dialogen mit TUMs und kein EMU-Sprachelement darstellt.

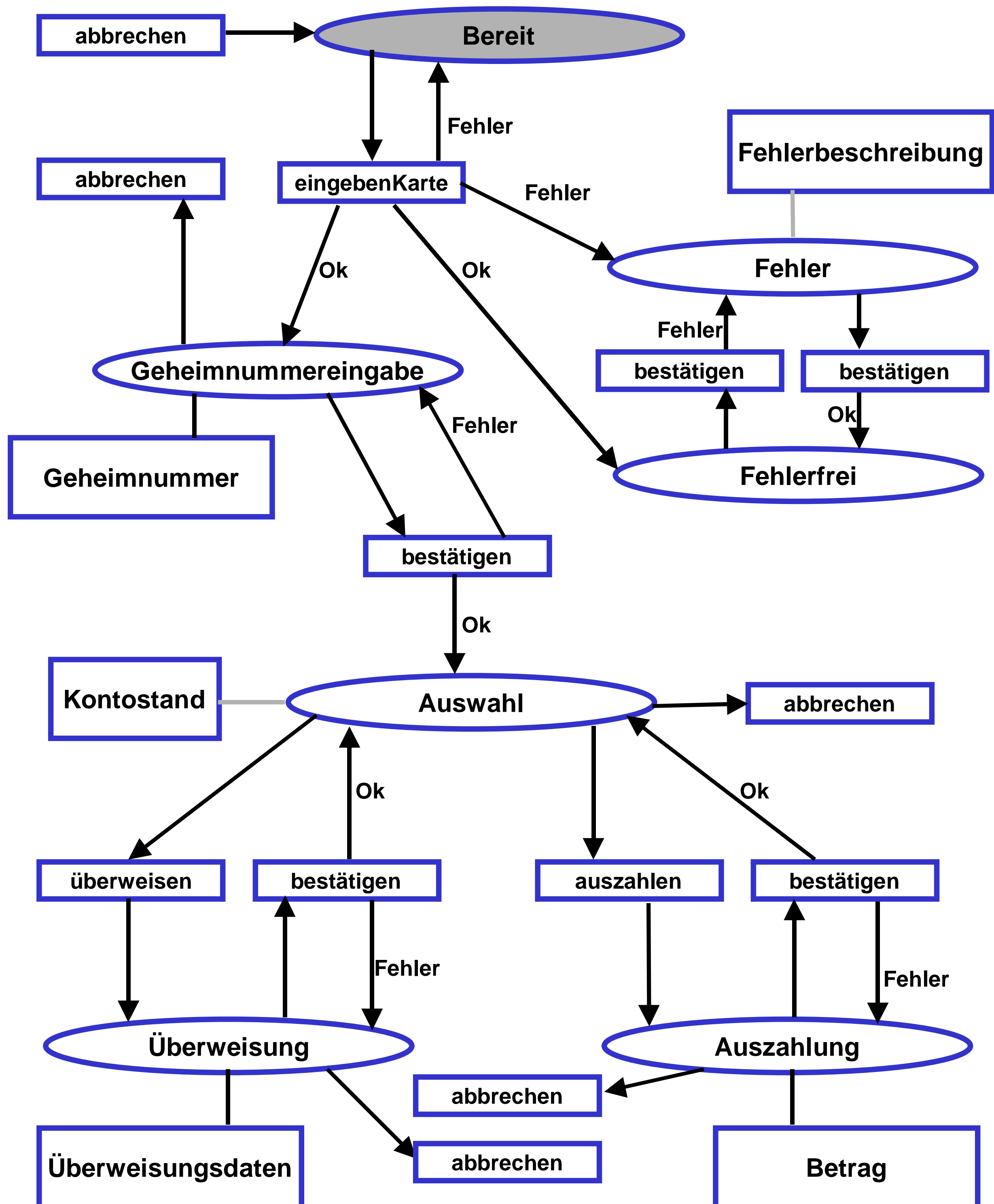


**Abbildung 6-3: TUM für das Beispiel Geldausgabeautomat**

Abbildung 6-4: Geldausgabeautomat nach ungedeckter Überweisung

### Aufgabenmodell

Da das im vorherigen Abschnitt besprochene TUM bereits das Aktivitätsnetz zu weiten Teilen zeigt, soll hier nur noch die Implementierung der Aktionen des EMU-Aufgabenmodells dargestellt werden.

```
actions of Geldausgabeautomat {
  abbrechen      [true]     -> {::};
  eingebenKarte  [true]     -> {:
    if (KarteOk()) return 0;
    Fehlerbeschreibung.setValue("Karte fehlerhaft");
    return 1;
  :} Ok | Fehler ;
  bestätigen [true] () -> {:
   if (GeheimnummerEingabe) {
     if (GeheimnummerOk()) return 0;
     Fehlerbeschreibung.setValue("Falsche Geheimnummer");
   }
   if (Überweisung) {
     int betrag =
       Überweisungsdaten.getBetrag().getValue();
     if (decktKonto(betrag)) {
       Kontostand.setValue(Kontostand.getValue()-betrag);
       return 0;
     }
   }
   if (Auszahlung) {
     int betrag=Auszahlungsbetrag.getValue();
     if (decktKonto(betrag)) {
       Kontostand.setValue(Kontostand.getValue()-betrag);
      return 0;
     }
   }
   return 1;
 :} Ok | Fehler;
 überweisen      [true]     -> {::};
```

```
     auszahlen       [true]     -> {::};
    }
```

Zur Fehlererkennung im Rumpf der Aktion `bestätigen` werden die Methoden `KarteOk()`, `GeheimnummerOk()` und `decktKonto(int)` benutzt, die als Methoden der Knotenklasse `Geldausgabeautomat` geeignet zu implementieren sind.

**Benutzermodell**

Im Benutzermodell werden die „Zugriffskanten" des TUMs aus Abbildung 6-3 textuell dargestellt.

```
access on Geldausgabeautomat {
  [Bereit]  exec eingebenKarte;
  [Fehler]
    read Fehlerbeschreibung,exec bestätigen;
  [Auswahl]
    read Kontostand,exec überweisen,
    exec auszahlen,exec abbrechen;
  [Überweisung]
    write Überweisungsdaten,exec bestätigen,exec abbrechen;
  [Auszahlung]
    write Auszahlungsbetrag,exec bestätigen,exec abbrechen;
  [GeheimnummerEingabe]
    exec bestätigen,write Geheimnummer,exec abbrechen;
}
User::= Bankkunde()
```

**Layoutmodell**

Eine Anpassung der allgemeinen Layoutregeln ist zur prototypischen Ausführung des Geldausgabeautomaten (vgl. Abbildung 6-2 und Abbildung 6-4) nicht erforderlich.

### 6.2.3 EMU-System Talk

**Anforderungsbeschreibung**

> *Zwei Personen sollen durch ein einfaches textuelles Eingabemedium (Kanal) synchron miteinander kommunizieren können. Dabei hat immer genau eine der Personen Schreibrecht auf den gemeinsamen Kanal, während die jeweils andere Person nur ein Leserecht besitzt. Nur die Person mit dem Schreibrecht kann die Kontrolle abgeben, d.h. das Schreibrecht der jeweils anderen Person übertragen.*

**Datenmodell**

`Talk` ist ein Tupel, das aus den beiden Personen und dem Kanal besteht. `Person` dient im Benutzermodell als Benutzertyp.

```
Talk::=Person:A Person:B Text:Kanal
Person::=String:Name
```

**Aufgabenmodell**

Im Aufgabenmodell wird eine Aktion zur Abgabe der Kontrolle und ein aus zwei Phasen bestehendes Aktivitätsnetz definiert. Dabei wird festgelegt, dass Person `A` zunächst schreiben darf.

```
actions of Talk {
  abgeben [true] -> {::};
}
actnet of Talk {
  ->A_schreibt
  A_schreibt->abgeben->B_schreibt
  B_schreibt->abgeben->A_schreibt
}
```

**Benutzermodell**

Im Benutzermodell erfolgt die Definition der phasenabhängigen Zugriffsfestlegung und der Benutzertypen.

```
access on Talk {
  [User]                     read Kanal;
  [A_schreibt & user==A]     write Kanal, exec abgeben;
  [B_schreibt & user==B]     write Kanal, exec abgeben;
}
User::=Person(Name)
```

**Ausführung**

An dieser Stelle soll der in Abbildung 5-12 skizzierte Zugriff eines Benutzers auf ein Mehrbenutzersystem exemplarisch diskutiert werden. In Abbildung 6-5 wird dazu das Formular des Benutzers `A` während der Phase `A_schreibt` gezeigt.

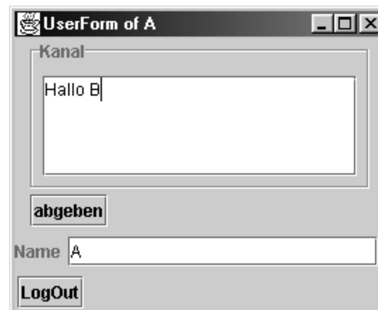


**Abbildung 6-5: Talkprogramm mit Kontrolle von A**

Neben dem im Benutzermodell explizit definierten Zugriff auf die Wurzelknoten des MAKs (schreibend auf `Kanal`, ausführend auf `abgeben`) erscheint zusätzlich die Komponente `Name` im Formular. Das ist der Fall, weil jeder Benutzer nicht nur Zugriff auf den Wurzelknoten des MAKs, sondern auch auf den „eigenen" Benutzerknoten (vgl. Abbildung 5-12) besitzt. Soll dieser Zugriff verhindert werden, so kann das Benutzermodell einfach durch die folgende Zugriffsvereinbarung für `Person` ergänzt werden:

```
access on Person {
}
```

Damit erhält kein Benutzer Zugriff auf die Komponenten von `Person`. Auf die gleiche Weise könnte man auch den Zugriff auf den Wurzelknoten verhindern. Die Benutzer könnten dann nur auf den „eigenen" Knoten zugreifen (vgl. Fall C in Abbildung 5-12). Diese Technik wird im Beispiel *Ad-hoc Workflows* eingesetzt.

### *6.2.4* EMU-System *Praktikum*

**Anforderungsbeschreibung**

> *Die Benutzer des Praktikums sind Studenten und ein Praktikumsleiter. Zu Beginn des Praktikums können sich Studenten mit Vordiplom als Teilnehmer anmelden. Zu Studenten sind dabei die üblichen Daten, wie etwa Name und Matrikelnummer usw., zu führen. Der Praktikumsleiter kann die Anmeldephase beenden und die Bearbeitungsphase einleiten. Während der Bearbeitungsphase stellt der Praktikumsleiter eine aktuelle Aufgabe, die von den Teilnehmern durch Abgabe einer Lösung bearbeitet wird. Anschließend kann der Praktikumsleiter die Lösung bewerten.*

**Datenmodell**

Folgenden Produktionen bilden das EMU-Datenmodell zum Beispiel *Praktikum*:

```
Praktikum::=Student*:Studenten Teilnehmer*:Teilnehmer
            Praktikumsleiter String:AktuelleAufgabe
Student::=String:Name String:Vorname Geschlecht
          Integer:Matrikelnummer Stand
Geschlecht::=Männlich | Weiblich
Stand::=Vordiplom | Hauptdiplom
Vordiplom::=String:Datum
Teilnehmer::=StudentRef Abgabe*:Abgaben Text:Arbeitsbereich
StudentRef::=Name->Student
Praktikumsleiter::=String:Name
Abgabe::=String:Aufgabe Text:Lösung Bewertung
Bewertung::=SehrGut|Gut|Befriedigend|Ausreichend|Ungenügend
```

Hier ist die Verwendung der Referenz `StudentRef` als Komponente im Tupel `Teilnehmer` zu beachten. Dies entspricht in etwa der Verwendung von Fremdschlüsseln bei relationalen Datenbanken zur Vermeidung von Redundanzen. Die Tupel `Student` und `Praktikumsleiter` dienen im Benutzermodell als Benutzertypen.

**Aufgaben- und Benutzermodell als TUM**

Wie schon beim Beispiel *Geldautomat*, bietet es sich auch hier an, die Zusammenhänge des Aufgaben- und Benutzermodells und damit den Ablauf eines EMU-Systems vom Typ *Praktikum* (genauer: die Zugriffe und Abläufe, die sich auf den Starttyp `Praktikum` und damit zur Laufzeit auf den Wurzelknoten beziehen) durch ein TUM graphisch darzustellen (Abbildung 6-6).

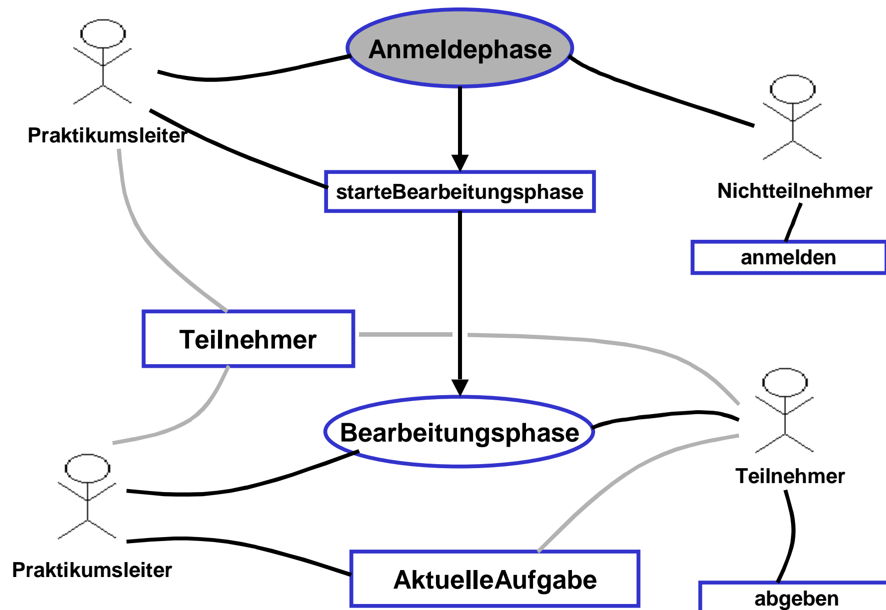


**Abbildung 6-6: TUM für das Beispiel *Praktikum***

Während der Initialphase (`Anmeldephase`) kann der „damit" verbundene Akteur mit der Bezeichnung `Praktikumsleiter` (in diesem Fall handelt es sich um genau einen Benutzer) das Datenobjekt `Teilnehmer` lesen und `Anmeldephase` durch Ausführung der Aktion `starteBearbeitungsphase` beenden. Dadurch wird gleichzeitig `Bearbeitungsphase` aktiviert. Man beachte, dass mehrere Akteure in einem TUM denselben Namen besitzen können. Der mit `Anmeldephase` verbundene Akteur beschreibt die Zugriffe des `Praktikumsleiters`, wenn `Anmeldephase` aktiv ist. Der gleichnamige Akteur, der mit `Bearbeitungsphase` verbunden ist, beschreibt die Zugriffe desselben Benutzers, wenn `Bearbeitungsphase` aktiv ist. D.h. während `Bearbeitungsphase` kann der Praktikumsleiter auf `Teilnehmer` lesend (graue Kante) und auf `AktuelleAufgabe` schreibend (schwarze Kante) zugreifen.

Der Akteur `Nichtteilnehmer` (hier handelt es sich im Allgemeinen um mehrere Benutzer, nämlich um alle Studenten mit Vordiplom) kann während `Anmeldephase` die Aktion `anmelden` ausführen und wird damit Teilnehmer des Praktikums. Technisch wird dazu im Aktionsrumpf von `anmelden` die Liste der Teilnehmer um ein neues Element ergänzt.

Man beachte, dass die Akteurbezeichnung `Nichtteilnehmer` keine ausführungstechnische Bedeutung besitzt, weil die zugehörige Bedingung (vgl. Abschnitt 5.1.6) wie folgt definiert ist:

```
Student.Stand.Vordiplom & !(Student in Teilnehmer)
```

Der Akteur `Nichtteilnehmer` beschreibt einen Benutzer, der vom Benutzertyp `Student` ist, dessen Variantenkomponente `Stand` die aktuelle Ausprägung `Vordiplom` besitzt und der nicht bereits als Teilnehmer geführt ist. Die Bedeutung von Ausdrücken

wie `Student in Teilnehmer` wurde in Abschnitt 5.3 definiert. Da dieser Akteur mit der Phase `Anmeldephase` verbunden ist, ist er nur aktiv, wenn diese Phase aktiv ist.

Nachdem ein Benutzer, der die obige Akteursbedingung erfüllt, sich angemeldet hat, erfüllt er die Akteursbedingung nicht mehr, weil er nun Teilnehmer ist. Entsprechend ändert sich sein Zugriffsadapter und damit seine Benutzungsoberfläche. Sobald die Bearbeitungsphase aktiv ist, ändert sich seine Benutzungsoberfläche erneut, weil nun `Teilnehmer`, die `AktuelleAufgabe` und die Aktion `abgeben` sichtbar sind.

In Abbildung 6-7 wird der Zusammenhang zwischen den Akteuren und den zugehörigen Formularen gezeigt. Im Gegensatz zu Abbildung 6-6 werden hier anstelle der Akteure die jeweiligen Formulare der entsprechenden Benutzer gezeichnet.

### Aufgabenmodell

Das folgende textuelle Aufgabenmodell stellt im Wesentlichen dieselben Aspekte wie Abbildung 6-6 dar. Zusätzlich sind die Aktionskommandos zu implementieren, die hier aus Platzgründen nicht dargestellt werden.

```
actions of Praktikum {
  starteBearbeitungsphase [true] -> {::};
  anmelden [true] () -> {:
    /* Erstellen und Einfügen eines neuen Teilnehmers */
  :};
  abgeben [true] () -> {:
    /* Erstellen und Einfügen einer neuen Abgabe */
  :};
}

actnet of Praktikum {
  ->Anmeldephase
  Anmeldephase->starteBearbeitungsphase->Bearbeitungsphase
}
```

### Benutzermodell

Im Benutzermodell werden zusätzlich zum Zugriff auf den Wurzelknoten (vgl. Abbildung 6-6) die Zugriffe auf Knoten der Typen `Teilnehmer` und `Abgabe` definiert (nicht ersichtlich in Abbildung 6-6, weil sich das dort dargestellte TUM nur auf den Starttyp bezieht). Bei `Teilnehmer`-Knoten kann der Praktikumsleiter auf die Komponente `Abgaben` lesend und der zugehörige Student (bestimmt durch den Wert der Referenz `StudentRef`) zusätzlich auf die Komponente `Arbeitsbereich` schreibend zugreifen. Bei einem Knoten vom Typ `Abgabe` kann der Praktikumsleiter zum Verbessern und Bewerten auf die Komponenten `Lösung` und `Bewertung` schreibend zugreifen. Ein Student (bzw. ein Teilnehmer) kann auf die Komponenten eines `Abgabe`-Knotens nur lesend zugreifen. Man beachte, dass durch die allgemeine Berechnung der Zugriffsadapter und der Zugriffsdefinition für `Teilnehmer` sichergestellt wird, dass nur der zugehörige Student Zugriff auf „seinen“ `Abgabe`-Knoten erhält.

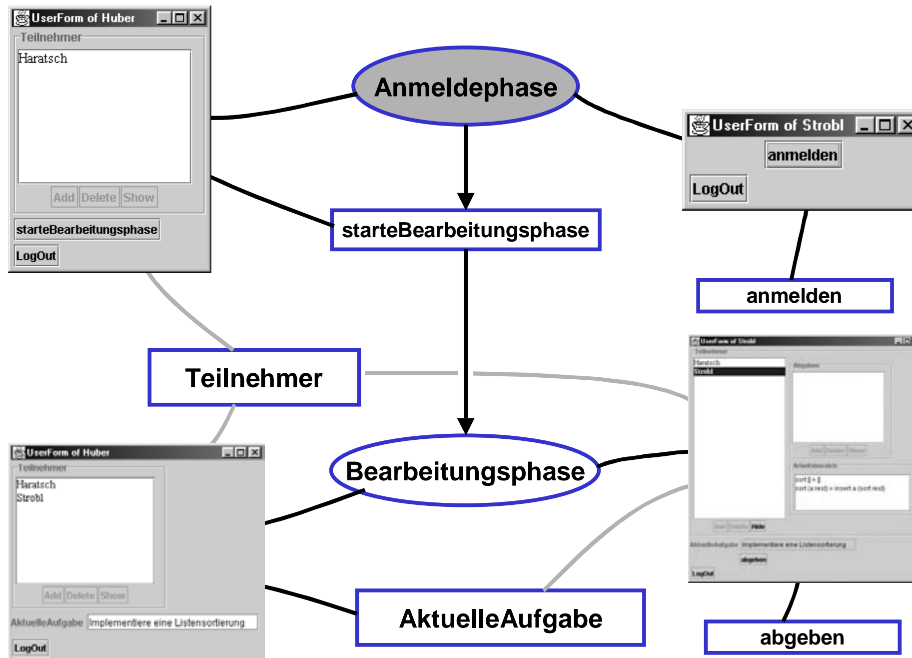


**Abbildung 6-7: TUM mit Formularen aus einer Beispielausführung**

```
access on Praktikum {
  [Anmeldephase & Praktikumsleiter]
    exec starteBearbeitungsphase, read Teilnehmer;
  [Anmeldephase & Student.Stand.Vordiplom &
   !(Student in Teilnehmer]
    exec anmelden;
  [Bearbeitungsphase & Student in Teilnehmer]
    read AktuelleAufgabe, read Teilnehmer, exec abgeben;
  [Bearbeitungsphase & Praktikumsleiter]
    write AktuelleAufgabe, read Teilnehmer;
}
access on Teilnehmer {
  [Praktikumsleiter]
    read Abgaben;
  [User==StudentRef]
    read Abgaben, write Arbeitsbereich;
}
access on Abgabe {
  [Praktikumsleiter]
    read Aufgabe, write Lösung, write Bewertung;
  [Student]
    read Aufgabe, read Lösung, read Bewertung;
}

access on Student {}
access on Praktikumsleiter {}

User::=Praktikumsleiter(Name)|Student(Name)
```

### 6.2.5 EMU-System *Ad-hoc Workflows*

**Anforderungsbeschreibung**

*Die Ablaufspezifikation von Workflows kann nicht immer statisch definiert werden [Aal98]. Daher soll ein EMU-System entwickelt werden, welches es ermöglicht, einen dynamischen Workflow zu verwalten. Dabei ist die Ablaufspezifikation, d.h. die kausalen Abhängigkeiten der Aufgaben und die Aufgabenzuordnung zu den verantwortlichen Benutzern, zur Ausführungszeit (re-)definierbar. Man spricht dabei von Ad-hoc Workflows (AHWs) [Aal98].*

*AHWs bestehen aus Aufgaben, Phasen und einem Team von Benutzern. Die Ablaufspezifikation soll durch ein petrinetzähnliches Netzwerk von Phasen (Stellen) und Aufgaben (Transitionen) definiert werden. Durch das Netzwerk soll auch die Zuordnung von Aufgaben zu den jeweils verantwortlichen Benutzern modelliert werden. Bei der Ausführungssemantik soll in Anlehnung an [Obe96] die (Petrinetz-)Marke als Datenobjekt realisiert werden, welches von den jeweiligen Aufgabenbearbeitern zu bearbeiten ist. Die Struktur des Datenobjekts soll dabei ebenfalls dynamisch zur Laufzeit sein.*

*Um die möglichen Änderungen an der Ablaufspezifikation ohne Informationsverlust durchführen zu können, ist zwischen einer Definitions- und Bearbeitungsphase zu unterscheiden. Während der Definitionsphase kann durch den Administrator des Workflows die Ablaufspezifikation geändert werden. Während der Bearbeitungsphase werden die Aufgaben den jeweiligen Benutzern zugeordnet (vgl. Arbeitslisten bei WFMS [WfM96]) und können von ihnen bearbeitet werden.*

**Datenmodell**

Folgenden Produktionen bilden das EMU-Datenmodell zum Beispiel *Ad-hoc Workflows*:

```
AHW::=Phase*:Phasen Aufgabe*:Aufgaben Bearbeiter*:Bearbeiter
      Workflowadministrator
Phase::=String:Name Markierung
Aufgabe::=String:Name Vorphase Nachphase Verantwortlich
Vorphase::=Name->Phase
Nachphase::=Name->Phase
Markierung::=Undefiniert|Datenobjekt
Bearbeiter::=String:Name ZugewieseneAufgabe*:Arbeitsliste
Workflowadministrator::=String:Name
ZugewieseneAufgabe::=String:Name Datenobjekt
ZVorphase ZNachphase
Verantwortlich::=Name->Bearbeiter
ZVorphase::=Name->Phase
ZNachphase::=Name->Phase

Datenobjekt::=String:Name Objekttyp
Objekttyp::=Tupelobjekt
           | Basisobjekt
Tupelobjekt::=Datenobjekt*
Basisobjekt::=String:Wert
```

Aufgrund der geforderten, petzinetzähnlichen Ablaufspezifikation (vgl. Abschnitt 2.1.4) können wir vom Datenmodell aus Beispiel 5-2 *Petrinetze* (vgl. Abbildung 5-10) ausgehen. Dieses wird um die Verwaltung der Benutzer des Workflows und ihre Zuordnung zu den einzelnen Aufgaben erweitert.

Die Arbeitslisten sind dabei den jeweiligen Bearbeitern zugeordnet, d.h. formal als Komponente des Benutzertyps `Bearbeiter`. Die Arbeitslisten der einzelnen Bearbeiter werden aufgrund der aktuellen Ablaufspezifikation und der jeweils aktuellen Phasenmarkierung berechnet und verteilt.

Daher wird zwischen den zur Definition der Ablaufspezifikation verwendeten Aufgaben (das Tupel `Aufgabe` entspricht der Petrinetz-Transition) und den an die Bearbeiter zugewiesenen Aufgaben (Tupel `ZugewieseneAufgabe`) unterschieden.

Zur Vereinfachung gehen wir im Folgenden davon aus, dass jeder Aufgabe als Vor- und Nachbereich nur höchstens eine Phase (`Aufgabe`-Komponenten `Vorphase` und `Nachphase`) zugeordnet werden kann. Dies liegt nicht daran, weil es mit den EMU-Sprachelementen nicht möglich wäre, den allgemeinen Fall zu modellieren, sondern weil ansonsten die Beispielbeschreibung (sie beinhaltet immerhin eine komplette, wenn auch einfache Workflowmodellierungssprache), infolge einer Reihe dann notwendiger Fallunterscheidungen, zu komplex für eine vollständige Betrachtung wäre.

Aufgaben, deren Vorphase markiert ist, werden im Folgenden als *aktiv* bezeichnet. Bei der Aufgabenverteilung werden für alle aktiven Aufgaben der Ablaufspezifikation (also die Elemente der Liste `AHW.Aufgaben`), für die ein Bearbeiter verantwortlich ist (verwaltet durch den Wert der Referenz `Aufgabe.Verantwortlich`), Elemente vom Typ `ZugewieseneAufgabe` erzeugt und als Elemente der Liste `Bearbeiter.Arbeitsliste` bei dem entsprechenden Bearbeiter eingefügt. Jede zugewiesene Aufgabe erhält denselben Namen wie die *Ursprungsaufgabe* aus der Ablaufspezifikation.

Die Aufgabenverteilung wird immer dann vorgenommen, wenn vom Definitionszustand in den Bearbeitungszustand gewechselt wird oder wenn ein Bearbeiter eine ihm zugewiesene Aufgabe abschließt (näheres dazu unten im Aufgabenmodell).

Beim Tupel `ZugewieseneAufgabe` wird eine Kopie des Datenobjekts der Vorphase der zugehörigen Ursprungsaufgabe in der Komponente `Datenobjekt` verwaltet. Ferner werden die Vor- und Nachphase der Ursprungsaufgabe als Komponenten verwaltet. Wir benutzen dazu jedoch die Selektoren bzw. Typbezeichner `ZVorphase` bzw. `ZNachphase`, um sie in der diagrammsprachlichen Darstellung des Ausführungszustands von den Vor- bzw. Nachphasen der Ablaufspezifikation (d.h. von `Aufgabe.Vorphase`, `Aufgabe.Nachphase`) unterscheiden zu können.

Ähnlich, wie in diesem Beispiel ein Teil der Ausführungsemantik (d.h. die Ablaufspezifikation des Workflows) erst zur Laufzeit definiert wird, kann – laut Anforderung – auch die Struktur des Datenobjekts nicht statisch festgelegt werden. Unter Verwendung von Varianten ist es jedoch kein Problem, eine einfache, strukturierte Datendefinitionssprache im EMU-Datenmodell selbst nachzubilden (vgl. die Produktion von `Datenobjekt`).

**Aufgabenmodell**

Im Aufgabenmodell von *Ad-hoc Workflows* sind die in der Anforderungsanalyse beschriebenen Definitions- und Bearbeitungsphase sowie entsprechende Einleitungsaktio-

nen (**definieren**, **bearbeiten**) notwendig. Diese werden dem Starttyp **AHW** zugeordnet.

Zum Tupel **ZugewieseneAufgabe** gibt es eine Aktion **abschließen**. Wird diese vom jeweiligen Aufgabenbearbeiter ausgeführt, so schaltet die entsprechende Ursprungsaufgabe, d.h. die Markierung der Vorphase wird entfernt bzw. auf **Undefiniert** gesetzt und die Nachphase wird mit dem Datenobjekt der zugewiesenen Aufgabe markiert. Nun werden alle Aufgaben aktiviert, deren Vorphase die nun neu markierte Phase (also die Nachphase der schaltenden Phase) ist und alle Aufgaben deaktiviert, deren Vorphase die nun unmarkierte Phase darstellt. Ferner sind die Arbeitslisten, d.h. die an die Bearbeiter zugewiesenen Aufgaben entsprechend zu aktualisieren.

```
actions of AHW {
  bearbeiten [true] -> {:
    verteilen();
 :};
  definieren [true] -> {:
    entziehen();
 :};
}
```

Die beiden Methoden **verteilen()** (erzeugt Elemente vom Typ **ZugewieseneAufgabe** und verteilt sie an die Bearbeiter) und **entziehen()** (löscht alle Elemente vom Typ **ZugewieseneAufgabe**) müssen als Instanzmethoden der Knotenklasse **AHW** geeignet implementiert werden.

Ein Bearbeiter kann das Datenobjekt einer ihm zugewiesenen Aufgabe bearbeiten und die Aufgabenbearbeitung durch die Aktion **abschließen** beenden.

```
actions of ZugewieseneAufgabe {
  abschließen [true] -> {:
    ZVorphase.getValue().getMarkierung().setUndefiniert();
    ZNachphse.getValue().getMarkierung().setDatenobjekt();
    ZNachphse.getValue().getMarkierung().
      getDatenobjekt().setCopy(Datenobjekt);
    root.verteilen();
 :};
}
```

Im Rumpf von **ZugewieseneAufgabe.abschließen** wird zunächst die Markierung der Vorphase entfernt, dann die Nachphase mit dem (i.d.R. vom Bearbeiter verändertem) Datenobjekt markiert und schließlich werden die Aufgaben neu verteilt.

Die in der Anforderungsbeschreibung genannten Definitions- und Ausführungsphasen werden durch ein, dem Starttyp **AHW** zugeordneten, Aktivitätsnetz modelliert.

```
actnet of AHW  {
  -> Definitionsphase
 Definitionsphase -> bearbeiten ->Bearbeitungsphase
 Bearbeitungsphase-> definieren ->Definitionsphase
}
```

**Benutzermodell**

Die Benutzertypen von *Ad-hoc Workflows* sind `Bearbeiter` und `Workflowadministrator`. Während der Definitionsphase kann der Workflowadministrator die Ablaufspezifikation, d.h. die Komponenten `Phasen`, `Aufgaben` und `Bearbeiter` des Starttyps `AHW` bearbeiten und die Aktion `bearbeiten` ausführen. Während der Bearbeitungsphase kann die Ablaufspezifikation nur gelesen und die Aktion `definieren` ausgeführt werden.

```
access on AHW {
  [Definitionsphase & Workflowadministrator]
    write Phasen, write Aufgaben,
    write Bearbeiter, exec bearbeiten;
  [Bearbeitungsphase & Workflowadministrator]
    read Phasen, read Aufgaben,
    read Bearbeiter, exec definieren;
}
access on ZugewieseneAufgabe {
  [Bearbeiter]
    read Name, write Datenobjekt,exec abschließen;
}
access on Bearbeiter {
  [Bearbeiter]
    read Arbeitsliste;
}
users::=Bearbeiter(Name)|Workflowadministrator(Name)
```

Man beachte, dass durch die obige Zugriffsdefinition für `AHW` ein Bearbeiter keinen Zugriff auf den Wurzelknoten (`AHW` ist Starttyp) des MAK hat. D.h. für Benutzer vom Typ `Bearbeiter` gilt Fall C von Abbildung 5-12. Durch die Zugriffsbeschränkung auf `ZugewieseneAufgabe` wird verhindert, dass ein Bearbeiter die Vor- bzw. Nachphase verändern kann.

**Layoutmodell**

Im Layoutmodell ergeben sich zunächst ähnliche Anpassungen wie in Beispiel 5-2 *Petrinetze* (vgl. Abbildung 5-11), etwa um die Phasen oval und die Aufgaben rechteckig zu zeichnen. Weiterhin bietet es sich an, ein Bild vom Bearbeiter als Knotenmarkierung zu verwenden und zu kennzeichnen, ob ein Bearbeiter aktuell angemeldet ist oder nicht. Durch die folgende Layoutregel kann der Entwickler diese Anforderung implementieren.

```
node Bearbeiter {:
  IMAGE_FILE=ImageManager.getImage(Name);
  if (Login.Aktiv)
    OUTLINE_STROKE=5;
  else
    OUTLINE_STROKE=0;
:}
```

Die Verwaltung und Belegung der Bilder der Bearbeiter erfolgt durch eine Hilfskomponente `ImageManager`. Die Markierung angemeldeter Bearbeiter geschieht durch eine Umrandung des entsprechenden graphischen Knotens. Die Zuordnung der Aufgaben zu ihren Bearbeitern (Referenzwert von `Aufgabe.Verantwortlich`) wird per Standard-

layoutregel durch eine gerichtete Kante dargestellt. Da jedoch in der graphischen Darstellung durch die gerichteten Kanten vorrangig der Kontrollfluss visualisiert werden soll, wird eine neue Layoutregel verwendet, durch die die graphische Kante für `Aufgabe.Verantwortlich` ungerichtet gezeichnet wird. D.h. am Ende der Kante wird kein Pfeil, sondern ein Punkt (dot) gezeichnet.

```
edge presentation VerantwortlichEdgePresentation {
  requires automaton Selectable;
  not_selected:
    (CurveFigure,ClickInteractor,DotDecorator,DotDecorator)
  selected:
    (CurveControl(CurveFigure),
     TextCurveInteractor,DotDecorator,DotDecorator)
}
```

**Ausführung**

In Abbildung 6-8 und Abbildung 6-9 wird eine mögliche Beispielausführung eines EMU-Systems vom Typ *AHW* skizziert. In Abbildung 6-8, Zustand (1) wurde bereits vom Administrator des Workflows eine Ablaufspezifikation definiert. Sie besteht zunächst aus den drei Phasen „Proposal“, „Discussion“, „Submission“, aus den drei Aufgaben „propose“, „refuse“, „accept“ und aus den beiden Bearbeitern „Brandl“ und „Eickel“. Aufgrund der Graufärbung des entsprechenden graphischen Knotens erkennt man, dass nur die Phase „propose“ markiert ist. Für die Aufgabe „propose“ ist der Bearbeiter „Brandl“, für die Aufgaben „refuse“ und „accept“ der Bearbeiter „Eickel“ verantwortlich. Auf der „höheren“ Sprach- bzw. Ausführungsebene betrachtet, befindet sich in (1) das EMU-System, bzw. genauer, das Aktivitätsnetz an der Wurzel, in der Phase `Bearbeitungsphase`. Dies erkennt man am skizzierten Formular des Administrators, welches lediglich aus der Aktion `definieren` besteht (diese Aktion dient zur Einleitung der Phase `Definitionsphase`, vgl. oben das Benutzermodell). Ferner erkennt man in Zustand (1), dass der Bearbeiter „Brandl“ die Aufgabe „propose“ in seinem Formular bearbeitet. Genauer betrachtet, handelt es sich dabei um einen Knoten von `ZugewieseneAufgabe`, weil es sich bei dem im Formular dargestellten Datum um ein Element der Arbeitsliste von „Brandl“ handelt. Die Referenzwerte der Komponenten `ZVorphase` und `ZNachphase` werden durch graue Kanten im Grapheditor visualisiert. Dadurch kann der Administrator erkennen, welche Bearbeiter zum Erreichen bestimmter Phasen notwendig sind. In Zustand (1) ist „Brandl“ zum Erreichen der Phase „Discussion“ notwendig. Daher führt ein Pfad grauer Kanten über „Brandl“ zu „Discussion“. Weiterhin erkennt der Administrator im Grapheditor, dass der Bearbeiter „Brandl“ am System angemeldet ist, weil der graphische Knoten umrandet dargestellt wird.

In der Aktionsfolge $AF_1$ schliesst der Bearbeiter „Brandl“ die ihm zugewiesene Aufgabe ab und der Bearbeiter „Eickel“ meldet sich am EMU-System an.

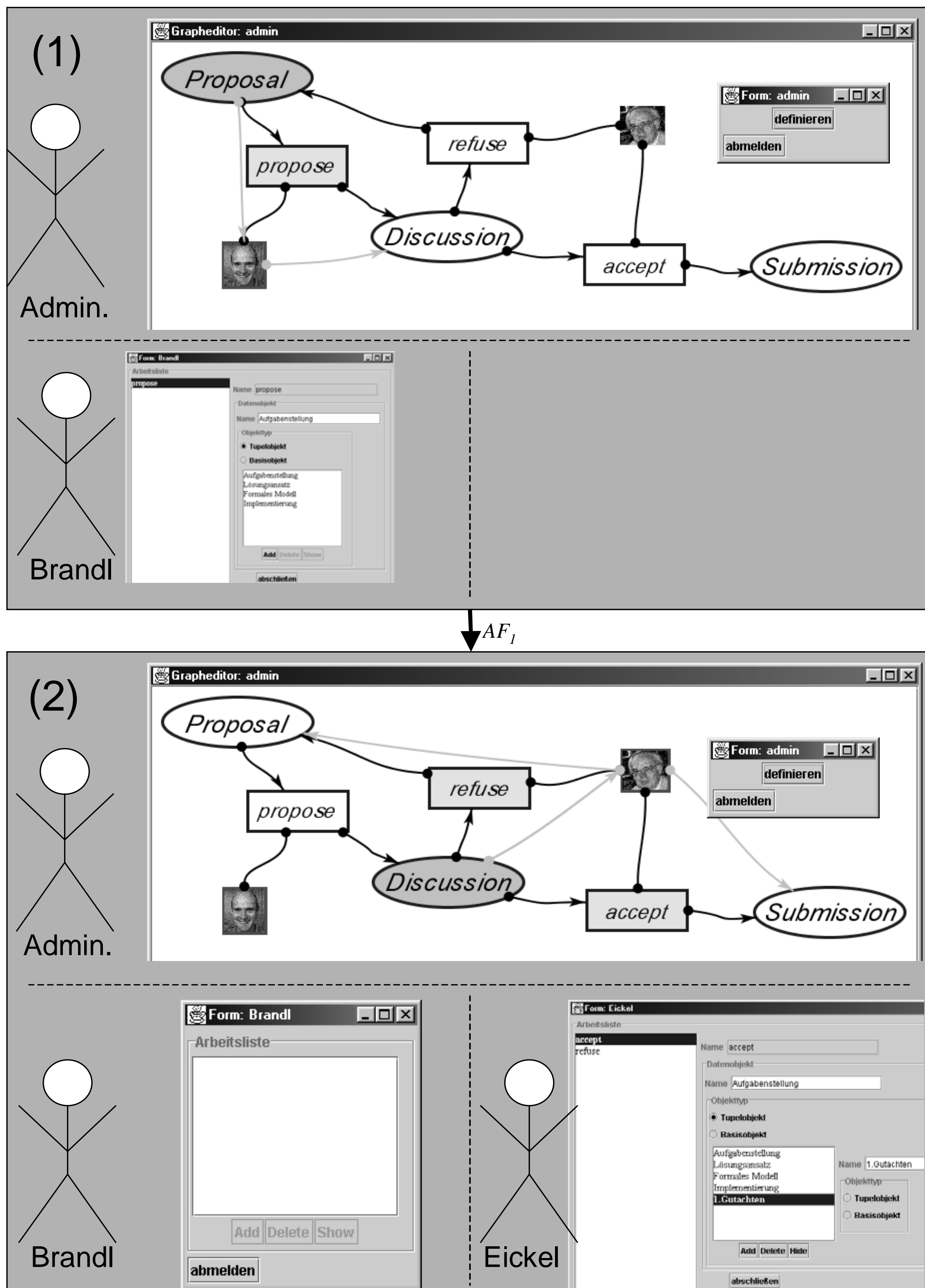


**Abbildung 6-8: Ausführung des Beispiels *Ad-hoc Workflows* (1)**

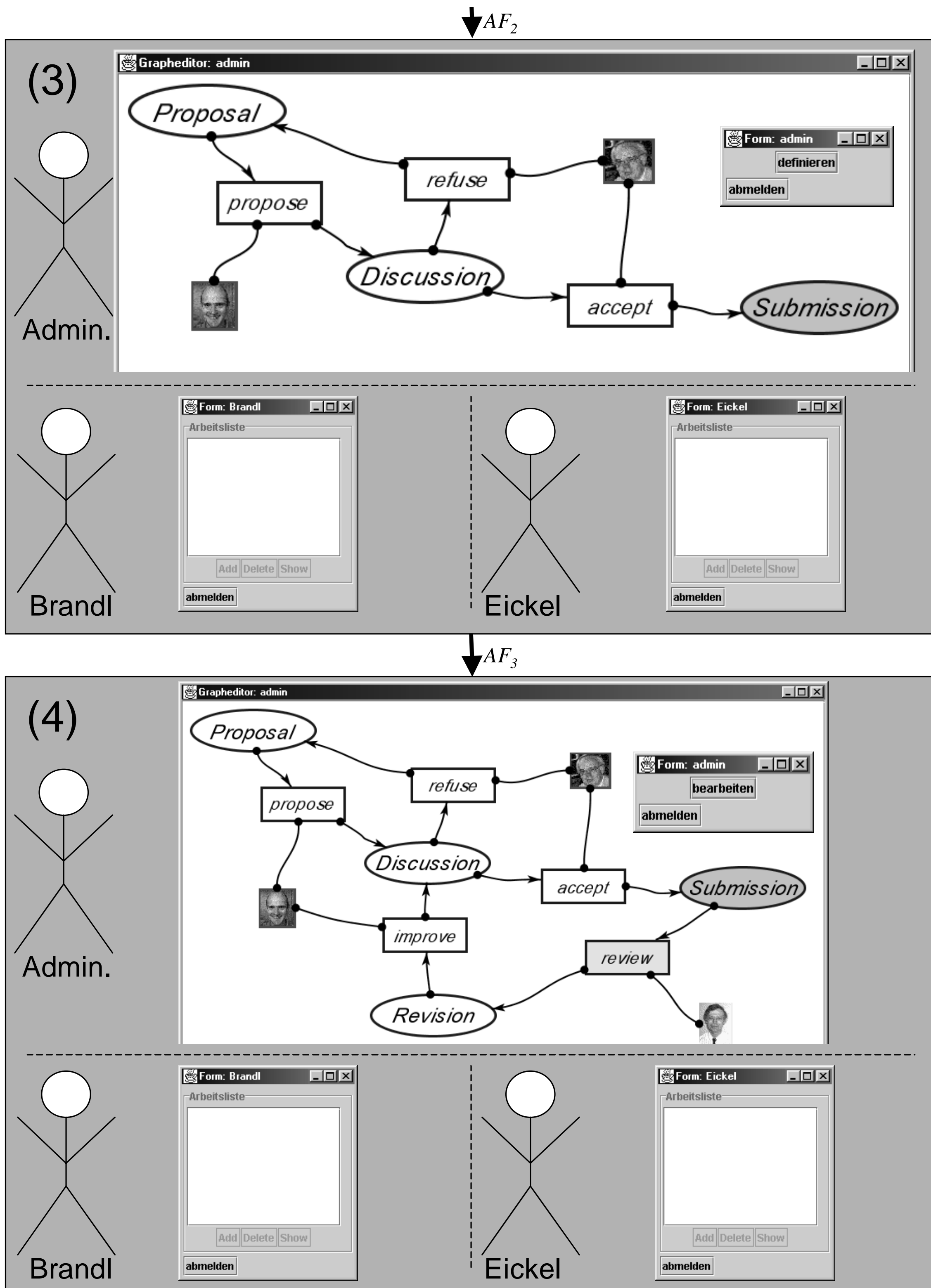


**Abbildung 6-9: Ausführung des Beispiels *Ad-hoc Workflows* (2)**

In Abbildung 6-8, Zustand (2) ist daher die Phase „Discussion“ markiert und der Bearbeiter „Eickel“ ist nun ebenfalls am EMU-System angemeldet. In der Arbeitsliste von „Eickel“ befinden sich – entsprechend seiner Verantwortlichkeit – zwei zugewiesene Aufgaben. In der rechten Seite des Formulars von „Eickel“ wird das zugehörige Datenobjekt dargestellt, das zuvor in Zustand (1) eingegeben wurde und nun erweitert wird. Die Arbeitsliste von „Brandl“ ist nun leer, weil keine Aufgabe aktiv ist, für die dieser Bearbeiter verantwortlich wäre. Wenn der Bearbeiter „Eickel“ die Aufgabe „accept“ abschließt (Aktionsfolge $AF_2$), so wird der in Abbildung 6-9 gezeigte Zustand (3) eingenommen. Hier sind die Arbeitslisten beider Bearbeiter leer, weil keine Aufgabe mehr aktiv ist. Jedoch kann der Administrator des Workflows die Definitionsphase durch Ausführen der Aktion `definieren` einleiten und anschließend die Ablaufspezifikation ändern.

In Abbildung 6-9, Zustand (4) wird eine mögliche Änderung der Ablaufspezifikation dargestellt. Die Änderung erfolgt innerhalb der Aktionsfolge $AF_3$. Es werden eine neue Phase „Revision“, zwei neue Aufgaben „review“ und „improve“ und ein neuer Bearbeiter eingefügt. Man beachte, dass die, während der vorherigen Ausführung markierte, Phase „Submission“ immer noch markiert ist, obwohl nun die Ablaufspezifikation bereits verändert wurde. Der Ausführungszustand bleibt also erhalten.

# 7 Verwandte Arbeiten

In diesem Kapitel wird EMUGEN mit anderen CADUI-Werkzeugen verglichen. Dabei sind vorrangig die folgenden Fragen von Bedeutung:

- Was ist der Zweck und das Ziel des CADUI-Werkzeugs? Handelt es sich vorrangig um ein Generierungskonzept (d.h. um kein vollständig implementiertes System), um einen Designassistenten (d.h. es geht vorrangig um die Verwaltung von Modellen und einer Unterstützung bei Entwurfsentscheidungen) oder um einen Generator, der ablauffähige interaktive Systeme bzw. Prototypen generiert?
- Welche Modellierungssprachen und Teilkomponenten werden verwendet? Inwiefern liegt dadurch eine Einschränkung bzw. Erweiterung im Vergleich zu EMUGEN vor?
- Wie beziehen sich die verschiedenen Modelle aufeinander und wie werden sie in der Laufzeitarchitektur integriert?
- Werden durch das CADUI-Werkzeug die für interaktive Informationssysteme im Sinne dieser Arbeit notwendigen dynamischen Formulare und Grapheditoren erstellt?

Obwohl sich alle hier betrachteten CADUI-Werkzeuge auf den ersten Blick ähneln (z.B. wird häufig ein Daten- und Aufgabenmodell wie bei EMUGEN verwendet), unterscheiden sie sich nicht nur in der Beantwortung obiger Fragen, sondern auch in der Art und Weise, wie sie in der Literatur beschrieben werden, grundlegend. Dies liegt auch daran, dass das Forschungsgebiet CADUI noch recht jung ist und seine Autoren einen teilweise stark unterschiedlichen Hintergrund aufweisen. Autoren mit Schwerpunkt Datenbankbereich definieren ihre Werkzeuge [BGK99] anders als solche mit Schwerpunkt Übersetzerbau [Sch97] und vor allem anders als Kognitionswissenschaftler [JMJ92]. So konzentrieren sich einige der Werkzeugbeschreibungen (z.B. [BGK99]) auf die Darstellung der teilweise sehr komplexen Funktionsweise der Benutzungsoberfläche (MB-UIDE) des

Werkzeugs selbst und beschreiben die einzelnen Eingabemodelle als Teile dieser Benutzungsoberfläche und nicht etwa in ihrer abstrakten Syntax (vgl. Anhang).

Daher stellt sich die Frage, ob man die verwandten CADUI-Werkzeugs wie in der jeweiligen Veröffentlichung beschreiben oder eine eigene, einheitliche Darstellungsform für alle Werkzeuge wählen soll. Eine einheitliche Präsentation birgt zwar die Gefahr der Verfälschung, insbesondere, weil die meisten der Werkzeuge nur unvollständig oder exemplarisch beschrieben sind, ist jedoch für einen Vergleich besser geeignet. Da es hier vor allem um eine Übersicht über die Gemeinsamkeiten und Unterschiede zu EMUGEN und weniger um Details geht, wird eine eigene, einheitliche Präsentation verwendet.

## 7.1 FUSE

FUSE (Formal User Interface Specification Environment) [LS96, Sch97] wurde am Lehrstuhl Eickel an der Technischen Universität München von 1991 bis 1996 entwickelt. Es handelt sich um ein sehr komplexes Generierungskonzept, welches weitgehend alle Aspekte interaktiver Systeme umfasst, jedoch an nicht allen Stellen vollständig ausgearbeitet wurde. Abbildung 7-1 zeigt die Gesamtarchitektur von FUSE [LS96] und wie dabei einige der in [VP99] angesprochenen Modelle (vgl. Abschnitt 1.2) zum Einsatz kommen.

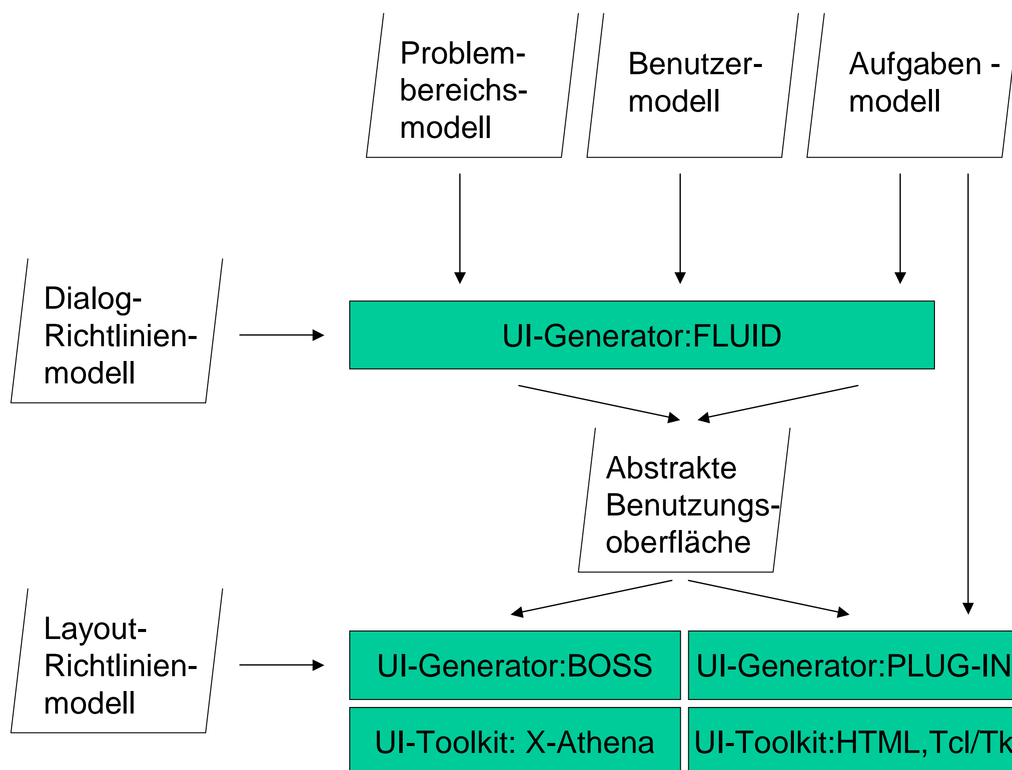


**Abbildung 7-1: Das FUSE-Generierungskonzept**

Zunächst werden von Applikationsexperten das Problembereichs-, Benutzer- und Aufgabenmodell erstellt. Der Zweck des Problembereichsmodells entspricht in etwa dem Datenmodell von EMUGEN. Der Generator FLUID [Bau96] erzeugt unter Berücksichtigung des Dialogrichtlinienmodells die formale Spezifikation einer abstrakten Benutzungsoberfläche (vgl. Abschnitt 2.2.1). Dabei ist als Problembereichsmodell eine alge-

braische Spezifikation vorgesehen. Aus dem Modell der abstrakten Benutzungsoberfläche, einem implementierungsnäheren Modell, erzeugt das BOSS-System [Sch97] unter Berücksichtigung des Layoutrichtlinienmodells eine ablauffähige Benutzungsoberfläche bei Benutzung der Interaktionsobjekte des UI-Toolkits X-Athena [NO90].

Das BOSS-System [Sch97] ist die einzige vollständig konzipierte, formal beschriebene und implementierte Teilkomponente von FUSE. Das BOSS-System bietet die folgenden Möglichkeiten:

- Spezifikation von Layoutrichtlinien
- Spezifikation von abstrakten Benutzungsoberflächen (vgl. Abschnitt 2.2.1)
- Automatische, intelligente Generierung von Benutzungsoberflächen in ihrer Layoutstruktur

Der Begriff *intelligente Generierung* ist durchaus angebracht, weil das BOSS-System durch Anwendung von Techniken aus dem Übersetzerbau (z. B. Grammatikflussanalyse [WM97]) Eigenschaften der abstrakten Benutzungsoberflächen berechnet (bzw. abschätzt, wenn sie nicht berechenbar sind), um daraus, unter Berücksichtigung der Layoutrichtlinien, das entsprechende Layout zu konstruieren. Genauer: es werden statisch (d.h. zur Generierungszeit) Layoutberechnungsregeln definiert, die festlegen, wie zur Laufzeit das im Allgemeinen dynamische Layout der abstrakten Benutzungsoberfläche berechnet wird.

Als einheitliche Spezifikationssprache verwendet das BOSS-Systeme HIT-Bausteine, die als Erweiterungen dynamischer, attributierter Grammatiken [Gan78] betrachtet werden können. Die Laufzeitmodelle des BOSS-Systems sind die bereits in Abschnitt 2.2.3 diskutierten HIT-Instanzen. Es wäre aus konzeptioneller Sicht möglich gewesen, BOSS als Generator für dynamische Formulare zu verwenden. Dies wurde ausschließlich aus implementierungstechnischen Gründen (Zielsprache von EMUGEN: Java) nicht gemacht und anstelle dessen mit FORMGEN [BK99] eine vereinfachte Form des BOSS-Systems verwendet.

Parallel zu BOSS erfolgt in FUSE durch PLUG-IN [LS96] die automatische Erstellung eines Laufzeit-Hilfesystems für das zu generierende interaktive System. Dieses Hilfesystem besteht aus einer Menge, dynamisch zur Laufzeit generierter, HTML-Seiten mit Informationen zur Ausführung des Systems (unter Berücksichtigung des aktuellen Ausführungszustands) und einem Zustandsübergangsdiagramm (basierend auf dem UI-Toolkit Tcl/Tk [Ous95]), welches die Systemausführung visualisiert.

Alternativ zur Darstellung in Abbildung 7-1 ermöglicht die FUSE-Komponente FIRE (Formal Interface Requirements Engineering, [Sch95]) die direkte Ausführung des Problembereichs- und Aufgabenmodells, also ohne Berücksichtigung von Dialog- und Layoutrichtlinien. Bei der Verwendung von FIRE besteht das Problembereichsmodell aus einer Menge von Funktionsdeklarationen. Die Verwendung zweier Notationen für das Problembereichsmodell macht deutlich, dass FUSE nicht als vollständig implementiertes System mit festgelegten Spezifikationssprachen und Generatoren verstanden werden soll, sondern als umfassendes Generierungskonzept.

Das FIRE-System benutzt wie viele CADUI-Werkzeuge beim Aufgabenmodell eine rekursive hierarchische Strukturierungsmöglichkeit (vgl. Diskussion in Abschnitt 4.4.1). Innere Knoten sind dabei zusammengesetzte Aufgaben, die aus Teilaufgaben bestehen,

die je nach Knotentyp sequentiell oder parallel auszuführen sind, während die Blätter (atomare, terminale Aufgaben) Funktionsaufrufe des Problembereichs darstellen. Die Zuordnung von Aufgaben zum Problembereich erfolgt also über die Blätter des Aufgabenmodells. Auch dies erfolgt in gleicher Weise bei anderen CADUI-Werkzeugen. Als Folge ergibt sich zur Laufzeit die interaktive Bearbeitung einer dynamischen Instanz des Aufgabenmodells (Aufgabenbaum).

### *Gemeinsamkeiten mit EMUGEN*

Da EMUGEN am gleichen Lehrstuhl wie FUSE entstanden ist, ergeben sich natürlich eine Reihe von Gemeinsamkeiten bzw. ähnliche grundlegende Ansätze. Die in Abbildung 7-1 gezeigte Architektur war Vorbild für das EMU-Generierungskonzept (vgl. Abbildung 1-2). Ähnlich wie EMUGEN generiert das FIRE-System ablauffähige Prototypen aus dem Daten- und Aufgabenmodell. Der Zusammenhang von (erweiterten) HIT-Instanzen mit der Modellierung und Implementierung der EMU-Systeme wurde bereits in Abschnitt 2.2.3 beschrieben. Ähnlich wie BOSS und PLUG-IN verschiedene Benutzungsoberflächen für eine Applikation erstellen (die eigentliche „Arbeits-Benutzungsoberfläche" bzw. das zugehörige Hilfesystem) verwaltet ein EMU-System verschiedene Benutzungsoberflächen für die einzelnen Benutzer (vgl. die Zugriffsadapter in Abschnitt 3.2.2). Dasselbe Konzept wurde also beim EMU-Generierungskonzept auf einer anderen Abstraktionsebene verwendet als bei FUSE.

### *Unterschiede zu EMUGEN*

Während FUSE als Generierungskonzept zu verstehen ist, stellt EMUGEN ein implementiertes, batch-orientiertes CADUI-Werkzeug dar, das die Erstellung ablauffähiger Prototypen unter Berücksichtigung der Eingabemodelle ermöglicht. Dazu gehört auch die vollständige Entwicklung und Implementierung von Daten-, Aufgaben- und Benutzermodellierungssprachen. Insbesondere die Verwendung eines konkreten Benutzermodells zur Beschreibung einer Menge von Benutzern und die Generierung von Mehrbenutzeranwendungen geht über die bei FUSE vorgesehenen Möglichkeiten hinaus. Die Verwendung eines Dialog-Generators wie FLUID wurde beim EMU-Generierungskonzept nicht integriert. Jedoch schlägt auch FUSE noch kein konkretes Konzept zur Formalisierung von Dialog-Richtlinien vor.

Als Erweiterung zum FIRE-System integriert EMUGEN das Benutzermodell in die Generierung und erstellt prototypische Mehrbenutzer-Systeme.

Beim Vergleich der Laufzeitarchitektur von Prototypen, die mit dem FIRE-System generiert wurden und EMU-Systemen fällt ein grundsätzlicher Unterschied auf. Während ein FIRE-Prototyp zur Laufzeit eine dynamische Instanz des Aufgabenmodells („Aufgabenbaum") verwaltet, besteht ein EMU-System (also die EMU-Laufzeitkomponente) aus einer dynamischen Instanz des Datenmodells (der Mehrbenutzer-Applikationsschnittstellenkern MAK). Diese Laufzeitarchitektur wurde bei EMU gewählt, weil wir Datenstrukturen als das allgemeinere Konzept als Aufgaben(instanzen bzw. bäume) betrachten (vgl. Abschnitte 4.4 und 5.4).

D.h., wenn man eine Aufgabeninstanz zur Laufzeit verwalten wollte, so kann man mit EMU (vgl. etwa das Beispiel *Ad-hoc Workflows*) Aufgaben als Datenstrukturen modellieren und erhält somit, wie beim FIRE-System, zur Laufzeit eine Aufgabeninstanz.

Die Generierung graphischer Grapheditoren ist bei FUSE nicht vorgesehen, jedoch wird mit Hilfe der Teilkomponente BOSS die Generierung dynamischer Formulare ermöglicht.

## 7.2 JANUS

Das JANUS-System [BHK96] wurde zunächst an der Ruhr-Universität Bochum entwickelt und wird mittlerweile kommerziell vertrieben. Abbildung 7-2 zeigt seine grundsätzliche Architektur.

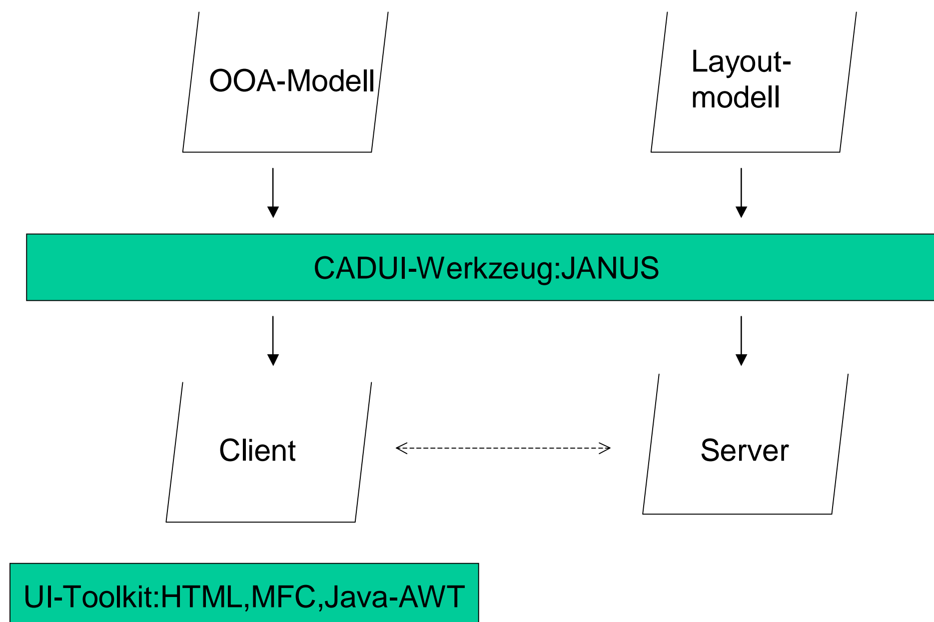


**Abbildung 7-2: Das JANUS-System**

Das JANUS-System erhält als Eingabe ein OOA-Modell [RBP91], welches in der kommerziellen Version als UML-Klassendiagramm [OMG01] graphisch modelliert werden kann. Für die Klassen des OOA-Modells können Attribute und Regeln angegeben werden, die beschreiben, wie ihre Objekte in den generierten, formular-orientierten Benutzungsoberflächen dargestellt werden. In Abbildung 7-2 sind diese Attribute und Regeln als Layoutmodell eingezeichnet. Es kann ein Standardlayoutmodell verwendet werden, welches in der Benutzungsoberfläche (MB-UIDE) von JANUS entsprechend den applikationsspezifischen Anforderungen angepasst werden kann.

Das JANUS-System erstellt ein interaktives Informationssystem, das aus beliebig vielen Clients und einem Server besteht, also nicht nur die eigentliche Benutzungsoberfläche. Dabei kann der Entwickler angeben, auf welchen UI-Toolkits (vgl. Abschnitt 2.3.2) die einzelnen Clients basieren sollen. Die Kommunikation zwischen Client und Server erfolgt durch Interprozesskommunikation oder, wenn eine Internetanwendung erstellt werden soll, über das TCP/IP Protokoll.

### *Gemeinsamkeiten mit EMUGEN*

Eine wichtige Gemeinsamkeit von JANUS und EMUGEN, die ansonsten von keinem der weiteren CADUI-Werkzeuge geteilt wird, ist die Generierung eines funktionalen Anwendungskerns der durch die ebenfalls automatisch generierten Benutzungsoberflächen

im Sinne einer Modellkomponente der MVC-Architektur (vgl. Abschnitt 2.2.2) zu bearbeiten ist.

Dieses Vorgehen ist die Voraussetzung für eine iterative Entwicklung, die sowohl von JANUS als auch von EMUGEN unterstützt wird. Dazu ist es nämlich notwendig, bereits aus den Anfangsüberlegungen zum Fachkonzept – die bei JANUS in ein erstes OOA-Modell abgebildet werden – lauffähige Prototypen zu generieren und mit den Kunden zu diskutieren. Damit erhält man von Anfang an eine starke Benutzerpartizipation. Lauffähige Prototypen ([BHK96] sprechen von *Pilotsystemen*, weil die Anwender damit bereits produktiv arbeiten können) die nicht nur das Layout der Benutzungsoberfläche, sondern auch ihre grundsätzliche Funktionsweise zeigen, benötigen bereits einen Anwendungskern, bzw. einen Prototyp des Anwendungskerns.

Ein OOA-Modell ist aus abstrakter Sicht dem Datentypkonzept von EMUGEN durchaus ähnlich (beide sind – etwa im Gegensatz zum Relationenmodell [Cod70] – objekt- und nicht mengenorientiert), obwohl es im Hinblick auf den Polymorphismus der zu bearbeitenden Daten durchaus wichtige Unterschiede im Detail gibt.

### *Unterschiede zu EMUGEN*

Diese Unterschiede beziehen sich beispielsweise auf die Verwendung von Varianten beim EMU-Datenmodell anstelle des Vererbungskonzepts im objektorientierten Sinne bei JANUS (im Detail besprochen in Abschnitt 4.4.4). Das JANUS-System umgeht das Problem der Bearbeitung polymorpher Datenstrukturen, wofür dynamische Formulare notwendig wären, durch eine entsprechende Einschränkung bei der Bearbeitung der Objekte. So hat der Benutzer beim Neuanlegen den dynamischen Typ eines Objekts explizit anzugeben und kann ihn später nicht mehr verändern. Aufgrund dieser Einschränkung genügt es auch, das Layout der Formulare statisch zu berechnen.

Bei der Layoutberechnung verwendet jedoch das JANUS-System eine wesentlich komplexere Wissensbasis [Kru99] in Bezug auf die ergonomische Gestaltung der Formularteile als EMUGEN.

Die technische Ausführung und Ausprogrammierung weiterer praxisrelevanter Details (z.B. kann der Entwickler bei JANUS verschiedene Interaktionsobjekte zur Bearbeitung von Referenzen bestimmen) ist bei JANUS wesentlich reichhaltiger und damit vollständiger als bei EMUGEN. Dazu sei angemerkt, dass es sich bei JANUS bereits um ein über Jahre ausgereiftes, kommerziell verfügbares Werkzeug handelt, während EMUGEN nur einen Forschungsprototyp darstellt. Daher verfügt JANUS auch über eine sehr bequeme Benutzungsoberfläche, bei der das OOA-Modell durch ein UML-Klassendiagramm definiert werden kann. Ferner generiert JANUS aus einem OOA-Modell zur Verwaltung des Anwendungskern einen eigenen Server und für die Benutzungsoberflächen HTML-, MFC- und Java-Clients, während die vom aktuellen Prototyp generierten EMU-Systeme als einzelne Prozesse ablaufen.

Von der grundsätzlichen Funktionalität verfügt jedoch EMUGEN über eine Reihe von Möglichkeiten, die durch das JANUS-System nicht unterstützt werden. So können durch JANUS nicht die Aspekte spezifiziert werden, die durch das Aufgaben- und Benutzermodell von EMU zu formulieren sind. Daher können auch keine Benutzergruppen mit verschiedenen Zugriffsrechten beschrieben und in die Generierung integriert werden. Es fehlt auch ein Konzept zur Spezifikation von Vor- und Nachbedingungen zu Aktionen.

Interessanterweise entstand im Umfeld von JANUS ein zu GRACE [Kle99] vergleichbares Werkzeug [Gil99] zur Generierung von Diagramm-Editoren aus OOA-Modellen. Jedoch wurde nicht wie im EMU-Generierungkonzept versucht, sowohl Diagramm-Editoren als auch formular-orientierte Benutzungsoberflächen zur Bearbeitung eines (also desselben) Anwendungskerns im Sinne einer MVC-Architektur zu integrieren. Eine sinnvolle Anwendung dieser Integration bei EMUGEN wurde durch das Beispiel *Ad-hoc Workflows* gezeigt.

## 7.3 TEALLACH

TEALLACH (gälischer Name für Schmiedefeuer) [BGM99] ist ein von den Universitäten von Edinburgh, Glasgow und Manchester gemeinsam entwickeltes CADUI-Werkzeug, welches die Erstellung von Benutzungsoberflächen für objektorientierte Datenbanken als vorrangiges Ziel verfolgt. Abbildung 7-3 zeigt seine Gesamtarchitektur.

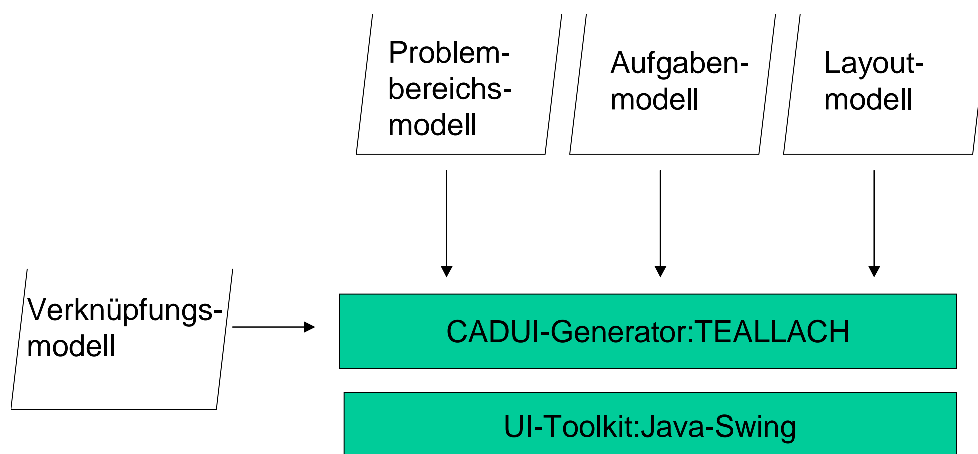


**Abbildung 7-3: Das TEALLACH-System**

Wie Abbildung 7-3 skizziert, werden bis auf das Benutzermodell (das zu Beginn des TEALLACH-Projekts noch berücksichtigt wurde) ähnliche Eingabemodelle wie bei EMUGEN verwendet. Jedoch ergeben sich eine Reihe von Unterschieden in Bezug auf die einzelnen Modellierungssprachen und den Bezügen zwischen den Modellen.

Das Problembereichsmodell von TEALLACH besteht aus einer Menge von Java-Klassen, die optional aus einer objektorientierten Datenbank-Schemadefinition erstellt wurden. Weitere Java-Klassen, deren Objekte etwa zur Verwaltung temporärer Daten der Applikation (z.B. Ergebnisse von Datenbankabfragen) dienen, können beliebig hinzugefügt werden.

Beim hierarchischen Aufgabenmodell von TEALLACH kann einem inneren Knoten, d.h. einer zusammengesetzten Aufgabe, die aus Teilaufgaben besteht, ein Ausführungsoperator zugewiesen werden. Beispielsweise kann solch ein Ausführungsoperator festlegen, dass die Teilaufgaben sequentiell, optional oder quasiparallel (Interleaving-Modus) auszuführen sind. Dieser Aufgabenmodellierungsansatz, bei dem den Aufgabenknoten ein Ausführungsoperator und damit eine feste Ausführungssemantik zugeordnet wird, wird

im Bereich CADUI beispielsweise auch im ADEPT-System [WJK93], im oben genannten FIRE-System und in der ConcurTaskTree-Notation [Pat99] angewandt. Ein ähnlicher Ansatz findet sich auch in der ISO-OSI Spezifikationssprache für nebenläufige Prozesse LOTOS [LOT88] wieder. Im unten diskutierten MASTERMIND-System findet man dieses Konzept bei der Dialogmodellierung, was die enge Verwandtschaft bzw. die nicht ganz klar abzutrennende Funktion des Aufgaben- und Dialogmodells zeigt.

Jeder atomaren Aufgabe (Blattknoten im Aufgabenbaum) werden beim TEALLACH-System Instanzmethoden der Klassen des Problembereichsmodells zugeordnet. Dazu werden den jeweils (einer atomaren Aufgabe) übergeordneten inneren Knoten des Aufgabenbaums (zusammengesetzte Aufgabe) sogenannte Zustandsobjekte der Klassen des Problembereichsmodells zugeordnet. Angenommen, einer zusammengesetzten Aufgabe $k$ des Aufgabenbaums ist das Zustandsobjekt $t$ von der Klasse $T$ zugeordnet, so kann einer atomaren Teilaufgabe $a$ von $k$ jede Instanzmethode $m$ von $T$ zugeordnet werden.

Das Layoutmodell von TEALLACH erlaubt die Beschreibung von abstrakten und konkreten Interaktionsobjekten, ein häufig bei Benutzungsoberflächenwerkzeugen auf unterschiedlichen logischen Ebenen angewandtes Konzept. Beispielsweise verwenden plattformunabhängige UI-Toolkits wie Java-AWT den Begriff *abstraktes Interaktionsobjekt* um auszudrücken, dass das „tatsächlich" am Bildschirm platzierte Interaktionsobjekt (Component, Widget, Fenster, etc.) erst zur Laufzeit auf ein plattformabhängiges Interaktionsobjekt abgebildet wird. Auf höherer logischer Ebene legen CADUI-Werkzeuge wie TRIDENT [VB93] oder eben TEALLACH abstrakte Interaktionsobjekte entsprechend ihrer abstrakten Funktionalität fest. TEALLACH verwendet dazu etwa Bezeichnungen wie *Display* (Interaktionsobjekt zur Darstellung von Daten), *Editor* (zum Bearbeiten von Daten) und *ActionItem* (zur Ausführung von Aktionen).

Das Verknüpfungsmodell von TEALLACH verwaltet die Bezüge zwischen den verschiedenen anderen Modellen einer Applikation. Diese Bezüge können in der MB-UIDE von TEALLACH auf sehr bequeme Art und Weise vorgenommen werden („drag & link"). Eine der möglichen Verknüpfungen wurde bereits angesprochen, nämlich die Verknüpfung des Aufgabenmodells (atomare Aufgabe) mit dem Problembereichsmodell (Instanzmethode). Weiterhin kann beispielsweise ein abstraktes Interaktionsobjekt $p$ mit einer atomaren Aufgabe $a$ verknüpft werden, um anzuzeigen, dass zur interaktiven Bearbeitung von $a$ das Interaktionsobjekt $p$ zu verwenden ist.

### *Gemeinsamkeiten mit EMUGEN*

Neben der ähnlichen Strukturierung der Eingabemodelle verwendet auch TEALLACH ein objektorientiertes Problembereichsmodell (entspricht dem EMU-Datenmodell). Entsprechend könnten die Gemeinsamkeiten von EMUGEN mit JANUS hier erneut angeführt werden.

### *Unterschiede zu EMUGEN*

Im Gegensatz zu EMUGEN und JANUS erstellt TEALLACH keinen Anwendungskern, auf den mehrere Benutzungsoberflächen und damit mehrere Benutzer zugreifen können. Da kein Benutzermodell verwendet wird, können auch keine Mehrbenutzersysteme spezifiziert und damit auch nicht generiert werden. Mit TEALLACH können auch keine Diagrammeditoren spezifiziert oder generiert werden.

## 7.4 MASTERMIND

MASTERMIND (Models Allowing Shared Tools and Explicit Representations to Make Interfaces Natural to Develop) [BDR97, SR00] entstand als gemeinsames Projekt des Georgia Institute of Technology und der University of Southern California.

Sowohl die Strukturierung der Eingabemodelle als auch die Laufzeitarchitektur der von MASTERMIND generierten interaktiven Systeme entsprechen einer direkten Umsetzung des Seeheim-Modells (Abbildung 7-4, vgl. Abschnitt 2.2.1, [Pfa85]).

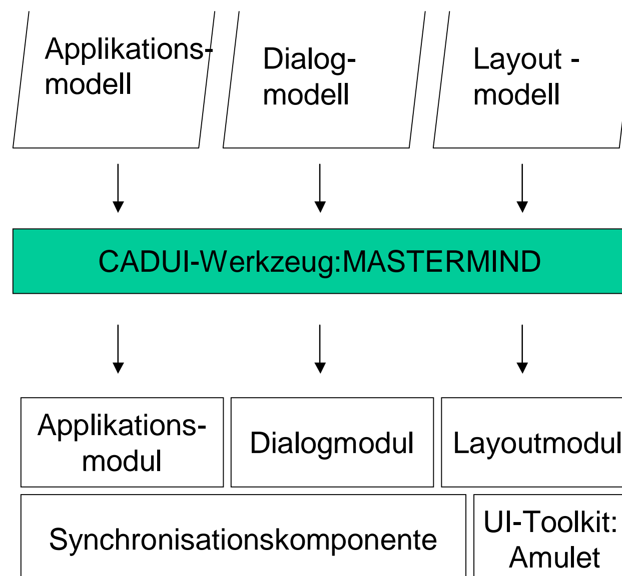


**Abbildung 7-4: Das MASTERMIND-System [SR98]**

Zur synchronen Kommunikation der Präsentations-, Dialog- und Applikationskomponenten dient eine Synchronisationskomponente. Dieses Vorgehen ist an die *PAC*-Architektur (*Presentation, Abstraction, Controller*) [BC91] angelehnt, wobei die Synchronisationskomponente der Controller-Komponente von PAC entspricht.

Die Applikationsmodellierungssprache von MASTERMIND ist IDL (Interface Definition Language) [Vos00], während für das Dialog- und das Präsentationsmodell die an LOTOS [BB87] angelehnte Sprache MDL [Sti99] (Mastermind Dialog Language) verwendet wird. Durch das Präsentationsmodell wird nicht das Aussehen der Benutzungsoberfläche spezifiziert, sondern das abstrakte Verhalten des Präsentationsmoduls. Dieses besteht etwa darin, Aktionen auszulösen, welche für das Dialogmodul von Bedeutung sind.

### *Gemeinsamkeiten mit EMUGEN*

Die Strukturierung der Eingabemodelle von MASTERMIND, TEALLACH und EMUGEN ähneln sich. Das Dialogmodell von MASTERMIND wird in [Sti99] auch als *Benutzeraufgabenmodell* bezeichnet und verfügt über ähnliche Sprachelemente, die etwa in TEALLACH und FIRE zur Aufgabenmodellierung verwendet werden. Zugehörig zum MASTERMIND-Projekt wurde mit ADAPTIVE FORMS [FS98] ein Generator für dynamische Formulare entwickelt, der dem bei EMUGEN verwendeten FORMGEN ent-

spricht. Dabei entspricht die Eingabesprache von ADAPTIVE FORMS der Datenmodellierungssprache von EMUGEN (wobei Referenzen und Rekursion fehlen). Jedoch werden die von ADAPTIVE FORMS generierten Formulare nicht wie bei EMUGEN in ein von MASTERMIND generiertes Gesamtsystem integriert.

### *Unterschiede zu EMUGEN*

Im Gegensatz zum EMU-Generierungskonzept betrachtet MASTERMIND die Eingabemodelle als formale Beschreibung autonomer Teilkomponenten, die über Ereignisse miteinander kommunizieren. Demgegenüber stellt die Laufzeitarchitektur von EMUGEN einen dynamischen, vorrangig durch das EMU-Datenmodell definierten, hierarchisch organisierten Graph dar, der durch Benutzerinteraktionen seinen Zustand bzw. seine Struktur ändert. D.h. die Aufteilung der EMU-Eingabemodellen findet sich – anders als bei MASTERMIND – nicht in der Laufzeitarchitektur der generierten EMU-Systeme wieder.

Bei der Benutzung von MASTERMIND und EMUGEN fällt ein weiterer Unterschied zwischen den beiden Werkzeugen auf. Die Modelle von MASTERMIND sind zwar deklarativ in dem Sinne, dass keine Programmierung in einer prozeduralen Sprache erforderlich ist, aber es ist für den Entwickler notwendig, alle Interaktionen als LOTOS bzw. MDL-Aktionen explizit zu beschreiben. EMUGEN hingegen benutzt das Konzept der Metatypen (z.B. Tupel, Varianten, usw.) um die Interaktionen, die sich insbesondere bei der Datenverwaltung ergeben, implizit zu bestimmen.

Das EMU-Generierungskonzept kann im Gegensatz zu MASTERMIND nicht als rein deklarativer Ansatz betrachtet werden, weil innerhalb der Aktionen – ähnlich wie bei Parser- und Scannergeneratoren – benutzerdefinierte Kommandos in einer prozeduralen Programmiersprache (in der aktuellen Implementierung: Java) geschrieben werden.

MASTERMIND unterstützt zwar nicht die Generierung graphischer Editoren wie EMUGEN, jedoch erlaubt das verwendete Konzept die flexible Einbindung externer Präsentationskomponenten. Ferner erstellt MASTERMIND keine Mehrbenutzersysteme.

## 7.5 MOBI-D

MOBI-D (Model-Based Interface Designer) [Pue97] ist ein an der Universität Stanford entwickeltes CADUI-Werkzeug. Ziel von MOBI-D ist nicht die automatische Generierung eines ablauffähigen Gesamtsystems (bzw. eines Prototypen des Gesamtsystems), sondern vielmehr eine möglichst komfortable Unterstützung des interaktiven Entwicklungsprozesses. Dazu bilden, wie beim generativen Ansatz, die Modelle des Entwicklungsprozesses die zentralen Elemente, die von MOBI-D verwaltet werden. Zum Entwicklungsprozess im Sinne von MOBI-D gehört neben der Bearbeitung der Modelle auch die Definition von Beziehungen zwischen den Elementen der einzelnen Modellen. Eine solche Beziehung kann etwa beschreiben, dass zur Erledigung einer Aufgabe $a$ (also eines Elements des Aufgabenmodells) ein Problembereichelement vom Typ $D$ einzugeben ist. Abbildung 7-5 zeigt die Modelle und die Beziehungen, die zwischen den Modellelementen in der Entwicklungsumgebung von MOBI-D definiert werden können.

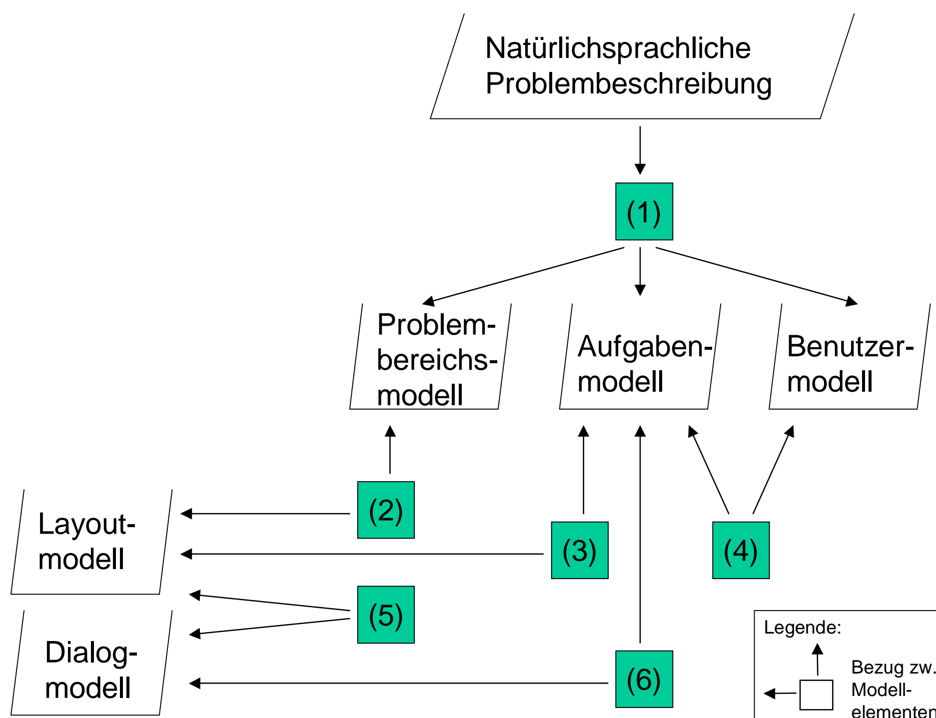


**Abbildung 7-5: Modelle und Beziehungen in MOBI-D**

Aus einer natürlichsprachlichen Problembereichs- bzw. Aufgabenbeschreibung können mit Hilfe der Teilkomponente U-TEL [TMP98] (Komponente (1) in Abbildung 7-5) teilautomatisiert Problembereichs- und Aufgabenelemente sowie verschiedene Benutzertypen identifiziert und extrahiert werden.

Das Problembereichsmodell von MOBI-D stellt eine Datentypbeschreibungssprache dar, wobei im Gegensatz zu EMUGEN keine direkte Umsetzung in geeignete Interaktionsobjekte zur Bearbeitung zugehöriger Datenelemente vorgesehen ist. Deshalb gibt es auch keine Metatypen wie Varianten oder Listen. Das Aufgabenmodell ermöglicht – wie bei FIRE, TEALLACH oder MASTERMIND – eine hierarchische Aufgabenbeschreibung unter Verwendung der bereits erwähnten Ausführungsoperatoren. Im Benutzermodell können – wie bei EMUGEN – Benutzertypen mit Attributen und Rollen von Benutzern definiert werden, jedoch werden nur Informationen bzgl. der Benutzertypen weiter verarbeitet.

Mit Hilfe des Teilwerkzeugs MOBILE [PCO99] kann der Entwickler interaktiv eine Zuordnung von Aufgaben zu Interaktionsobjekten sowie deren graphische Anordnung definieren (Komponente (3) in Abbildung 7-5). Dies geschieht unter Berücksichtigung der Zuordnung von Benutzertypen zu Aufgaben (Komponente (4)). Das Teilwerkzeug TIMM [PE99] unterstützt den Entwickler bei der Definition der Beziehungen zwischen dem Problembereichs- und dem Aufgabenmodell mit dem Layoutmodell (Komponente (2) und Komponente (6) in Abbildung 7-5). Hier wird beispielsweise angegeben, welche Interaktionsobjekte (Elemente des Layoutmodells) zur Bearbeitung der Datenelemente des Problembereichs verwendet werden sollen und in welcher Weise die Aufgabenbearbeitung dem Endbenutzer visualisiert wird.

### *Gemeinsamkeiten mit EMUGEN*

Der in EMUGEN implementierte Ansatz, verschiedene Benutzertypen in den Entwicklungsprozess zu integrieren, wurde bereits bei der MOBI-D-Teilkomponente MOBILE realisiert. Allerdings ist bei MOBILE keine Zugriffsdefinition bzgl. der Benutzereigenschaften (Rollen) wie bei EMUGEN vorgesehen. Es kann lediglich für jeden Benutzertyp ein spezielles Layout festgelegt werden. Durch Berücksichtigung von Benutzereigenschaften ist das Benutzerkonzept von EMUGEN flexibler und mächtiger.

### *Unterschiede zu EMUGEN*

Ein grundsätzlicher Unterschied zwischen den beiden Systemen besteht in der unterschiedlichen Zielsetzung (EMUGEN ist ein Generator, MOBI-D ein Modellverwalter und Designassistent). Daraus ergibt sich beispielsweise, dass bei EMUGEN keine Unterstützung bei der Transformation natürlichsprachlicher Aufgabenbeschreibungen in die formalen Modelle erfolgt und bei MOBI-D keine Prototypgenerierung des Gesamtsystems vorgesehen ist. Ein statisches Preview der modellierten Präsentation einer Benutzungsoberfläche wird jedoch durch die MOBI-D-Teilkomponente MOBILE ermöglicht.

Wie auch alle anderen bisher betrachteten CADUI-Werkzeuge, die ein Aufgabenmodell verwenden, erfolgt bei MOBI-D – anders als bei EMUGEN – keine Zuordnung von Aufgaben zu den Typen des Datenmodells, sondern umgekehrt eine Zuordnung von Datenelementen zu Aufgaben [PE99].

Ein weiterer Unterschied liegt in einer Modellierungsvereinfachung bei EMUGEN. Jedes der einzelnen Modelle bei MOBI-D beschreibt in gewisser Weise Hierarchien (z.B. hierarchische Daten und Aufgaben oder einen hierarchischen Fensterverbund im Layout). Bei EMUGEN wird nur genau ein hierarchisches Modell, nämlich das Datenmodell verwendet, woraus – sofern nichts anderes definiert wurde – sämtliche anderen Hierarchien (z.B. auch die Fensterhierarchie) abgeleitet werden. Da dies bei MOBI-D nicht so gemacht wird, erhält der Entwickler zwar mehr Freiheit, muss jedoch auch mehr Zusammenhänge explizit selbst definieren.

## 7.6 Zusammenfassung des Vergleichs

Alle im Bereich CADUI bisher bekannten Werkzeuge weisen einige grundlegende Unterschiede mit EMUGEN auf. Die wichtigste Eigenschaft, die EMUGEN dabei auszeichnet, ist die Möglichkeit der Spezifikation mehrerer Benutzer und die Integration des Mehrbenutzeraspekts in ein ablauffähiges System.

Ferner integriert kein verwandtes CADUI-Werkzeug generierte Grapheditoren in generierte interaktive Systeme. D.h. es gibt zwar neben dem von EMUGEN benutzten GRACE noch andere Generatoren für Grapheditoren (z.B. [Gil99], für eine weitere Übersicht siehe [Kle99]), jedoch sind diese noch nicht in ein Generierungskonzept unter Berücksichtigung von Aufgaben- und Benutzermodellen integriert worden.

Einige der Werkzeuge (z.B. JANUS, TEALLACH) verfügen im Gegensatz zu EMUGEN über eine komfortable Benutzungsoberfläche. Jedoch können mit EMUGEN selbst – wie die Beispiele in Kapitel 6 zeigen – entsprechende Benutzungsoberfläche (wenngleich bislang nur im prototypischen Rahmen) generiert werden.

# 8 Zusammenfassung und Ausblick

## 8.1 Ergebnisse

In der Arbeit wurde ein generativer Ansatz zur Entwicklung interaktiver Informationssysteme und ihrer Benutzungsoberflächen mit mehreren Benutzern gezeigt. Dazu wurde die Syntax und Semantik entsprechender Modellierungssprachen motiviert, entwickelt, formal beschrieben und prototypisch implementiert.

Zusätzlich zu bereits bestehenden CADUI-Werkzeugen ist es erstmals gelungen, Prototypen von Mehrbenutzeranwendungen zu generieren und dabei auch diagrammsprachliche Editoren zu integrieren.

Die Beschreibung einer Reihe verschiedenartiger Beispiele belegt die grundsätzliche Anwendbarkeit dieses Ansatzes und die Möglichkeiten des implementierten Generators, ausführbare Prototypen zu generieren.

Die Mächtigkeit des implementierten EMU-Generierungskonzepts zeigt sich insbesondere an der Tatsache, dass es sich nicht nur um Minimalbeispiele handelt, die spezifiziert und generiert wurden, sondern unter anderem auch um zwei neue Diagrammsprachen (Beispiele *TUM* und *Ad-hoc Workflows*). Dies war im Rahmen einer einzelnen Arbeit nur deshalb möglich, weil die Möglichkeiten, die der EMU-Generierungsansatz bietet, nämlich die generative und damit auch die iterative Entwicklung von Mehrbenutzeranwendungen und Diagrammsprachen, bereits bei der Erstellung der Arbeit ausgenutzt wurden. D.h. nur durch die schrittweise Verfeinerung von Spezifikationen und ein jeweils anschließendes Testen und Bewerten der jeweils generierten Systeme bzw. Diagrammspracheditoren war es möglich, solch verhältnismäßig komplexe Beispielanwendungen zu erstellen.

Um zu einem implementierbaren Generierungskonzept zu gelangen, mussten eine Reihe von Problemen auch auf pragmatische Weise gelöst werden. Dazu zählt beispielsweise die Verwendung von Java als Kommandosprache für die Aktionen, das Konzept der Ak-

tionsresultate, die Generierung einer API, die bei der Programmierung der Aktionskommandos verwendet werden kann, die Verknüpfung der Modellelemente (z.B. Aktionen zu Tupeltypen) und die Verwendung von Entwurfsmustern sowohl bei der Erstellung des Generators als auch beim Entwurf der Zielarchitektur generierter EMU-Systeme (z.B. Zugriffsadapter).

Die interne – für den Entwickler transparente – Funktionsweise der generierten EMU-Systeme verwendet erweiterte Übersetzerbauprinzipien (z.B. inkrementelle Attributauswertung [RT88] zur Berechnung der Zugriffsadapter).

Die Implementierung wird unter anderem deshalb als *prototypisch* bezeichnet, weil die generierten EMU-Systeme in ihrer vollen Funktionalität nicht verteilt ablaufen. Jedoch wurden im Rahmen dieser Arbeit in zwei Software-Entwicklungsprojekten vereinfachte Versionen von EMUGEN implementiert, die verteilte Systeme generieren.

Bei beiden Ansätzen werden aus einem Datenmodell, das im Wesentlichen dem EMU-Datenmodell entspricht, und einem Layoutmodell lauffähige Client-Server-Systeme generiert. Dabei werden durch den Client die Benutzungsoberflächen und durch den Server der MAK verwaltet. In [Gri00] stellen sowohl Client als auch Server Java-Programme dar, die mittels *Remote-Method-Invocation* (*RMI*) miteinander kommunizieren. In [HH02] greift der Client über ein WML-Dokument auf einen durch ein generiertes Servlet verwalteten MAK zu.

Weitere Einschränkungen der aktuell implementierten Version von EMUGEN bestehen darin, dass nur die Menge an Interaktionsobjekten unterstützt wird, die ausreicht, um den Generierungsansatz bei interaktiven Mehrbenutzersystemen zu vermitteln. D.h. es können natürlich bei realistischen Anwendungen Anforderungen bezüglich komplexer Interaktionsobjekte auftreten, die beispielsweise strukturierte Daten durch Tabellen oder Bäume interaktiv für den Benutzer darstellen. Hier musste auf eine Vollständigkeit verzichtet werden. Jedoch wurde durch die Einbeziehung der Generierung diagrammsprachlicher Editoren exemplarisch gezeigt, dass auch durchaus komplexe Benutzungsoberflächen generierbar und vor allem auch in ein Gesamtsystem integrierbar sind. Ferner ist es bereits bei der aktuellen Version möglich, beliebige externe Interaktionsobjekte einzubinden, aber eben nicht, beliebige zu generieren.

## 8.2 Weitere Arbeiten

Die im vorherigen Abschnitt angesprochenen, fehlenden Aspekte der aktuellen Implementierung ergeben natürlich sofort entsprechende Erweiterungsmöglichkeiten von EMUGEN.

Während die oben erwähnten, ansatzweise implementierten Erweiterungen hinsichtlich einer verteilten Ausführung jeweils auf einem im Generator fest eingebauten Kommunikationskonzept basieren, wäre die strikte Fortführung des generativen Ansatzes die Konzeption einer Modellierungssprache zur formalen Beschreibung des Kommunikationsaspekts. D.h. man hätte dann zusätzlich zum Daten-, Aufgaben-, Benutzer- und Layoutmodell eine weitere Eingabekomponente für EMUGEN. In [Gri00] wurde dieser Ansatz bereits in Grundzügen durch eine Erweiterung des Layoutmodells realisiert. Hier kann der Entwickler synchron und asynchron kommunizierende Benutzungsoberflächen(-teile) spezifizieren.

Bei der Konzeption einer Client-Server-Architektur für EMU-Systeme unter Berücksichtigung des EMU-Benutzermodells wäre es wohl sinnvoll, die Zugriffsadapter der einzelnen Benutzer durch Client-Proxy [GHJ96] am Server zu verwalten. Ein ähnlicher Ansatz wird in [Har01] zur Verwaltung endgerätespezifischer, interaktiver Netzdienste verfolgt. Die dort verwendeten, server-seitig ablaufenden Netzdienstprozesse erstellen dabei Client-Benutzungsoberflächen, die von der jeweiligen Dienst- und Client-Beschreibung (*client-profile*) abhängen. Dieser Ansatz könnte in das EMU-Generierungskonzept integriert werden. Dabei wäre die Client-Benutzungsoberfläche zusätzlich vom jeweiligen Benutzer bzw. seinen aktuellen Eigenschaften und Rollen abhängig. Dieser Ansatz wäre sowohl für zustandsorientierte, wie auch zustandslose Verbindungen (mit den damit verbundenen Einschränkungen) denkbar.

Weiterhin wäre es sinnvoll, die in dieser Arbeit als Beispiele informell eingeführten Konzepte von *TUM* und *Ad-hoc Workflows* genauer auszuarbeiten bzw. zu formalisieren.

Die deklarativen Beschreibungstechniken, die im Aufgaben- und Benutzermodell verwendet wurden, sollten es ferner ermöglichen, Verifikationsansätze durchzuführen. Möglich wäre es beispielsweise, die als *TUM* eingeführte, graphische Darstellung als konstruktive Spezifikation eines temporallogischen Modells zu verwenden. Konkret könnte der TUM-Grapheditor beispielsweise als Front-End für das in [Bra95] entwickelte Rahmenwerk zur Verifikation abstrakter Benutzungsoberflächen dienen. Aussagen über die (konstruktive) Spezifikation könnten dann als temporallogische Formel formuliert werden und mit Hilfe eines Model-Checkers [CS99] automatisch überprüft werden. Ansatzweise wurde dies bereits im Rahmen dieser Arbeit in einer Diplomarbeit [Lan01] durchgeführt. Dort wurde als konstruktive, graphische Spezfikationssprache eine eingeschränkte Form von Harel-Automaten [Har87] verwendet und ein entsprechender Grapheditor generiert. Die vom Model-Checker erzeugten Gegenbeispiele bei einer nicht erfüllten Formel können bereits in [Lan01] in der graphischen Spezifikation visualisiert werden.

Als wichtigste Erweiterung ergibt sich jedoch die Überarbeitung des (implementierten) EMU-Generierungskonzepts. Aus Sicht des Autors sollte dies im Idealfall nicht durch einen einzelnen Bearbeiter erfolgen, weil die Allgemeinheit, die grundsätzlich mit einem generativen Ansatz verbunden ist, mit den Details, die bei einer praktisch anwendbaren Umsetzung zu berücksichtigen sind, nur in einem Entwicklerteam in angemessener Weise zu integrieren sein dürften.

# Anhang

## A) Abstrakte Syntax der EMU-Modelle

Wir verwenden zur Darstellung der abstrakten Syntax die aus [Eic93] bekannte und im Rahmen dieser Arbeit durch ein SEP [DW01] implementierte graphische Darstellung von Datentypen. Da diese auch als Eingabe für das EMU-Datenmodell verwendet werden kann, wäre es auch möglich, daraus eine graphische Entwicklungsumgebung für EMUGEN selbst zu generieren.

Ein Datenmodell besteht aus mindestens einer Tupelproduktion, gefolgt von einer beliebig langen Sequenz weiterer Produktionen, wobei ein Typidentifikator höchstens an einer linken Produktionsseite vorkommen darf, also höchstens einmal definiert wird. Die erste Tupelproduktion in der textuellen Aufschreibung definiert den Starttyp $start_{DM}$ eines Datemodells *DM*.

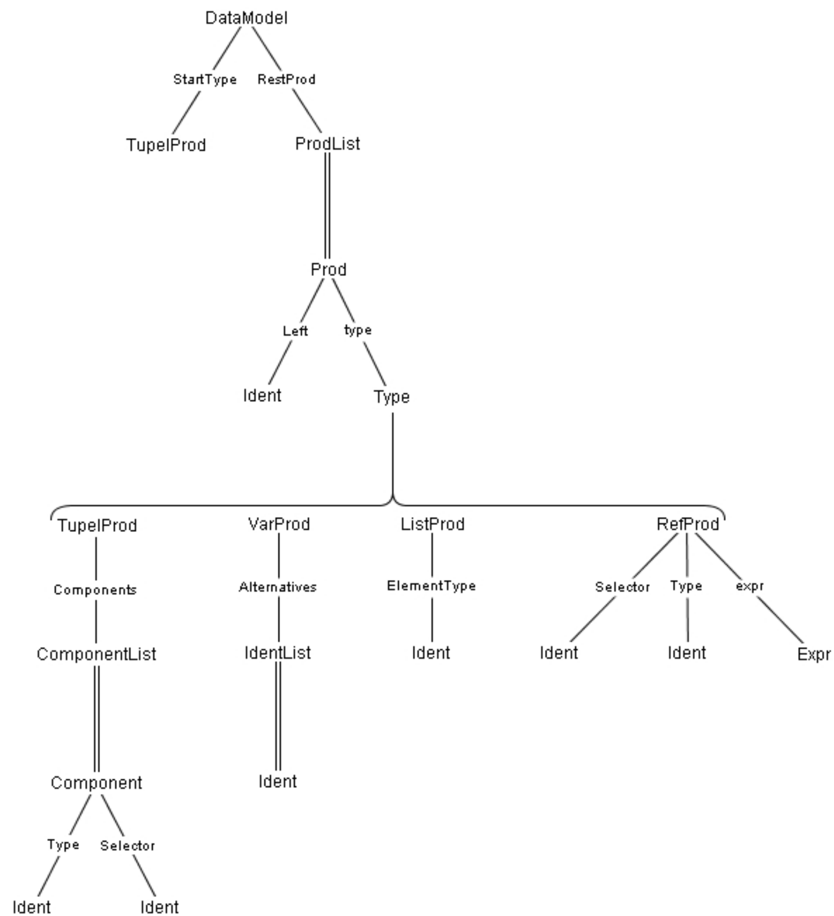


**Abbildung A-1: Abstrakte Syntax des EMU-Datenmodells**

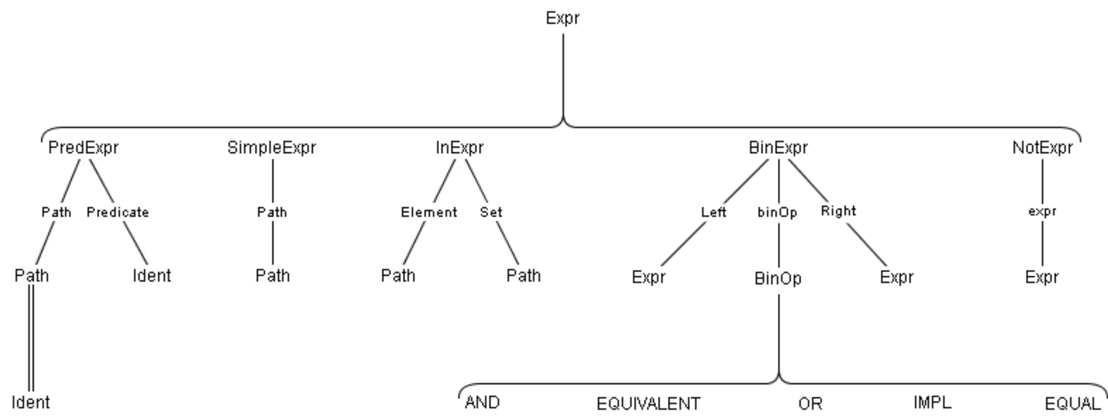


**Abbildung A-2: Abstrakte Syntax für Ausdrücke**

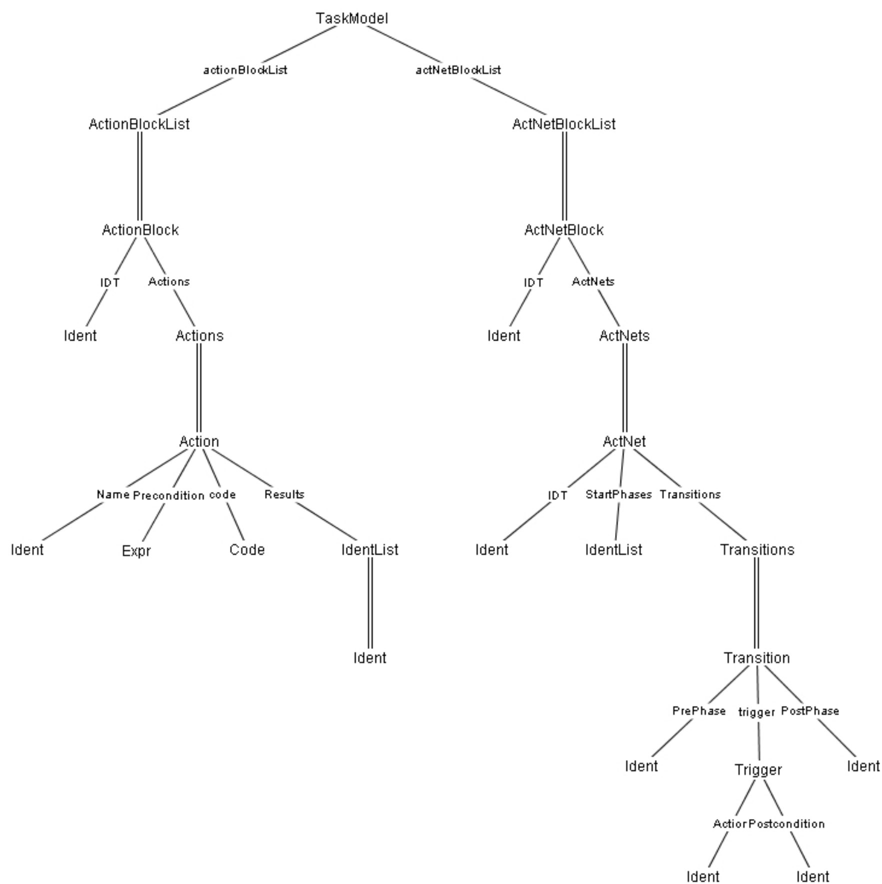


**Abbildung A-3: Abstrakte Syntax des EMU-Aufgabenmodells**

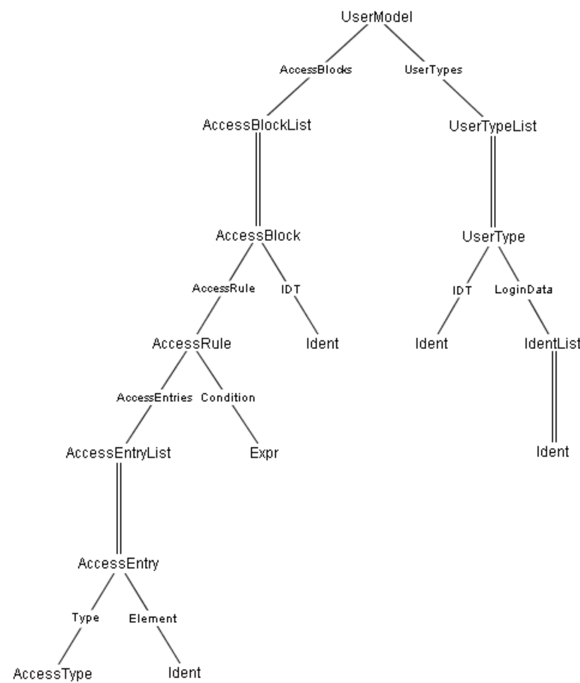


**Abbildung A-4: Abstrakte Syntax des EMU-Benutzermodells**

## B) Konkrete Syntax der EMU-Modelle

Zur Beschreibung der konkreten (textuellen) Syntax der formalen Sprache für die EMU-Eingabemodelle verwenden wir eine übliche BNF-Notation, wie sie etwa auch in [Bro98] verwendet wird. Dabei werden terminale Symbole der zu definierenden formalen Sprache fett gekennzeichnet.

### 1) Datenmodell

```
<Datamodel> ::=  <TupelProd> <Prod>*
<Prod>      ::=  <TupelProd>     |
                 <VarProd>       |
                 <ListProd>      |
                 <RefProd>       |
<TupelProd> ::=  <Ident>::=<Component>*
<Component> ::=  <Ident>{:<Ident>)}
<VarProd>   ::=  <Ident>::=<Ident>|<Ident>(|<TypeIdent>)*
<ListProd>  ::=  <Ident>::=<Ident>*
<RefProd>   ::=  <Ident>::=<Ident>-><Ident>{[<Expr>]}
<Expr>      ::=  {<Path>.}<Ident>(<Path>)   |
                 <Path>                     |
                 <Path>in<Path>             |
                 !< Expr>                   |
                 <Expr><BinOp><Expr>
<Path>      ::=  {<Ident>(.<Ident>)*}
<BinOp>     ::=  &|||->|<->|==
```

### 2) Aufgabenmodell

```
<TaskModel>      ::=  (<ActionBlock*><ActNetBlock*>
<ActionBlock>    ::=  actions of <Ident> {<Action>*}
<Action>         ::=  <Ident>[Expr]->{:<Java-Code>:}
                      {<Ident>(|<TypeIdent>)*};
<ActNetBlock>    ::=  actnet of <Ident> {<Transition>*}
<Transition>     ::=  <Ident>-><Ident>{.<Ident>}-><Ident>
```

### 3) Benutzermodell

```
<UserModel>      ::=  (<AccessBlock*><UserTypes>)*
<AccessBlock>    ::=  access on <Ident> {<AccessRule>*}
<AccessRule>     ::=  [Expr] <AccessType><Ident>
                                (,<AccessType><Ident>)*;
<AccessType>     ::=  read | write | exec
<UserTypes>      ::=  User::=<Ident>(<Ident>*)
                                (|<Ident>(<Ident>*))*
```

## C) API der generierten Knotenklassen

Mit dem UML-Diagramm [OMG01] in Abbildung A-5 wird schematisch die API eines EMU-Systems gezeigt. Um den Zusammenhang der generierten mit den statisch vorhandenen Klassen (grau gefärbt in Abbildung A-5) zu zeigen, werden die generierten Klassen gemäß der folgenden Beispieleingabe dargestellt:

```
T::=T1:S1 T2:S2 T3:S3
A::=A1|A2|A3
L::=E*
R::=S->T
```

Ferner erhält das Tupel `T` im Aufgabenmodell die Aktion `a` und ein Aktivitätsnetz, das aus den Phasen `p1` und `p2` besteht.

Die Architektur der statisch vorhandenen Klassen der Metatypen bilden ein Kompositionsmuster [Pre95, GHJ96]. Die generierten Klassen werden abhängig von ihrem Metatyp von den statischen Klassen abgeleitet. Die API der statischen Klassen ermöglicht die Implementierung anwendungsunabhängiger Aspekte.

Die API der generierten Knotenklassen der Kompositionstypen Tupel, Variante und Liste verfügen neben den üblichen Zugriffs- und Manipulationsmöglichkeiten über die Methoden `getCopy()`, die eine Kopie des gesamten Teilbaums eines inneren Knotens liefert, und `setCopy(..)`, die einen übergebenen Baum als neuen Teilbaum einsetzt. Die (generierte) Implementierung von `setCopy(..)` ist nicht ganz trivial, weil es dabei gilt, die Benutzungsoberflächen, die den betroffenen Teilbaum momentan visualisieren, geeignet zu aktualisieren (vgl. Diskussion in Abschnitt 4.4.4).

Jede Aktion wird als Instanzvariable der zugehörigen Tupelklasse (im Beispiel: `T.a`) verwaltet. Eine solche Instanzvariable verweist dabei auf ein Objekt einer anonymen Klasse, welche die Schnittstelle `Action` und damit deren Methoden `command()` und `enabled()` abhängig von der Eingabespezifikation implementiert. `Action.command()` liefert als Rückgabewert den Index des Resultats. Jede Phase eines Aktivitätsnetz ergibt eine boolesche Instanzvariable in der zugehörigen Tupelklasse (im Beispiel: `T.p1`, `T.p2`).

Die generierten Knotenklassen instanziieren ferner das Besuchermuster [GHJ96], wodurch der Entwickler in der Lage ist, den MAK als Syntaxbaum im Sinne des Übersetzerbaus zu verwenden und semantische Auswertungen durch Unterklassenbildung der Klasse `Visitor` vorzunehmen. Für die Standardtraversierungen des Syntaxbaums (Top-Down, Bottom-Up) werden entsprechende Methoden der Knotenklassen bereitgestellt.

Ein EMU-System durch die Ausführung der Methode `EMUSystem.main(..)` gestartet.

Zur besseren Übersicht wurden in Abbildung A-5 nur die API für den Entwickler und nicht die Teile, die zur technischen Abwicklung (z.B. zur Kommunikation gemäß der MVC-Architektur) notwendig sind, eingezeichnet.

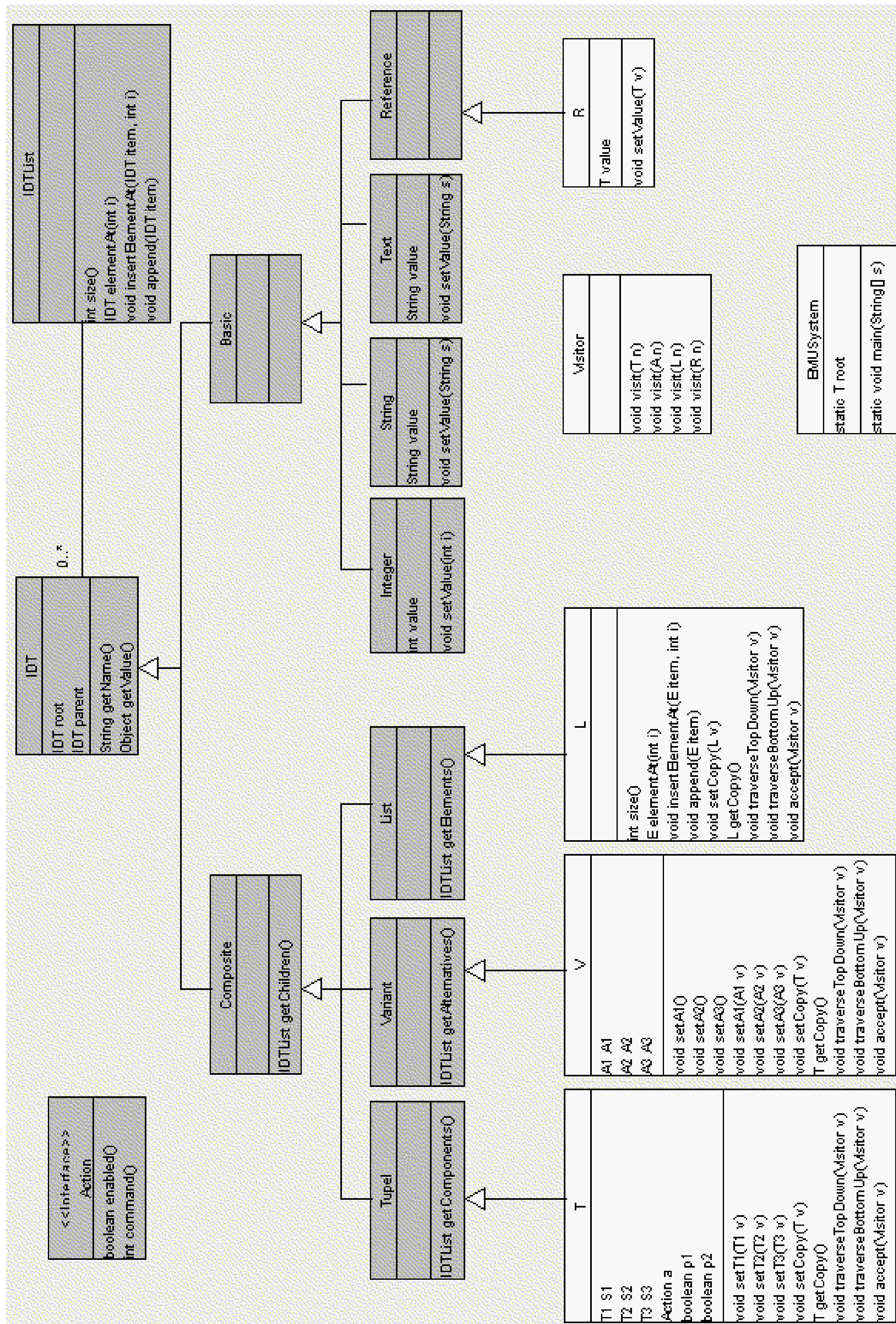


**Abbildung A-5: Schematische Darstellung der API von EMU-Systemen**

# Literaturverzeichnis

[Aal98] W.M.P. van der Aalst. *The Application of Petri Nets to Workflow Management*. The Journal of Circuits, Systems and Computers, 8(1):21-66, 1998.
*Beschreibt ausführlich die Anwendung von Petrinetzen zur Beschreibung von Workflow-Schemata und auch Ansätze von ad hoc Workflows.*

[Bal00] H. Balzert. *Lehrbuch der Software-Technik: Software-Entwicklung*. 2. Auflage, Spektrum Akademischer Verlag, 2000.
*Referenz für grundlegende, pragmatische Methoden zur Software-Entwicklung.*

[Bau96] B. Bauer. *Generating User Interface from Formal Specifications of the Application*. In Proceedings of CADUI'96, Presses Universitaires de Namur, 1996.
*Verwendet algebraische Spezifikationen (Axiome, Vor- und Nachbedingungen) zur Dialoggenerierung.*

[Bra95] A. Brandl. *Verifizierbare und interpretierbare Modelle graphischer Benutzungsoberflächen*. Diplomarbeit, Technische Universität München, 1995.
*Verifikation graphischer Benutzungsoberflächen mit Temporallogik. Wie in [Lan01] gezeigt, kann man so auch graphisch (mit generierten Grapheditoren) beschriebene Abläufe verifizieren.*

[Bra01] A. Brandl. *EmuGen: A Generator for Multiple-User Interfaces.* In Proceedings of HCI International, LEA, 2001.
*Beschreibt das EMU-Beispiel „Praktikum" mit Ausführung.*

[Bro98] M. Broy. *Informatik: Eine grundlegende Einführung: Band 1: Programmierung und Rechenstrukturen*. 2.Auflage, Springer, 1998.
*Referenz für grundlegende formale Begriffe.*

[Boe81] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
*Einführung des Wasserfallmodells, woran sich auch das Vorgehensmodell zur Entwicklung von EMU-Systemen orientiert.*

[BB87] T. Bolognesi and E. Brinksma. *Introduction to the ISO Specification Language LOTOS*. Computer Networks and ISDN Systems (14). Elsevier North-Holland, 1987.
*LOTOS wird bei Mastermind als Dialogmodell und Sprache zur Beschreibung der Kommunikation der einzelnen Modelle verwendet.*

[BC91] L. Bass, J. Coutaz. *Developing Software for the User Interface*. Addison-Wesley, 1991.
*Beschreibt die PAC-Architektur, die vom verwandtem CADUI-Werkzeug MASTERMIND als Architektur der generierten interaktiven Systeme verwendet wird.*

[BDD93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, R. Weber. *The Design of Distributed Systems – An Introduction to FOCUS (revised version)*. SFB-Bericht 342/2-2/92 A, Technische Universität München, 1993.
*Hieraus entstammt der Ansatz zur Semantikbeschreibung interaktiver Systeme.*

[BGK99] P.J. Barclay, T. Griffiths, J. McKirdy, N. W. Paton, R. Cooper, J. Kennedy. *The Teallach Tool: Using Models For Flexible User Interface Design*. In Proceedings of CADUI'99, LEA, 1999.
*Teallach ist ein zu EMUGEN vergleichbares CADUI-Werkzeug. In dieser Arbeit wird das Gesamtkonzept von Teallach dargestellt, während in den weiteren Arbeiten zu Teallach das Aufgabenmodell genauer betrachtet wird.*

[BK99] A. Brandl, G. Klein. *FormGen: A Generator for Adaptive Forms Based on EasyGUI*. In Proceedings of HCI International, LEA, 1999.
*Zeigt die automatische Generierung dynamischer Formulare aus abstrakten Datentypen.*

[Che76] P.M. Chen. *The entity relationship model - toward a unified view of data*. ACM Transactions on Database Systems, 1, Seiten 9-36, 1976.
*Zur Motivation des modellbasierten UI-Entwicklungsansatzes wird eine Analogie zur Datenbankvorgehensweise hergestellt. Potenziell könnte auch ein ER-Modell bzw. ein UML-Klassendiargramm als Datenmodell verwendet werden. Das EMU-Datenmodell wird mit dem ER-Modell verglichen.*

[Cod70] E.F. Codd. *A relational model of data for large shared data banks*. Communications of the ACM, 13, Seiten 377-387, 1970.
*Das EMU-Datenmodell wird mit dem relationalen Modell verglichen.*

[CL99] L.L. Constantine, L.A.D. Lockwook. *Software for Use - A Practical Guide to the Models and Methods of Usage-Centered Design*. ACM Press, 1999.
*Beschreibt UI-Designrichtlinien, die von EMU zu berücksichtigen sind um sie bei den automatische generierten Uis einzuhalten. Ferner wird es dem Entwickler ermöglicht, über das Layoutmodell diese Richtlinien zu implementieren.*

[CMN83] S.K. Card, T.P. Moran, A. Newell. *The Psychology of Human-Computer Interaction*. LEA, 1983.
*Einführung des GOMS-Modells, einer Alternative zum EMU-Aufgabenmodell.*

[CS99] E.M. Clarke, *H. Schlingloff. Model Checking. To appear in: Handbook of Automated Deduction*. A. Voronkov (ed.), Elsevier, 1999.
*Beschreibt Grundlagen des Model-Checkings. Im Ausblick wird die Möglichkeit der Verifikation des Aufgaben- und Benutzermodells beschrieben.*

[DeM79] T. DeMacro. *Structured Analysis and System Specification*. Yourdon Press, 1979.
*SA benutzt mit dem Data Dictionary ein zum EMU-Datenmodell ähnliches Datenmodell und damit einen hierarchischen Ansatz. Der in SA propagierte top-down-Ansatz kann gut mit EMUGEN vollzogen werden. Grapheditoren für SA-Datenflußmodelle könnte man mit EMUGEN generieren (vgl. TUM-Modell in Kap. 5).*

[Den91] E. Denert. *Software-Engineering*. Springer, 1991.
*Beschreibt eine Vorgehensweise, die auch mit EMUGEN praktiziert werden kann. EMUGEN kann als Implementierung der dort vorgeschlagenen "idealen Maschine" angesehen werden.*

[DW01] F. Deißenböck, S. Winter. *VCG: Ein Werkzeug zur visuellen Erstellung abstrakter Datentypen*. Systementwicklungsprojekt, Technische Universität München, 2001.
*Werkzeug zur visuellen Darstellung von Datentypen nach [Eic93].*

[Eic93] J. Eickel. *Übersetzung von Programmiersprachen*. Vorlesung, Technische Universität München, 1993.
*Daraus wurde etwa das Konzept der Datentypvisualisierung entnommen.*

[Eic98] J. Eickel. *Generierung von Bedienoberflächen*. Vorlesung, Technische Universität München, 1998.
*Spezifikation von Benutzungsoberflächen im Hinblick auf ihrer Generierung mit Zustandsautomaten und Datentypen sowie die Generierung des Layouts über Layoutmodelle. Grundlegende Aspekte zur Anwendung von Übersetzerbaukonzepten zur Generierung von UIs.*

[End94] A. Endres. *Graphische Benutzungsoberflächen*. Vorlesung, Technische Universität München, 1994.
*Beschreibt genaue (DIN) UI-Terminologie und bringt Überblick über viele relevante UI-Aspekte.*

[Esp95] J. Esparza. *Petrinetze*. Vorlesung, Technische Universität München, 1995.
*Referenzliteratur zu Petrinetze. Die Aktivitätsnetze des EMU-Aufgabenmodells und die Workflowbeschreibungssprache des Beispiels „Ad-hoc Workflows" sind an Petrinetze angelehnt.*

[FS98] M. Frank, P. Szekely. *Adaptive Forms: An interaction paradigm for entering structured data*. In Proceedings of IUI'98, ACM, 1998.
*Vergleichbarer Ansatz wie FORMGEN bzw. BOSS zur Generierung dynamischer Formulare.*

[Gan78] H. Ganzinger. *Optimierende Erzeugung von Übersetzerteilen aus implementierungsorientierten Sprachbeschreibungen*. Dissertation, Technische Universität München, 1978.
*Dynamische AGs und die im formalen Teil verwendete Baumnotation werden dort eingeführt. Durch die Varianten des EMU-Datenmodells können die grundlegenden Eigenschaften dynamischer AGs, wie attributwertabhängige Produktionsanwendung, modelliert werden.*

[Gil99] M. Gille. *Diagramm-Editoren: Generierung aus objektorientierten Modellinformationen*. Spektrum Akademischer Verlag, 1999.
*Beschreibt ein zu GRACE ähnliches Werkzeug. Jedoch hat GRACE – in Bezug auf das EMU-Generierungskonzept – eine Reihe wichtiger Vorteile, weshalb es in EMUGEN verwendet wird.*

[Gri00] P. Grieser. *Generierung von Formularen zum Zugriff auf entfernte Datenstrukturen*. Systementwicklungsprojekt, Technische Universität München, 2000.
*Client-Server-Version einer vereinfachten Version von EMUGEN. Bei den generierten Systemen kommunizieren Client (Formular) und Server (Datenstruktur/Modell) über RMI.*

[GFJ96] G. Groh, K.P. Fähnrich, C. Janssen, Hrsg. *Werkzeuge zur Enwicklung graphischer Benutzungsoberflächen*. Handbuch der Informatik, Oldenburg, 1996.
*Konferenzband mit einer Einteilung und Beschreibung von CADUI-Werkzeugen (u.a. GENIUS).*

[GHJ96] E. Gamma, R. Helm, R. E. Johnson, J. Vlissides. *Entwurfsmuster*. Addison Wesley, 1996.
*Die generische UI-Entwicklung hat viele Gemeinsamkeiten zur Entwicklung nach Entwurfsmustern, insbesondere die Verwendung von Rahmenwerken.*

[GJS96] J. Gosling, B. Joy, G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

*Java ist die Kommandossprache zur Implementierung der EMU-Applikationsschnittstelle und der semantischen Aktionen.*

[GR83] A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.

*Einige UI-Grundkonzepte (wie MVC) entstammen Smalltalk.*

[Har87] D. Harel. *Statecharts: A Visual Formalism for Complex Systems*. In Science of Computer Programming, Elsevier Science Publisher, 1987.

*Zum Vergleich mit anderen Spezifiktionsmöglichkeiten für Dialoge und Aufgaben.*

[Har01] R. Haratsch. *A client-server architecture for providing client-specific network services on the application layer.* In Proceedings of APC 2001.

*Zeigt eine Client-Server Architektur, die eine verteilte Ausführung von EMU-Systemen mit verschiedenen, evtl. auch mobilen, Endgeräten ermöglichen könnte.*

[Hud99] S. E. Hudson. *CUP LALR Parser Generator for Java*. http://www.cs.princeton.edu/~appel/modern/java/CUP, 1999.

*EMUGEN stellt ähnlich wie Parser-Generatoren ein Werkzeug zur Erstellung eines Front-Ends dar. Genau wie bei CUP bzw. YACC erfolgt die Schnittstelle zur Applikation bzw. zur semantischen Analyse durch (prozedurale) Aktionen.*

[Hus99] A. Hussey. *Formal Object-Oriented User-Interface Design*. Technical Report, Software Verification Research Centre School of Information Technology Univerity of Queensland, 1999.

*Zeigt genauer die Ansätze von [HC99]. Unter Verwendung von UAN wird ferner gezeigt, wie abstrakte Object-Z Spezifikationen verfeinert werden können, wodurch Benutzerinteraktionen auf Layoutebene entstehen.*

[HC99] A.Hussey, D.Carrington. *Model-Based Design of User-Interfaces Using Object-Z*. In Proceedings of the Third International Conference on Computer-Aided Design of User Interfaces, 1999.

*Eine verwandte CADUI-Arbeit. Interssant, weil mit Object-Z ebenfalls eine konkrete UI-Spezifikationssprache vorgestellt wird. Jedoch erfolgt keine automatische Generierung, sondern lediglich die Darstellung einer formale Vorgehensweise.*

[HCI01] ACM SIGCHI. *Curricula for Human-Computer Interaction*. http://sigchi.org/cdg/index.html , 2001.

*Liefert einen Überblick über HCI und eine Sammlung der wichtigsten Webseiten.*

[HH02] M. Heller, M. Hilgers. *Generierung von WEB-Benutzungsoberflächen basierend auf Servlet- und WML-Technologie*. Systementwicklungsprojekt, Technische Universität München, 2002.

*Ähnlich wie [Gri00]. Hier stellen die Formulare WML-Dokumente dar und der MAK wird durch ein Servlet verwaltet.*

[ISO94] ISO 9241. *Ergonomic Requirements for Office Work with Visual Display Terminals*. Draft International Standard, 1994.

*Normierte Gestaltungsregeln, wie sie insbesondere bei der Formualargenerierung von BOSS und FORMGEN berücksichtigt werden.*

[Joh92] P. Johnson. *Human Computer Interaction: Psychology, Task Analysis and Software Engineering*. McGraw-Hill, 1992.

*Beschreibung kognitiver UI-Aspekte. Einführung von Task Knowledge Structure TKS, dem Aufgabenmodell für das CADUI-Werkzeug ADEPT.*

[JBS97] S. Jablonski, M. Böhm, W. Schulze. *Workflow-Management - Entwicklung von Anwendungen und Systemen*. dpunkt-Verlag, 1997.
*Referenz zu Techniken, zur Vorgehensweise und zur Terminologie beim Workflow-Management.*

[JF88] R. E. Johnson, B. Foote. *Designing Reusable Classes*. Journal of Object-Oriented Programming, 1(2):22-25, 1988.
*Einer der Orginalartikel zu Frameworks, die in Kapitel 2 als UI-Werkzeuge betrachtet werden.*

[JMJ92] P. Johnsen, P. Markopoulus, H. Johnson. *Task Knowledge Structures: A specification of user task models and interaction dialogues*. In Proceedings of 11th Interdisciplinary workshop on informatics and psychology, 6, 1992.
*Beschreibung von TKS, einer Alternative zum EMU-Aufgabenmodell.*

[Kir99] M. Kirchmer. *Business Process Oriented Implementation of Standard Software.* 2.Auflage, Springer, 1999.
*Beschreibt, wie man Vorgänge bzw. Geschäftsprozesse in Standardsoftware umsetzt. In Kap. 2 erwähnt als Gegenpol zur Entwicklung bzw. Generierung von Individualsoftware.*

[Kle99] G. Klein. *Generating Graphical Editors for Graph-like Datastructures*. Diplomarbeit, Technische Universität München, 1999.
*Die Arbeit zu GRACE, dem von EMUGEN benutzten Grapheditor-Generator.*

[Kru99] V. Kruschinski. *Layoutgestaltung grafischer Benutzungsoberflächen: Generierung aus OOA-Modellen*. Spektrum Akademischer Verlag, 1999.
*Eine Dissertation aus dem JANUS-Umfeld. Die Layoutregeln sind hier festeinprogrammiert und informell beschrieben. Es geht hier vorrangig um ergonomische Aspekte.*

[KLM97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, J. Irwin. *Aspect-Oriented Programming*. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1997.
*In gewisser Weise kann der generative Ansatz von EMUGEN als eine konkrete Anwendung von AOP betrachtet werden.*

[KP88] G.E. Krasner, S.T.Pope. *A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80*. Journal of Object-Oriented Programming, 1(3):26-49, 1988.
*Orginalartikel zu MVC. Der MAK des EMU-Systems entspricht einem hierarchischen MVC-Modell; die Einbenutzer-UIs der einzelnen Benutzer entsprechen jeweils MVC-(View/Controller)-Komponenten.*

[Lan01] T. Lange. *Konzeption und Implementierung eines Werkzeugs zur graphischen Spezifikation und Verifikation von Benutzungsoberflächen auf Basis von Model-Checking*. Diplomarbeit, Technische Universität München, 2001.
*Wendet Teile des Generierungskonzepts dieser Arbeit an.*

[Lar92] J.A. Larson. *Interactive Software: Tools for Building Interactive User Interfaces.* Prentice Hall, 1992.
*Zeigt eine Reihe von Spezifikationsmöglichkeiten für Benutzungsoberfläche wie Statecharts, Datenflussdiagramme mit Eingabeslots, Petrinetze, etc.*

[Lev92] J.R. Levine, T. Mason, D. Brown. *Lex & Yacc*. O'Reilly, Sebastaopol, 1992.
*Zum Vergleich von EMUGEN mit Parsergeneratoren.*

[Lie86] H.Lieberman. *Using prototypical objects to implement shared bahavior in object-oriented systems*. In Proceedings of Object-Oriented Programming Systems, Languages, and Applications, 1986.

*Beschreibt wohl erstmalig das Delegationsmuster, welches im erzeugten Code der EMU-Systeme angewandt wird.*

[LS96] F.Lonczewski, S.Schreiber. *Generating User Interfaces with the FUSE-System*. Technischer Bericht, Technische Universität München, Technical Report TUM-I9612, 1996.

*Beschreibt das CADUI-Werkzeug FUSE.*

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs,1989.

*CCS wird als Alternative für das EMU-Aufgabenmodell betrachtet.*

[Mor81] T. P. Moran. *The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems*. International Journal of Man-Machine Studies, 15, 1981.

*Einführung des kognitiven Dialogmodells CLG.*

[NO90] A. Nye, T. O'Reilly. *X Toolkit Intrinsics Reference Manual*. O'Reilly & Associates, Inc., 1990.

*Beispiel für ein (sehr komplexes) UI-Toolkit (Sammlung von C Prozeduren). Das BOSS-System verwendet als UI-Toolkit das hier beschriebene X-Athena-Toolkit (genauer: C++ Wrapperklassen, die X-Athena verwenden).*

[Obe96] A. Oberweis. *Modellierung und Ausführung von Workflows mit Petrinetzen*. Springer-Verlag 1996.

*Die vielfältigen dort aufgeführten Aspekte hatten einen wichtigen Einfluß auf den Workflow- bzw. Mehrbenutzeraspekt bei EMUGEN.*

[Oes98] B. Oestereich. *Objektorientierte Softwareentwicklung Analyse und Design mit der Unified Modeling Language*. 4. Auflage (UML 1.3), Oldenburg, 1998.

*Das dort beschriebene Vorgehensmodell kann auch mit EMUGEN durchgeführt werden. Einige Probleme der OO-Modellierung, z.B. bzgl. Vererbung/Delegation, werden dort angesprochen und hatten Einfluß auf das EMU-Datenmodell.*

[Ous95] J. K. Ousterhout: *Tcl und Tk: Entwicklung graphischer Benutzerschnittstellen für das X Windows System*. Addision-Wesley, 1995.

*Skriptsprache und UI-Toolkit zur Implementierung von Benutzungsoberflächen. Wird von FUSE/PLUG-IN verwendet.*

[OMG01] Object Management Group. *OMG Unified Modeling Language Specification*. Version 1.4, ftp://ftp.omg.org/pub/docs/formal/01-09-67.pdf , 2001.

*Anwendungsfall- und Aktivitätsdiagramme von UML hatten Einfluss auf das Beispiel TUM. Einige Autoren verwenden Teile UML als Eingabe (z.B. JANUS), allerdings meist nur Klassendiagramme.*

[Pat99] F. Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 1999.

*Aktuelles, im Sinne des Autors formales, Buch zu CADUI.*

[Pfa85] G.E. Pfaff. *User Interface Management Systems*. Springer, 1985.

*Buch zur Konferenz in Seeheim, wo das gleichnamige Grundmodell für Benutzungsoberflächen erstmals vorgestellt wurde.*

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.
*Hatte wichtigen Einfluß auf die Software-Architektur und Kommunikation in den generierten Klassen.*

[Pod00] M. Podolsky. *Ein durchgängiger, integrierter Ansatz für den Entwurf objektorientierter, Workflow-basierter Systeme*. Dissertation, Technische Universität München, 2000.
*Verwendet ebenfalls Petrinetze zur Ablauf-Definition von Workflows.*

[Pue96] A.R. Puerta. *Issues in Automatic Generation of User Interfaces in Model-Based Systems*. Computer-Aided Design of User Interfaces, ed. by Jean Vanderdonckt. Presses Universitaires de Namur, Namur, Belgium, 1996.
*Diskutiert Pro und Contra der automatischen Generierung von BOs bei MB-UIDE. Der Artikel zeigt, dass viele Autoren im CADUI-Bereich die automatische Generierung völlig ablehnen und eher eine informelle Unterstützung fordern. D.h. modellbasiert heißt nicht generativ. Daher können deren CADUI-Werkzeuge auch nur bedingt mit EMUGEN verglichen werden.*

[Pue97] A.R Puerta. *A Model-Based Interface Development Environment*. IEEE Software, 14(4), Juli/August 1997.
*Beschreibt das verwandte System MOBI-D.*

[PCO99] A.R. Puerta, E. Cheng, T. Ou, J. Min. *MOBILE: User-Centered Interface Building*. In Proceedings of CHI'99, ACM, 1999.
*MOBILE ist die MOBI-D Komponente zur Definition des Layouts der Aufgabenbearbeitung. Dies kann hier auch abhängig von Benutzertypen definiert werden.*

[PH94] A. Poetzsch-Heffter. Developing Efficient Interpreters Based on Formal Language Specifications. In Compiler Construction, Volume 786 of Lecture Notes in Computer Science, Springer, 1994.
*Zeigt eine Alternative zur Referenzsemantik von EMU.*

[RBP91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy. *Object-Oriented Modeling and Design*. Englewood Cliffs, Prentice Hall, 1991.
*Beschreibt das OOA-Modell, das von anderen MB-UIDE, z.B. vom Janus-System, als Modellierungssprache für das Datenmodell verwendet wird.*

[RHG99] J. Robins, D. Hilbert, A. Gauthier. *GEF: Graph Editing Framework*. http://www1.ics.uci.edu/pub/arch/gef/ , 1999.
*Wird in der Arbeit als Beispiel eines komplexen Rahmenwerks aufgeführt. Wurde als mögliche Implementierungsgrundlage der generierten Grapheditoren in Betracht gezogen.*

[RT88] T.W. Reps, T. Teitelbaum. *The Synthesizer Generator – A System For Constructing Language-Based Editors*. Springer, 1988.
*Beschreibt die zur Berechnung der Adapter in EMU-Systemen notwendige inkrementelle Attributauswertung.*

[Sch95] R. Schwab. *Generierung von Standardbedienoberflächen aus Applikationsbeschreibungen*. Diplomarbeit, Technische Universität München, 1995.
*Entwicklung des FIRE-Systems, welches (abgesehen vom Mehrbenutzeraspekt und Grapheditoren) EMUGEN ähnelt, weil es ebenfalls direkt auf dem Problembereichsmodell aufsetzt.*

[Sch97] S. Schreiber. *Spezifikatonstechniken und Generierungstechniken für graphische Benutzungsoberflächen*. Dissertation, Technische Universität München, 1997.
*Beschreibt das BOSS-System im Detail und das FUSE-Gesamtkonzept im Überblick.*

[Sch99] J. Schlichter. *Verteilte Anwendungen*. Vorlesung, Technische Universität München, 1999.
*Zur Abgrenzung von bestimmten Problemstellungen aus dem Gebiet der Verteilten Anwendungen wie etwa Verfahren zur Nebenläufigkeitskontrolle.*

[Shn98] B. Shneiderman. *Designing the User Interface - Strategies for Effective Human-Computer-Interaction*. 3.Auflage, Addison Wesley, 1998.
*Beschreibt das Vorgehen zur Entwicklung praktisch einsetzbarer Benutzungsoberflächen in seiner ganzen Vielfalt. Der Ansatz, in dynamischen Formularen nur die zur aktuellen Aufabenbearbeitung benötigten Teile anzuzeigen, wird dort in Abschnitt 13.6. erklärt.*

[Sta96] C.Stary. *Interaktive Systeme: Software-Entwicklung und Software-Ergonomie*. 2.Auflage, Vieweg, 1996.
*Definiert eine Reihe von Begriffen korrekt nach DIN. Zeigt ähnlich wie [Lar92] verschiedene UI-Spezifikationsansätze wie z.B. auch algebraische Spezifikationen im Überblick. Geht ähnlich wie [Shn98] auf UI-Entwicklungmethoden und dabei auch auf psychologische Faktoren und Modelle (z.B. GOMS) ein.*

[Sti99] K. Stirewalt. *MDL: A Language for Binding UI Models*. In Proceedings of the Third International Conference on Computer-Aided Design of User Interfaces, 1999.
*MDL ist die Aufgaben- bzw. Dialgomodellierungssprache von MASTERMIND, einem verwandten CADUI-Werkzeug.*

[Str92] B. Stroustrup. *Die C++-Programmiersprache*. Addison Wesley, 1992.
*Implementierungsnahe Teile von EMUGEN sind auch an C++ angelehnt.*

[Sze96] P. Szekely. *Retrospective and Challenges for Model-Based Interface Development*. In Computer-Aided Design of User Interfaces, ed. by Jean Vanderdonckt. Presses Universitaires de Namur, Namur, Belgium, 1996.
*Beschreibt eine allgemeine Architektur für MB-UIDEs und Anforderungen, die dabei zu berücksichtigen sind.*

[SR00] R.E.K. Stirewalt, S.Rugaber. *The Model-Composition Problem in User-Interface Generation*. In Automated Software Engineering, 7, Kluwer Academic Publisher, 2000.
*Eine aktuelle Veröffentlichung zum MASTERMIND-Projekt. Beschreibt Probleme bei der Integration verschiedener Modelle bei MB-UIDE. Viele dieser Probleme ergeben sich bei EMUGEN nicht, weil bei EMU die logische Struktur explizit vom Entwickler hierarchisch definiert wird und daraus die Struktur der weiteren Komponenten, wie z.B. der Formulare, abgeleitet wird.*

[SSC95] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, E. Salcher. *Declarative Models for User Interface Construction Tools: The Mastermind Approach*. In Proceedings of Engineering for Human-Computer Interaction, London, 1995.
*Erstes Papier zu MASTERMIND, einem verwandtem CADUI-Werkzeug.*

[TMP98] R.C. Tam, D. Maulsby, A. Puerta. *U-TEL: A Tool for Eliciting User Task Models from Domain Experts*. In Proceedings of IUI'99, ACM, 1998.
*Beschreibt eine Möglichkeit, aus einer natürlichsprachlichen Anforderungsdefinition ein konstruktives Modell für die MB-UIDE MOBI-D zu erzeugen. Ein (wegen einer Reihe von Vereinfachungen) technisch nicht besonders schwieriger Vorgang (Verben werden Aufgaben, Substantive werden Klassen, etc.).*

[Tee98] G. Teege. *CSCW-Systeme*. Vorlesung, Technische Universität München, 1998.
*Beschreibt u.a. die Benuzter-Interaktionen beim Workflow-Management und z.B. das WFMS FlowMark.*

[VB93] J. Vanderdonckt, F. Bodart. *Encapsulating Knowledge For Intelligent Automatic Interaction Object Selection.* In Proceedings of ACM Interchi'93, ACM, 1993.
*Beschreibt ein Konzept für abstrakte Interaktionsobjekte.*

[VP99] J. Vanderdonckt, A.Puerta. *Introduction to Computer-Aided Design of User Interfaces*. In Proceedings of the Third International Conference on Computer-Aided Design of User Interfaces, 1999.
*Zur Einführung in die Grundlagen von CADUI.*

[Vos00] G.Vossen. *Datenmodelle, Datenbanksprachen und Datenbakmanagementsysteme*. 4.Auflage, Oldenburg, 2000.
*Beschreibt neben den statischen Aspekten von Datenbanken auch die Bedeutung der dynamischen Aspekte. Wird als aktuelle Referenz zur Datenbank-Terminologie verwendet.*

[WfM96] Workflow Management Coalition. *The Workflow Reference Model (WFMC-TC-1003).* http://www.wfmc.org/standards/docs/TC-1011_term_glossary.pdf ,1996.
*Referenz zur Terminologie beim Workflow-Management.*

[Win93] G. Winskel. *The Formal Semantics of Programming Languages – An Introduction*. MIT Press, 1993.
*Referenz zur Terminologie und zu Techniken der Semantik.*

[WJK93] S. Wilson, P. Johnson, C. Kelly, J. Cunningham and P. Markopoulos. *Beyond Hacking: A Model Based Approach to User Interface Design*. In Proccedings of HCI, 1993.
*Beschreibt das ADEPT-System, bei dem wohl erstmals im CADUI-Bereich Aufgaben mit (LOTOS-) Ausführungsoperatoren verknüpft wurden.*

[WM97] R. Wilhelm, D. Maurer. *Übersetzerbau – Theorie, Konstruktion, Generierung*. 2.Auflage, Springer, 1997.
*Für grundlegende Aspekte aus dem Übersetzerbau (abstrakte Syntax, Attributierung, etc.).*

[WWK97] G. Weikum, D. Wodtke, A. Kotz-Dittrich, P. Muth, J. Weissenfels. *Spezifikation, Verififkation und verteilte Ausführung von Workflows in MENTOR*. Informatik Forschung und Entwicklung, 12, 1997.
*Spezifiziert Workflows mit State- und Activitycharts.*

[XML98] W3C. *Extensible Markup Language 1.0*. http://www.w3.org/TR/REC-xml. 1998.
*XML wird als alternatives Datenmodell diskutiert.*