

**Prüfer:** Prof. Dr. rer. nat. Jochen Ludewig

**Betreuer:** Dipl.-Inform. Ralf Melchisedech

**begonnen am:** 23. Oktober 1995

**beendet am:** 22. April 1996

**CR-Klassifikation:** I.6.1, I.6.2, I.6.7, I.6.8, K.6.1

Diplomarbeit Nr. 1345

**Konzeption und Realisierung  
einer Basismaschine  
für SESAM-2**

Ralf Reißing

Institut für Informatik  
Universität Stuttgart  
Breitwiesenstraße 20 – 22  
D – 70565 Stuttgart

# Inhaltsverzeichnis

<b>1 Einführung</b>	<b>1</b>
1.1 SESAM	1
1.1.1 Was ist SESAM?	1
1.1.2 Ziele	2
1.1.3 Implementierungen	2
1.2 Aufgabenstellung	3
1.3 Aufbau der Arbeit	3
<b>2 SESAM</b>	<b>4</b>
2.1 Lernmodelle	4
2.1.1 Definition	4
2.1.2 Ziele	4
2.1.3 Chancen	8
2.1.4 Risiken	9
2.2 SESAM	9
2.2.1 Vision und Ziele	9
2.2.2 Grenzen	11
2.3 SESAM-Modelle	11
2.3.1 Aufbau	12
2.3.2 Ausführung	14
2.4 Implementierungen	16
2.4.1 SESAM-1	16
2.4.2 SESAM-Lite	17
2.4.3 SESAM-2	17
<b>3 Die Basismaschine</b>	<b>21</b>
3.1 Aufgabe	21
3.2 Grundsätze	22
3.2.1 Programmierbarkeit	22
3.2.2 Lesbarkeit	23
3.2.3 Zuverlässigkeit	23
3.3 Konzepte	23
3.3.1 Vorgaben durch die Aufgabenstellung	23
3.3.2 Annahmen über die Hochsprache	24
3.3.3 Wesentliche neue Konzepte	24

<b>4 Die Basissprache</b>	<b>27</b>
4.1 Übersicht	27
4.2 Schemamodell	28
4.2.1 Attributtypen	28
4.2.2 Entitätstypen	30
4.2.3 Relationstypen	33
4.3 Regelmodell	34
4.3.1 Regeln	34
4.3.2 Benutzerkommandos	38
4.3.3 Benutzerdefinierte Funktionen	40
4.4 Situationsmodell (Startsituationsteil)	40
4.4.1 Erzeugung von Entitäten	41
4.4.2 Erzeugung von Relationen	41
4.4.3 Sonstige Initialisierungen	42
4.5 Modellausführung	42
4.5.1 Modellzeit	42
4.5.2 Ausführungsalgorithmus	43
4.5.3 Beispiel einer Modellausführung	45
4.6 Vergleich mit SESAM-1 und SESAM-Lite	48
4.6.1 Entitäts- und Relationstypen	48
4.6.2 Events	48
4.6.3 Regeln	49
4.6.4 Externe Namen von Entitäten	49
4.6.5 Modellzeit	49
<b>5 Realisierung der Basismaschine</b>	<b>50</b>
5.1 Entwurf	50
5.1.1 Interpreter-Ansatz	50
5.1.2 Code-Generator-Ansatz	52
5.1.3 Vergleich der Ansätze	57
5.1.4 Regelausführungsalgorithmus	58
5.2 Implementierung	61
5.2.1 Rahmenbedingungen	61
5.2.2 Umfang	61
5.2.3 Probleme	62
5.2.4 Bewertung	64
5.3 Beispiellauf	65
5.3.1 Übersetzung	65
5.3.2 Ausführung	65

<b>6 Zusammenfassung und Ausblick</b>	<b>69</b>
6.1 Zusammenfassung .....	69
6.1.1 Projektverlauf .....	69
6.1.2 Bewertung der Ergebnisse .....	74
6.2 Ausblick .....	75
6.2.1 Erweiterungen von BASE-2 .....	75
6.2.2 Erweiterungen der Basismaschine .....	77
6.2.3 Neuimplementierung .....	78
<b>A Glossar</b>	<b>79</b>
<b>B Beispielprogramm</b>	<b>88</b>
<b>C BASE-2-Grammatik</b>	<b>93</b>
<b>Literaturverzeichnis</b>	<b>96</b>

## Abbildungsverzeichnis

Abbildung 1.1:	SESAM und die Außenwelt . . . . .	1
Abbildung 2.1:	Erstellungsprozeß von Simulationsmodellen . . . . .	5
Abbildung 2.2:	Entstehung verbesserter geteilter mentaler Modelle durch Diskussion . . . . .	6
Abbildung 2.3:	Teile eines SESAM-Modells . . . . .	12
Abbildung 2.4:	Beispiel für ein Schemamodell . . . . .	12
Abbildung 2.5:	Beispiel für ein Situationsmodell . . . . .	13
Abbildung 2.6:	Graphische Veranschaulichung einer Regel . . . . .	14
Abbildung 2.7:	Vergleich Feuern und Aktivierung/Deaktivierung . . . . .	15
Abbildung 2.8:	Architektur des SESAM-1-Systems . . . . .	16
Abbildung 2.9:	Architektur des SESAM-2-Systems . . . . .	18
Abbildung 2.10:	Unterschiede zwischen Hochsprache und Basissprache . . . . .	19
Abbildung 3.1:	Schnittstellen der Basismaschine nach außen . . . . .	21
Abbildung 4.1:	Aufbau einer Modellbeschreibung in BASE-2 . . . . .	28
Abbildung 4.2:	Zusammenhang zwischen Vererbung und Typverträglichkeit . . . . .	32
Abbildung 4.3:	Mehrfachvererbung und Typverträglichkeit . . . . .	32
Abbildung 4.4:	Graphische Darstellung des Strukturteils . . . . .	35
Abbildung 4.5:	Graphische Darstellung des Ausführungsalgorithmus . . . . .	43
Abbildung 4.6:	Ausgangssituation des Beispiels . . . . .	46
Abbildung 4.7:	Situation nach der Einstellung . . . . .	46
Abbildung 4.8:	Regelinstanzen im ersten Simulationsschritt . . . . .	47
Abbildung 4.9:	Situation nach Ausführung von LetSpecify . . . . .	47
Abbildung 4.10:	Regelinstanzen im zweiten Simulationsschritt . . . . .	47
Abbildung 5.1:	Arbeitsweise des Interpreteransatzes . . . . .	51
Abbildung 5.2:	Architektur des Interpreters . . . . .	52
Abbildung 5.3:	Arbeitsweise des Code-Generator-Ansatzes . . . . .	53
Abbildung 5.4:	Architektur des Code-Generators . . . . .	54
Abbildung 5.5:	Architektur des ausführbaren Modells . . . . .	54
Abbildung 6.1:	Zeitplan . . . . .	70

# 1 Einführung

*A beginning is the time for taking the most delicate care that the balances are correct.  
(Frank Herbert, Dune)*

Diese Arbeit beschäftigt sich mit der Konzeption und der Realisierung einer Basismaschine für das SESAM-2-System. In diesem Kapitel wird das SESAM-Projekt kurz vorgestellt, um die Rahmenbedingungen und das Thema der Arbeit einordnen zu können. Darauf folgt eine kurze Beschreibung der Aufgabenstellung. Abschließend wird ein Überblick über den Aufbau der Arbeit gegeben.

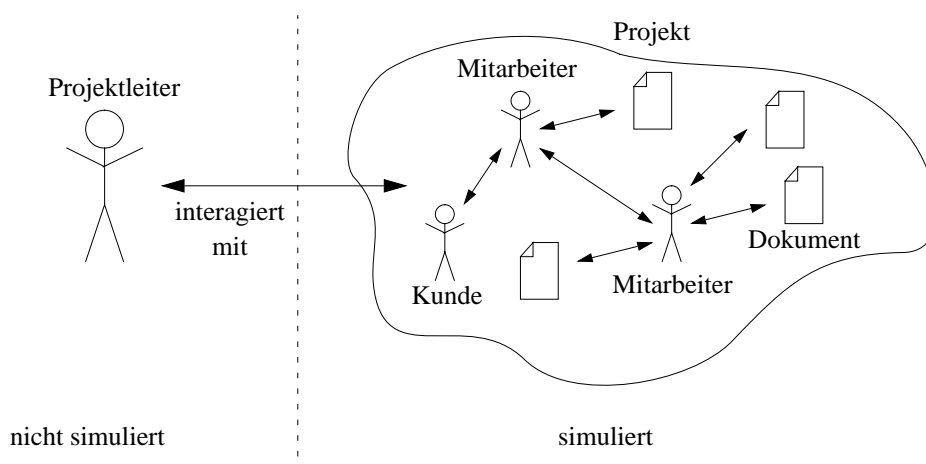
## 1.1 SESAM

### 1.1.1 Was ist SESAM?

SESAM ist ein Projekt der Abteilung Software Engineering am Institut für Informatik der Universität Stuttgart. SESAM ist eine Abkürzung für „Software Engineering Simulation by Animated Models“. Inhalt des SESAM-Projekts ist die *Simulation* von Software-Entwicklungsprojekten mit Hilfe rechnergestützter Modelle.

Ein SESAM-Modell umfaßt alle Merkmale eines Software-Entwicklungsprojekts – bis auf den Projektleiter. Die Rolle des Projektleiters übernimmt der Benutzer, indem er mit dem Modell interagiert. SESAM kann auch als ein Spiel aufgefaßt werden, bei dem der Spieler ein (simuliertes) Software-Entwicklungsprojekt leitet und dabei seine Fähigkeiten als Projektleiter unter Beweis stellen soll.

Abbildung 1.1 stellt die Abgrenzung des simulierten Projekts vom Projektleiter, der vom Benutzer gespielt wird, noch einmal dar.



**Abbildung 1.1: SESAM und die Außenwelt**

Im Projekt selbst interagieren simulierte Mitarbeiter und Kunden miteinander und arbeiten an Dokumenten. Dokument ist hier ein Oberbegriff für schriftlich und elektronisch vorliegende Unterlagen, die das Projekt betreffen. Das können zum Beispiel die Spezifikation oder der Programm-Code, aber auch Ausschreibungen für Software-Entwicklungswerkzeuge sein.

### 1.1.2 Ziele

Mit dem Projekt SESAM werden im wesentlichen zwei verschiedene Ziele verfolgt:

- Der Einsatz als Lehrmittel und
- der Gewinn eines besseren Verständnisses der Gesetzmäßigkeiten innerhalb des Ablaufs eines Software-Entwicklungsprojekts.

Beim Einsatz des SESAM-Systems als Lehrmittel lernt der Spieler durch das Spiel, welche Fehler man bei der Leitung eines Software-Entwicklungsprojekts machen kann und wie sich diese Fehler vermeiden lassen.

Um ein besseres Verständnis der Gesetzmäßigkeiten eines Software-Entwicklungsprojekts zu gewinnen, wird ein solches Projekt in ein Simulationsmodell umgesetzt. Grundlage der Modellierung sind Daten eines bereits in Wirklichkeit durchgeführten Projekts. Das Projekt-Modell wird simuliert und so lange angepaßt, bis die Eigenschaften des Modells die des echten Projekts widerspiegeln. Während des Modellierungsprozesses gewinnt man Erkenntnisse über mögliche Wirkungsbeziehungen innerhalb des Projekts. Zum Beispiel könnte eine unvollständige Anforderungsspezifikation den Zeitverzug am Ende des Projekts vergrößert haben, da der Kunde immer wieder Nachbesserungen gefordert hat.

SESAM stellt Modellierungswerkzeuge und einen Modellsimulator zur Verfügung. Auf diese Weise können beliebig viele Modelle von Software-Entwicklungsprojekten erstellt und ausgeführt werden.

In Kapitel 2 wird genauer auf die Hintergründe und Ziele von SESAM sowie auf die inhaltlichen Konzepte eingegangen. Als einführende Literatur zu SESAM sei auf Ludewig (1994) und den Text „Overview of SESAM“, der Bestandteil der Online-Dokumentation von SESAM-1 ist, verwiesen.

### 1.1.3 Implementierungen

Das SESAM-System wurde vollständig in Smalltalk-80 realisiert. Dabei wurden nacheinander drei Prototypen und darauf aufbauend das Pilotsystem SESAM-1 implementiert. SESAM-1 wurde durch einen evolutionären Entwicklungsprozeß in die zur Zeit verwendete SESAM-Version 1.2 überführt. Einen wichtigen Anteil an der Entwicklung des Pilotsystems hatte Kurt Schneider, dessen Dissertation (Schneider, 1994) die Konzepte und die Realisierung von SESAM-1 ausführlich darstellt.

Probleme mit der Wartbarkeit des recht monolithischen SESAM-1-Systems, die auch auf fehlende Dokumentation zurückzuführen ist, sowie der Wunsch nach neuen Konzepten in der Modellbeschreibungssprache führten dazu, daß eine Neuimplementierung des SESAM-Systems in Ada 95 beschlossen wurde. Das neue System soll den Namen SESAM-2 tragen.

## 1.2 Aufgabenstellung

SESAM-2 wird im wesentlichen aus zwei Komponenten bestehen: Werkzeugen zur Modellerstellung und Werkzeugen zur Modellausführung. Die *Basismaschine* ist ein Werkzeug zur Modellausführung. Sie realisiert einen rudimentären Ausführungsmechanismus für SESAM-Modelle, die in einer speziellen Modellbeschreibungssprache beschrieben sind.

Schwerpunkt dieser Arbeit ist die Konzeption der Basismaschine. Dabei steht der Entwurf der Modellbeschreibungssprache der Basismaschine, der sogenannten Basissprache, im Vordergrund. Nach der Konzeption ist eine formale Spezifikation anzufertigen, die die Syntax und die Semantik der Basissprache präzise faßt. Anhand einer prototypischen Implementierung sind die Realisierbarkeit und die Brauchbarkeit der entwickelten Konzepte nachzuweisen oder zu widerlegen. Aus den so gewonnenen Erkenntnissen können dann Handlungsempfehlungen darüber abgeleitet werden, welche Konzepte endgültig übernommen werden können und welche noch der Überarbeitung bedürfen.

Die Implementierung soll trotz ihres prototypischen Charakters möglichst viele Bestandteile haben, die in einer endgültigen Implementierung der Basismaschine wiederverwendet werden können. Daher wird großer Wert auf den Entwurf geeigneter abstrakter Datentypen und eine sinnvolle Modularisierung gelegt. Auf diese Weise bedarf es nur noch der Neuimplementierung einiger Module, um zu einer effizient arbeitenden Realisierung der Basismaschine zu gelangen.

## 1.3 Aufbau der Arbeit

In Kapitel 2 wird auf das SESAM-Projekt genauer eingegangen. Kapitel 3 zeigt die Grundsätze der Konzeption der Basismaschine und die Konzepte selbst. Die während der Konzeption entworfene Basissprache BASE-2 wird in Kapitel 4 vorgestellt. Kapitel 5 befaßt sich mit der Realisierung der Basismaschine als experimenteller Prototyp und den dabei gemachten Erfahrungen. Kapitel 6 schließlich gibt einen Überblick über den Ablauf der Arbeit und die dabei gewonnenen Ergebnisse. Daran schließen sich in Form eines Ausblicks Gedanken über neue und überarbeitete Konzepte der Basismaschine an. Im Anhang schließlich finden sich ein Glossar, ein Beispielprogramm in BASE-2 und die Grammatik von BASE-2 in Erweiterter Backus-Naur-Form (EBNF).

## 2 SESAM

*The tools we use have a profound (and devious) influence on our thinking habits, and, therefore, on our thinking abilities.  
(Edsger W. Dijkstra)*

SESAM-Modelle sind eine spezielle Art von Lernmodellen. Was ein Lernmodell ist und welche Zielsetzungen von Lernmodellen in SESAM verwirklicht werden, das beschreibt dieses Kapitel. Darüber hinaus wird noch auf die Modellierungskonzepte in SESAM eingegangen. Außerdem werden die Architekturen der SESAM-Systeme SESAM-1 und SESAM-2 vorgestellt.

### 2.1 Lernmodelle

*Heute erfährt jeder weit mehr als er versteht.  
Dennoch ist es eher die Erfahrung als das Verstehen, was das Verhalten beeinflusst.  
(Marshall McLuhan)*

#### 2.1.1 Definition

Ein Lernmodell ist zunächst einmal ein Modell, an dem man etwas lernen kann. Hier soll der Begriff Lernmodell jedoch in einem etwas eingeschränkteren Sinne gebraucht werden. Ein *Lernmodell* ist hier ein Simulationsmodell mit Benutzungsoberflächen, das auf einem Rechner simuliert wird. Es dient der spielerischen Erfahrung eines Ausschnitts der Realität, der so komplex ist, daß er nicht sofort vollständig erfaßt und verstanden werden kann.

Ein bekanntes Beispiel für rechnergestützte Lernmodelle sind die sogenannten „Management Flight Simulators“ (vgl. z. B. Senge, 1990), bei denen der Spieler die Rolle eines Managers übernimmt und strategische und operative Entscheidungen für sein Unternehmen fällen muß. Je nach „Weisheit“ der Entscheidungen nimmt das Unternehmen einen rasenden Aufschwung, kümmert vor sich hin oder bricht zusammen. Doch im Gegensatz zur Realität ist ein Zusammenbruch nicht schlimm. Man kann sogar das Spiel noch einmal von vorne beginnen und diesmal erkannte Fehler vermeiden.

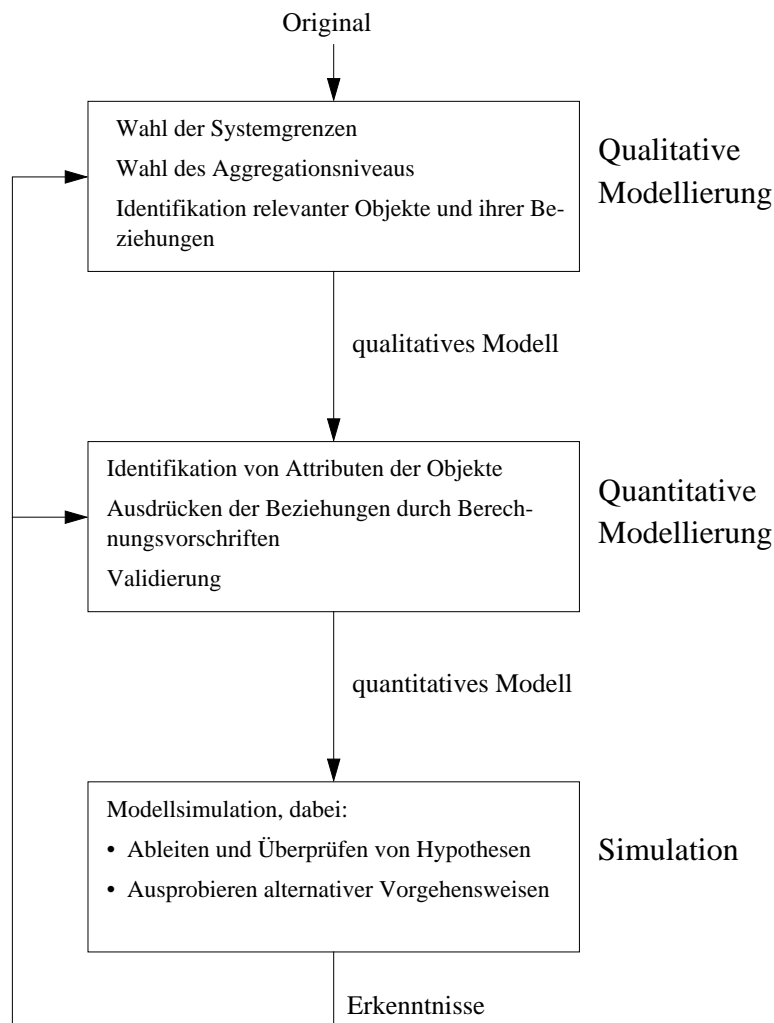
#### 2.1.2 Ziele

Das Lernen an einem Lernmodell geschieht im wesentlichen in zwei Bereichen. Das sind die Bereiche der Modellerstellung und der Modellverwendung. Modellverwendung ist hier das Spielen mit dem Modell.

##### 2.1.2.1 Lernen durch Modellerstellung

Bei der Modellerstellung arbeiten meistens mehrere Personen, im folgenden Modellierer genannt, an der Erstellung eines Simulationsmodells für den betrachteten Realitätsschnitt. Der Realitätsschnitt wird dabei als Original bezeichnet. Die beschriebene Vorgehensweise orientiert sich an der Darstellung von Senge (1990), es wird jedoch vom spezifischen Ansatz der Modellimplementierung dort (System Dynamics) abstrahiert.

Abbildung 2.1 faßt den Prozeß der Modellerstellung graphisch zusammen. Die Modellierungsschritte werden durch Rechtecke dargestellt. Die Pfeile kennzeichnen Informationsströme.



**Abbildung 2.1: Erstellungsprozeß von Simulationsmodellen**

Ausgehend vom *Original* erzeugt die qualitative Modellierung ein *qualitatives Modell*, das im nächsten Schritt, der quantitativen Modellierung formalisiert und quantifiziert wird, wodurch ein *quantitatives Modell* entsteht. Das quantitative Modell wird simuliert. Die dabei gewonnenen *Erkenntnisse* fließen in die vorhergehenden Modellierungsschritte ein, indem an dem Modell Verbesserungen in Hinblick auf eine bessere Abbildung des Originals vorgenommen werden. Es kommt also zu einer Rückkopplung der Erkenntnisse aus der Modellsimulation auf den gesamten Modellierungsprozeß.

Im folgenden werden die drei Modellierungsschritte genauer ausgeführt und die Begriffe *qualitatives* und *quantitatives Modell* erläutert.

**Qualitative Modellierung.** Zunächst wird der betrachtete Ausschnitt der Realität, das *Original*, von seiner Umgebung abgegrenzt (Wahl der Systemgrenzen). Die Schnittstellen des Systems nach außen werden definiert. Danach wird das gewünschte Aggregationsniveau des Modells

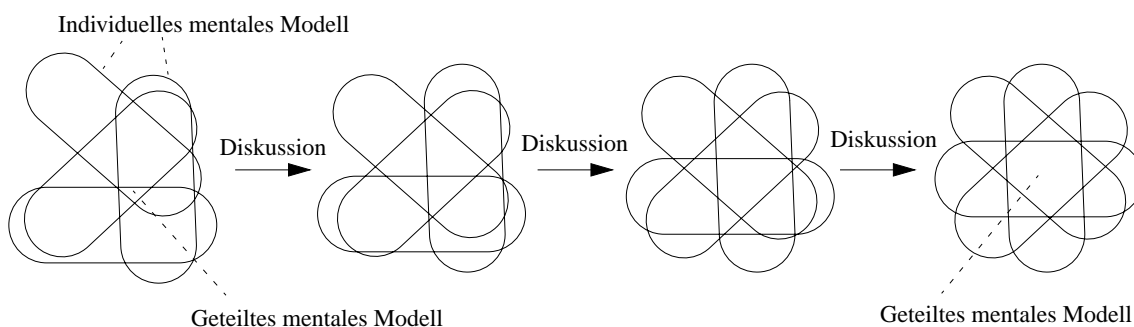
festgelegt. Das Aggregationsniveau bestimmt, wie detailliert das Original modelliert wird. Bei einem hohen Aggregationsniveau werden viele Bestandteile des Originals zu einem Modellbestandteil zusammengefaßt (aggregiert).

In dem abgegrenzten System werden anschließend sukzessive Bestandteile und ihre Beziehungen identifiziert. Gegebenenfalls werden diese Bestandteile weiter verfeinert, bis das gewünschte Aggregationsniveau erreicht ist. Auf diese Weise erhält man das *qualitative Modell*. In diesem sind Aussagen über Zusammenhänge und Wirkungsbeziehungen zwischen allen Bestandteilen möglich. Diese Aussagen haben jedoch nur einen qualitativen Charakter, eine Präzisierung durch Zahlen ist nicht möglich.

In diesen Modellierungsschritt, bei dem viele Eigenschaften des Originals verloren gehen, fließen die Vorstellungen der einzelnen Modellierer über den Aufbau des Originals ein. Diese Vorstellungen werden auch als *individuelles mentales Modell* (Kiesler, Sproull, 1982) bezeichnet. Individuelle mentale Modelle werden von Kiesler und Sproull definiert als vereinfachte, erfahrungsbasierte Repräsentationen der Wirklichkeit, die jedem Verstehen und Entscheiden zugrunde liegen. Solche Modelle können durchaus Widersprüche enthalten oder an der Wirklichkeit vorbeigehen.

Um seine Vorstellungen den anderen Modellierern mitteilen zu können, muß jeder Modellierer seine zunächst recht intuitiven Vorstellungen strukturieren und formulieren. Durch den Diskussionsprozeß in der Gruppe werden die individuellen mentalen Modelle überprüft und, falls notwendig, auch verändert. Es entsteht in einem evolutionären Prozeß eine gemeinsame Vorstellung, ein sogenanntes *geteiltes mentales Modell* (Kim, 1993 und Strata, 1989). Das geteilte mentale Modell kann als eine Art Schnittmenge der individuellen mentalen Modelle der Gruppenmitglieder aufgefaßt werden, die sich im Laufe des Modellentwicklungsprozesses aneinander angeglichen haben.

Abbildung 2.2 (nach Langfield-Smith, 1992, S. 362) veranschaulicht diesen Vorgang graphisch. Die individuellen mentalen Modelle von vier Modellierern sind als Flächen dargestellt. Wo sich alle Flächen überlappen, liegen die gemeinsamen Vorstellungen, das geteilte mentale Modell. Im Laufe der Modellierung gleichen sich die individuellen mentalen Modelle aneinander an, so daß das geteilte mentale Modell umfassender wird.



**Abbildung 2.2: Entstehung verbesserter geteilter mentaler Modelle durch Diskussion**

**Quantitative Modellierung.** Das qualitative Modell, das zunächst nur qualitative Aussagen über die Systembestandteile und ihre Beziehungen untereinander enthält, muß nun formalisiert und quantifiziert werden, um es in eine dem Rechner verständliche Form zu bringen. Aus dem qualitativen Modell erhält man so ein quantitatives Modell. Das quantitative Modell kann anschließend mit Hilfe des Rechners simuliert werden.

Zur Formalisierung und Quantifizierung gibt es verschiedene Ansätze. Beispielhaft soll kurz der bei System Dynamics (Forrester, 1971 und 1974) verwendete Ansatz vorgestellt werden. System Dynamics verfolgt einen diskreten Modellierungsansatz. Ein System-Dynamics-Modell besteht aus Zustandsgrößen (sogenannten Levels), die den Modellzustand bilden, und Flußgrößen (sogenannten Raten), die die Veränderung des Modellzustands über die Zeit beschreiben. Bei der Simulation wird in jedem Simulationsschritt aus dem momentanen Modellzustand der nächste Zustand berechnet. Zur Herstellung eines quantitativen Modells aus einem qualitativen Modell werden die Bestandteile des Modells in Zustandsgrößen, die Beziehungen zwischen den Bestandteilen in Flußgrößen umgesetzt.

**Simulation.** Durch die Simulation können Hypothesen über das Verhalten des Originals in bestimmten Situationen am Modell überprüft werden. Dazu werden beispielhafte Szenarien am Rechner durchgespielt. Um die Anwendbarkeit der Simulationsergebnisse auf das Original glaubhaft zu machen, bedarf es jedoch vorher der Validierung des Modells. Dabei werden Situationen, für die das Verhalten des Originals bekannt ist, am Modell nachvollzogen und das Originalverhalten mit dem Modellverhalten verglichen. Weicht das Modellverhalten ab, liegt ein Modellierungsfehler vor und der Modellentwicklungsprozeß muß erneut durchlaufen werden. Die aus der Simulation gewonnenen Erkenntnisse fließen dabei ein.

Nach einigen Iterationen dieses Prozesses liegt in den meisten Fällen ein brauchbares Modell vor. Die Rückübertragbarkeit von Erkenntnissen aus der Modellsimulation auf das Original kann jedoch nie bewiesen werden. Es kann höchstens durch eine intensive Validierung das Vertrauen in das Modell und die daraus abgeleiteten Aussagen gestärkt werden.

Einen sehr ausführlichen Überblick über die Eigenschaften, die Erstellung und die Verwendung von Simulationsmodellen gibt Bossel (1994). Der von ihm verwendete Modellierungsansatz stützt sich auf System Dynamics.

### 2.1.2.2 Lernen durch Modellverwendung

Das bei der Modellerstellung entstandene Modell kann dazu eingesetzt werden, Personen, die nicht am Modellentwicklungsprozeß beteiligt waren, die Erkenntnisse der Modellierungsgruppe zugänglich zu machen. Zu diesem Zweck macht man aus dem Simulationsmodell ein Lernmodell. Man versieht es mit einer geeigneten Benutzungsoberfläche, durch die ein Benutzer das Modell simulieren kann. Dabei hat er die Möglichkeit, in den Verlauf der Simulation einzugreifen. Er darf zu bestimmten Zeitpunkten der Simulation Entscheidungen vornehmen. Bei einem simulierten Unternehmen wären solche Entscheidungen zum Beispiel eine Senkung der Verkaufspreise der Produkte oder die Ausweitung der Produktionskapazität durch die Einstellung zusätzlichen Personals.

Die Folgen seiner Entscheidungen werden dem Benutzer im Verlauf der weiteren Simulation – mehr oder weniger – deutlich. Eine Senkung der Verkaufspreise kann zum Beispiel dazu führen, daß der Absatz der Produkte zunimmt. Der erzielte Umsatz reicht jedoch nicht aus, alle für

die Produktion anfallenden Kosten zu decken, so daß das Unternehmen trotz seines Markterfolgs schließlich Konkurs anmelden muß. Ein derartiger Fehler wird dem Benutzer sicher nicht ein weiteres Mal unterlaufen – sofern er die Ursachen des Mißerfolgs erkannt hat. Bei der Identifikation der Fehlerursachen kann ihm das Modell nicht helfen, da es kein Verständnis dessen besitzt, was es tut: es rechnet lediglich.

Der Spieler lernt dazu, indem er – ausgelöst durch unerwünschte Ergebnisse – Wirkungszusammenhänge innerhalb des Modells zu ergründen sucht und begreift. In Zukunft wird er seine Erkenntnisse in seine Entscheidungen einbeziehen und bessere Ergebnisse erzielen. Ein Spiel kann mit denselben Ausgangsbedingungen wiederholt werden, was in der Realität häufig nicht möglich ist: Ist das Unternehmen im Konkurs, kann man die Zeit nicht zurückdrehen und noch einmal von vorne anfangen. Weil in jedem Versuch dieselben Ausgangsbedingungen herrschen, wird ein besseres Ergebnis, das sich aus der Anwendung einer geeigneteren Herangehensweise ergibt, besonders deutlich.

Die Benutzung eines Lernmodells weist viele Aspekte eines Spiels auf. Man macht sich auf diese Weise den angeborenen Spieltrieb des Menschen zunutze. Das Erzielen eines möglichst guten Ergebnisses im Spiel stellt einen Anreiz dar, sich mit den zugrunde liegenden Regeln des Spiels (hier die Wirkungsbeziehungen innerhalb des Modells) auseinanderzusetzen und sie zu seinem Vorteil einzusetzen. Denn nur, wer die Regeln kennt und sie zu seinem Vorteil nutzt, kann dauerhaft in einem Spiel erfolgreich sein.

Das Spiel als Mittel zur Schulung der körperlichen und geistigen Fähigkeiten ist auch bei anderen höheren Tieren, insbesondere im Kindesalter, weit verbreitet, so daß man daraus schließen kann, daß es sich beim Spiel um eine besonders effiziente Methode des Trainings von Körper und Geist handelt.

### **2.1.3 Chancen**

Die Anwendung von Lernmodellen zum Sammeln von Erfahrungen in dem modellierten Realitätsausschnitt hat zwei wesentliche Vorteile. Zum einen ist ein gefahrloses Training möglich. Fehler, die in Wirklichkeit den Untergang des Unternehmens bedeuten würden, sind hier im Endeffekt sogar erwünscht. Denn gerade die Fehler initiieren Lernprozesse in besonderem Maße. Die Frage „Was habe ich bloß falsch gemacht?“ führt zu Erkenntnissen, die bei bloßer Vermittlung durch einen Lehrer kaum einen so tiefen Eindruck hinterlassen hätten. Fehler, die beim Spielen gemacht werden, werden in der Realität nicht mehr unterlaufen.

Zum zweiten ist es in einem Lernmodell möglich, Zeit und Raum zu komprimieren. Es ist nicht erforderlich, das Unternehmen in Echtzeit zu simulieren, sondern es können innerhalb von wenigen Stunden viele Monate der wirklichen Zeit simuliert werden. Dies ermöglicht die Erprobung langfristig angelegter Strategien am Modell. Außerdem ist das Ausprobieren von alternativen Handlungsmöglichkeiten in mehreren Simulationsläufen möglich, und damit die Optimierung der Unternehmensführung. Dagegen ist es nicht erforderlich, im Unternehmen umherzugehen und sich seine Informationen zusammenzusuchen. Die Informationen sind alle an einer Stelle, der Benutzungsoberfläche des Lernmodells, zusammengefaßt.

### 2.1.4 Risiken

Die Verwendung von Simulationsmodellen hat jedoch auch ihre Risiken. Eines davon ist das oft blinde Vertrauen in die Prognosefähigkeit des Modells. Was das Modell vorhersagt, so eine weitverbreitete Auffassung, wird auch eintreten. Oftmals wird übersehen, daß das Modell lediglich eine unvollkommene Näherung der Realität darstellt und nur eingeschränkte Aussagen zuläßt. Ein Simulationsmodell kann als Prognoseinstrument immer nur die Entscheidungsfindung unterstützen, darf jedoch nie die einzige Grundlage einer Entscheidung sein.

Ein weiteres Risiko beim Einsatz von Lernmodellen ist, daß der Aspekt des Erzielens guter Ergebnisse vom Spieler zum alleinigen Spielzweck wird. Er spielt mit dem Modell herum, ohne die Ergebnisse seines Tuns einer kritischen Reflexion zu unterziehen. Stattdessen wird durch Herumprobieren eine Vorgehensweise identifiziert, mit der sich im Spiel optimale Ergebnisse erzielen lassen. Dieses auch als Video-Gaming-Syndrom (Senge, Sterman, 1992) bezeichnete Phänomen läßt sich umgehen, indem das Lernmodell in einen größeren didaktischen Zusammenhang integriert wird. In einem sogenannten *Lernlabor* (Sterman, 1994) wird die Auseinandersetzung mit dem Lernmodell vorbereitet, begleitet und nachbereitet. Die Spieler tauschen in Zwischen- und Abschlußbesprechungen ihre gewonnenen Erkenntnisse untereinander aus. Der Tutor stellt das dem Lernmodell zugrundeliegende qualitative Modell mit seinen Wirkungsbeziehungen vor. Durch die beim Spielen gemachten Erfahrungen und eine eigene gedankliche Modellbildung ist das Verständnis der Spieler für das qualitative Modell um einiges höher als wenn ihnen lediglich das Modell selbst präsentiert worden wäre.

Problematisch sind in diesem Zusammenhang noch folgende Aspekte: Die Realität ändert sich fortlaufend, so daß eigentlich das hinter dem Lernmodell stehende qualitative und quantitative Modell ständig angepaßt werden müßte. Diesen Aspekt kann man über kurzfristige Zeiträume jedoch meistens vernachlässigen. Viel verheerender wirken sich Denkfehler und falsche Zusammenhänge im Lernmodell aus. Durch die intensive Art des Lernens werden diese falschen Strukturen ebenso tief verankert wie die korrekten; ein nachträgliches Umlernen erfordert einen Aufwand.

## 2.2 SESAM

Im folgenden wird das SESAM-System als spezifisches System zur Realisierung von Lernmodellen betrachtet. Zunächst werden die Ideen hinter SESAM vorgestellt. Abschnitt 2.3 führt die Konzepte der Modellierung in SESAM ein. Danach wird in Abschnitt 2.4 auf einzelne Implementierungen des SESAM-Systems eingegangen.

### 2.2.1 Vision und Ziele

SESAM-Modelle sind Lernmodelle für den Bereich der Software-Entwicklungsprojekte, bei denen aus Anforderungen eines Kunden ein Programm spezifiziert und implementiert werden soll. Das Ziel des Software-Projektmanagements ist es, das Software-Projekt erfolgreich abzuschließen. Erfolgreich heißt nach Frühauf, Ludewig und Sandmayr (1988), daß alle Anforderungen des Kunden durch die entstandene Software angemessen realisiert werden (Aspekte der Produktqualität) sowie das geplante Budget und der vom Kunden geforderte Abschlußtermin

des Projekts eingehalten werden (Aspekte der Prozeßqualität). Mit SESAM soll der Benutzer in die Lage versetzt werden, mehr über die Zusammenhänge in Software-Projekten herauszufinden, um diese Erkenntnisse für ein erfolgreicherer Projektmanagement einsetzen zu können.

### 2.2.1.1 Lernen durch Modellerstellung

Auch im Projekt SESAM werden die beiden wesentlichen Zielbereiche von Lernmodellen verfolgt. Beim Lernen durch Modellerstellung will man vorhandene, eher qualitativ formulierte Hypothesen<sup>1</sup> über Zusammenhänge innerhalb eines Software-Projekts quantifizieren und durch die anschließende Simulation überprüfen. Auch neue Hypothesen können in die Modelle integriert und ihre Auswirkungen im Gesamtzusammenhang untersucht werden. Das SESAM-System wird so als Forschungswerkzeug verwendet.

Ein bekanntes Beispiel für eine Hypothese im Bereich des Software-Projektmanagements ist das sogenannte Brookssche Gesetz (*Brooks' Law*, Brooks, 1982). Dieses Gesetz besagt, daß die Einstellung zusätzlichen Personals in ein bereits verspätetes Projekt zu weiteren Verspätungen führen wird. Begründen läßt sich dieses Phänomen damit, daß das neue Personal durch das vorhandene erst einmal eingearbeitet werden muß, was anfangs zu einem allgemeinen Produktivitätsrückgang führt. Hinzu kommt der größere Kommunikations- und Organisationsaufwand bei einer erhöhten Anzahl von Projektmitarbeitern.

Da es sich bei der Erstellung des SESAM-Systems ebenfalls um ein Software-Projekt handelt, fließen Erkenntnisse aus den Modellsimulationen mit SESAM in den Entwicklungsprozeß von SESAM ein. Umgekehrt gehen natürlich auch Erfahrungen mit der Entwicklung von SESAM in die Modellierung ein.

### 2.2.1.2 Lernen durch Modellverwendung

Das Lernen durch Modellverwendung wird im Bereich der Lehre genutzt. SESAM-Modelle werden Studenten im Rahmen des Fachpraktikums „Projektmanagement“ zugänglich gemacht. Die Studenten übernehmen in einem simulierten Software-Entwicklungsprojekt die Rolle des Projektleiters und sind für den Erfolg des Projekts verantwortlich. Wie ein Projektleiter wird ein Spieler mit den Problemen der Informationsbeschaffung und des Zeitdrucks konfrontiert. Informationen über den Projektstand fließen spärlich, meistens nur auf Anfrage. Auskünfte der Mitarbeiter sind mit Vorsicht zu genießen, da natürlich deren Ansichten und Interessen den Informationsgehalt verfälschen können. Zudem kostet jede Aktion den Projektleiter wertvolle Zeit, die er nicht mehr für andere Dinge verwenden kann.

Analog dem Lernlaborkonzept werden die Spiele der Studenten vom Betreuer des Praktikums vorbereitet, begleitet und zusammen mit den Studenten ausgewertet. Um ein überhastetes Vorgehen zu erschweren, wird der Zugang zu dem Modell auf wenige Stunden pro Woche beschränkt. Außerdem wird das schriftliche Niederlegen von Planungsüberlegungen verlangt. An ein abgeschlossenes und analysiertes Spiel schließt sich ein erneuter Spielzyklus an, bei dem die gemachten Erfahrungen umgesetzt werden können. Der Lernvorgang während des ersten Spiels wird dann durch bessere Ergebnisse belohnt. Der Spieler hat so ein besonderes Erfolgserlebnis.

---

1. von Ludewig (1994) auch als „Bauernregeln“ bezeichnet

Der Einsatz in der Lehre ist natürlich nicht der einzige Verwendungszweck von SESAM. Im Rahmen einer Kooperation mit der Industrie soll SESAM auch zur Schulung von Projektleitern und -mitarbeitern eingesetzt werden.

### 2.2.2 Grenzen

SESAM-Modelle erheben nicht den Anspruch, alle Aspekte des Software-Projektmanagements abzudecken. Es wäre theoretisch denkbar, ein Lernmodell zu schaffen, das ein Software-Projekt in einer Virtual-Reality-Umgebung simuliert. Der Spieler interagiert in einer künstlichen Welt mit künstlichen, aber fast echt aussehenden Menschen. Er kann sich innerhalb der simulierten Firma frei bewegen, um Mitarbeiter zu besuchen und Dokumente zu betrachten. Auf diese Weise wären zum Beispiel auch die subtilen Signale der Körpersprache der Mitarbeiter dem Projektleiter zugänglich, und er könnte darauf reagieren. Diesem Ideal stehen nicht nur die heute noch unüberwindlichen Realisierungsschwierigkeiten entgegen, es würde auch der Vorteil der Komprimierung von Zeit und Raum verloren gehen. Deshalb beschränken sich SESAM-Modelle von vornherein auf die eher organisatorischen Aspekte des Projektmanagements. Die Benutzerinteraktion geschieht auf textueller Ebene, die Abstraktionsebene des Geschehens ist recht hoch.

Durch die hohe Abstraktion fallen viele soziale Aspekte weg. Das heute so wichtige Teammanagement kann nur am Rande erlebt werden. Die hierarchische Organisationsstruktur<sup>1</sup> ist heutzutage bei kleineren Projekten (unter etwa 10 Mitgliedern) eher ungewöhnlich, in der Regel arbeitet der Projektleiter im Projekt mit.

Die Mitarbeiter treten nur durch gelegentliche Äußerungen in Textform in Erscheinung, so daß keine direkte sensorische Information über zum Beispiel ihre Körpersprache oder ihren Tonfall vorliegt. Das ist gerade so, als wolle man ein Software-Projekt ausschließlich durch das Verschicken und Empfangen von (elektronischer) Post führen. Die für einen Manager so wichtige soziale Kompetenz im Umgang mit Mitarbeitern ist so kaum erlernbar. Der Schwerpunkt von SESAM liegt deshalb eindeutig im Bereich der Planung und der Organisation von Software-Entwicklungsprojekten. Mehr kann und darf man von SESAM nicht erwarten.

## 2.3 SESAM-Modelle

In diesem Abschnitt werden diejenigen Modellierungskonzepte von SESAM zusammengefaßt, die in allen Implementierungen (vgl. hierzu Abschnitt 2.4) vorhanden waren und somit den konzeptionellen Kern des SESAM-Modellierungsansatzes darstellen.

SESAM-Modelle sind diskrete Modelle. Diskrete Modelle besitzen einen Modellzustand, der sich aus einer Menge von Zustandsgrößen zusammensetzt, und einer Menge von Berechnungsvorschriften, mit denen man aus dem aktuellen Modellzustand den Folgezustand gewinnt. Jeder Zustand wird mit einem Zeitpunkt assoziiert. Die so definierte Modellzeit wird bei einem *Simulationsschritt* (der Berechnung des Folgezustands) um die Simulationsschrittweite erhöht.

---

1. der Projektleiter arbeitet nicht selbst im Projekt mit, sondern leitet und organisiert nur.

### 2.3.1 Aufbau

Ein fertiges SESAM-Modell, ein sogenanntes *dynamisches Modell*, besteht aus drei Teilmodellen. Dies sind das Schemamodell, das Situationsmodell und das Regelmodell. Abbildung 2.3 (nach Schneider, 1994, S. 23) zeigt die Beziehungen zwischen den drei Teilmodellen.

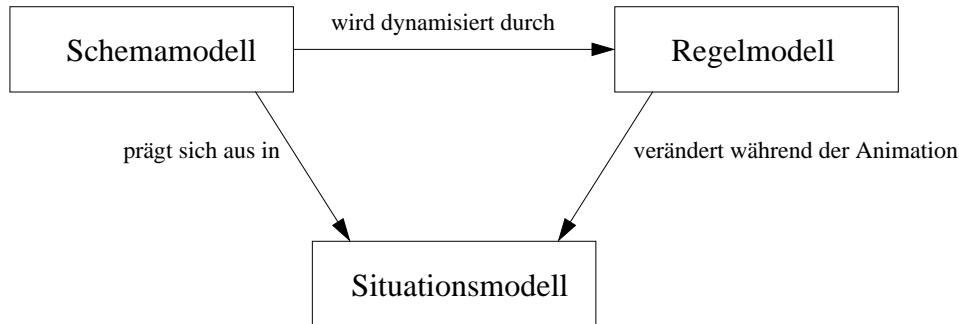


Abbildung 2.3: Teile eines SESAM-Modells

#### 2.3.1.1 Schemamodell

Das *Schemamodell* beschreibt, ähnlich einem Entity-Relationship-Modell, die abstrakte Welt, in der das Modell abläuft. Die Objekte dieser Welt werden durch Klassen, sogenannte *Entitätstypen*, beschrieben. Die Entitätstypen haben zur Beschreibung ihrer Eigenschaften *Attribute*. Die Beziehungen zwischen Entitätstypen werden durch Beziehungsklassen, sogenannten *Relationstypen*, repräsentiert. Auch die Relationstypen können Attribute besitzen.

Dies soll an einem kleinen Beispiel illustriert werden. Abbildung 2.4 zeigt eine graphische Repräsentation eines Schemamodells. Die verwendete Notation orientiert sich an der des Entity-Relationship-Modells. Entitätstypen werden durch Rechtecke, Relationstypen durch Rauten mit Verbindungskanten dargestellt. Die mit Linien an den zugehörigen Typ gebundene Kreise sind die Attribute.

Im Beispiel werden zwei Entitätstypen *Person* und *Document* durch einen Relationstyp *WorksOn* verbunden. *Person* hat die Attribute *name* und *age*, *Document* die Attribute *name* und *version*. Der Relationstyp *WorksOn* besitzt das Attribut *intensity*.

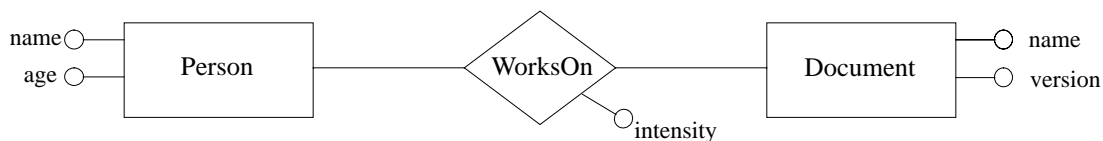


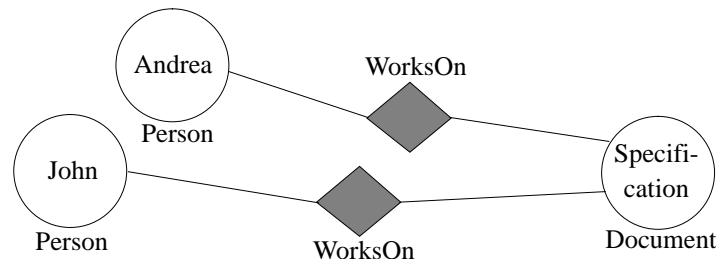
Abbildung 2.4: Beispiel für ein Schemamodell

#### 2.3.1.2 Situationsmodell

Das *Situationsmodell* repräsentiert den eigentlichen Modellzustand, auf den das Regelmodell angewandt wird. Es enthält Instanzen der im Schemamodell eingeführten Entitäts- und Relationstypen. Damit kann ein Situationsmodell als eine Instanziierung des Schemamodells aufgefaßt werden. Zusätzlich umfaßt das Situationsmodell auch die Modellzeit.

Das Situationsmodell, das den Anfangszustand eines dynamischen Modells beschreibt, wird auch als *Startsituation* bezeichnet. Die Instanzen von Entitätstypen heißen *Entitäten*, die Instanzen von Relationstypen *Relationen*.

Ein Beispiel für ein Situationsmodell zum Schemamodell in Abbildung 2.4 zeigt Abbildung 2.5. Zwei Personen, Andrea und John, arbeiten an einem Dokument namens *Specification*. Auf die Angabe der Attribute und ihrer Werte wurde aus Gründen der Übersichtlichkeit verzichtet. Entitäten werden durch Kreise, Relationen durch ausgefüllte Rauten dargestellt.



**Abbildung 2.5: Beispiel für ein Situationsmodell**

### 2.3.1.3 Regelmodell

Das *Regelmodell* beschreibt die Dynamik des Modells. SESAM-Modelle sind regelbasiert. Dies bedeutet, daß Veränderungen am Modellzustand durch Regeln durchgeführt werden. Die Regeln erwarten einen bestimmten Zustand des Modells und werden ausgeführt, wenn dieser Zustand gefunden werden kann. Bei der Ausführung nimmt die Regel am Modellzustand Veränderungen vor.

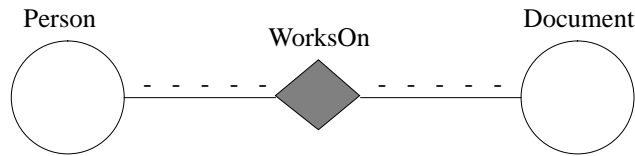
Der gewünschte Zustand wird im *Bedingungsteil* der Regel festgelegt. Da der Modellzustand aus Entitäten und Relationen besteht, kann der gewünschte Zustand durch eine Struktur aus einer Menge von Entitäten und einer Menge von Relationen zwischen diesen Entitäten beschrieben werden. Über dieser Struktur können dann noch Bedingungen zu den Attributen formuliert werden.

Der *Aktionsteil* der Regel enthält die Veränderungen des Modellzustands, die bei erfülltem Bedingungsteil durchgeführt werden sollen. Der Aktionsteil kann folgende Aktionen enthalten:

- Attributwerte verändern,
- neue Entitäten erzeugen,
- neue Relationen erzeugen, d.h. neue Verbindungen einfügen,
- vorhandene Entitäten entfernen (einschließlich aller Verbindungen) oder
- vorhandene Relationen entfernen, d.h. Verbindungen löschen.

Als Beispiel betrachte man eine zum eingeführten Schemamodell passende Regel, die alle *worksOn*-Beziehungen entfernt. Die geforderte Struktur sowie die auf der Struktur durchzuführenden Änderungen lassen sich graphisch wie in Abbildung 2.6 veranschaulichen. Die graphische Notation ist auch hier an Schneider (1994) angelehnt.

Benötigt wird eine *Person* und ein *Document* sowie eine *worksOn*-Beziehung zwischen den beiden. Die Minus-Zeichen deuten an, daß die Kante im Verlauf der Regelausführung gelöscht werden soll.



**Abbildung 2.6: Graphische Veranschaulichung einer Regel**

Kurt Schneider (1994) hat gezeigt, daß sich das Situationsmodell als Hypergraph<sup>1</sup> interpretieren läßt, auf den die Regeln in Form von Graphgrammatikproduktionen<sup>2</sup> angewendet werden. Der Hypergraph hat als Knoten die Entitäten, als Kanten die Relationen. Diese alternative Darstellungsweise wird im folgenden nicht weiter vertieft, da sie zum Verständnis des Ansatzes wenig beiträgt. Stattdessen eignet sie sich gut zur Formalisierung, insbesondere der Modelldynamik, von SESAM-Modellen.

Die drei genannten Teilmodelle Schemamodell, Situationsmodell und Regelmodell können zu einem dynamischen Modell kombiniert werden. Dabei erzeugt der Austausch zum Beispiel der Startsituation ein neues dynamisches Modell, ohne daß die anderen Teilmodelle geändert werden müßten. Auf diese Weise gewinnt man einen Modellierungsbaukasten aus Teilmodellen, dessen Bausteine jedoch recht grob sind.

Eine Aufteilung in Submodelle erscheint beim Schemamodell und beim Situationsmodell nicht sinnvoll, beim Regelmodell hingegen schon. Bisher wurde eine Aufteilung des Regelmodells in Submodelle nicht durchgeführt. Der Aktivitätenansatz in der neuen Modellbeschreibungssprache für SESAM-2 von Bernd Schneider (1996a, Kapitel 14) geht jedoch in diese Richtung. Hier werden Regeln, die einer übergeordneten Tätigkeit, zum Beispiel der Spezifikationsphase eines Projekts, zugeordnet werden können, in einer sogenannten *Aktivität* zusammengefaßt.

### 2.3.2 Ausführung

Die Ausführung, d.h. die Simulation, eines dynamischen Modells ist vom Prinzip her einfach. Zu Beginn wird der in der Startsituation beschriebene Anfangszustand hergestellt. Danach werden die im Regelmodell enthaltenen Regeln in jedem Simulationsschritt auf den Modellzustand angewandt. Durch das Absetzen von Kommandos, die ebenfalls den Modellzustand verändern können, kann der Spieler den Verlauf der Simulation beeinflussen.

Schwieriger hingegen ist es, festzulegen, wann welche Regel angewendet wird und welche Konsequenzen ihre Ausführung hat. Zur Ausführung von Regeln haben sich im Laufe der Entwicklung des SESAM-Systems zwei unterschiedliche Vorgehensweisen herausgebildet: das „Feuern“ und die „Aktivierung/Deaktivierung“.

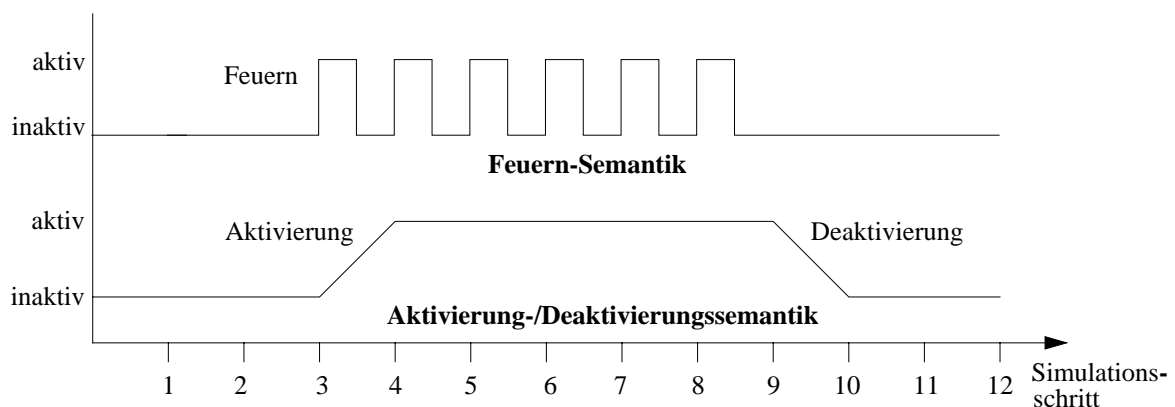
Das *Feuern* ist das einfachere der beiden Konzepte. Eine Regel feuert, wenn ihr Bedingungsteil erfüllt ist. Feuern bedeutet, daß der Aktionsteil der Regel ausgeführt wird. Danach wartet die Regel darauf, wieder feuern zu können.

- 
1. Ein Hypergraph ist ein Graph, bei dem eine Kante nicht nur wie üblich genau zwei, sondern beliebig viele Knoten verbinden kann.
  2. Graphgrammatiken sind Grammatiken, die im Unterschied zu den Chomsky-Grammatiken nicht über Zeichenketten, sondern über Graphen operieren. Näheres zu Graphgrammatiken kann Göttler (1988) und Schürr, Westfechtel (1992) entnommen werden.

Das Konzept der *Aktivierung/Deaktivierung* ist etwas komplizierter. Der Aktionsteil einer Regel zerfällt in drei Teile: einen *Aktivierungsteil*, einen *Aktivteil* und einen *Deaktivierungsteil*. Findet eine Regel einen passenden Teilzustand vor, aktiviert sie sich. Dabei führt sie den Aktivierungsteil aus. Solange der Bedingungsteil erfüllt ist, bleibt die Regel aktiv. Nun führt sie in jedem weiteren Simulationsschritt den Aktivteil aus. Wird der Modellzustand durch die Regel selbst oder durch andere Regeln so verändert, daß der Bedingungsteil nicht mehr erfüllt ist, so deaktiviert sich die Regel. Dabei führt sie den Deaktivierungsteil aus.

Die Idee hinter diesem Konzept ist, daß viele Effekte im Modell eigentlich keinen punktuellen Charakter haben, wie es das Feuernkonzept nahelegt, sondern eher kontinuierlichen Charakter. Diese Kontinuität wird durch die Aktivierung bei Beginn und die Deaktivierung zum Ende des Effekts modelliert. Bei jedem Simulationsschritt kann dann der Effekt selbst durch den Aktivteil wirksam werden.

Um den Unterschied zwischen den beiden Konzepten etwas deutlicher zu machen, nehmen wir an, daß der Bedingungsteil einer Regel zwischen dem dritten Simulationsschritt und dem achten Simulationsschritt (jeweils einschließlich) erfüllt ist. Davor und danach ist der Bedingungsteil nicht erfüllt. Abbildung 2.7 veranschaulicht nun, welche Aktionen der Regel bei Feuern und bei Aktivierung/Deaktivierung in den einzelnen Simulationsschritten ausgeführt werden.



**Abbildung 2.7: Vergleich Feuern und Aktivierung/Deaktivierung**

In der ersten Zeile ist die Feuern-Semantik dargestellt. In jedem Simulationsschritt, in dem die Bedingung der Regel erfüllt ist, feuert die Regel. Sie wird aktiv und nach ihrer Ausführung sofort wieder inaktiv.

Die zweite Zeile stellt die Aktivierungs-/Deaktivierungssemantik dar. Hier wird die Regel im dritten Simulationsschritt aktiviert. Sie führt die Aktivierungsaktionen aus. In den folgenden Schritten ist sie aktiv und führt jeweils ihre Aktivaktionen aus. Im neunten Simulationsschritt, dem ersten Schritt, in dem ihr Bedingungsteil nicht mehr erfüllt ist, führt sie die Deaktivierungsaktionen aus und geht in den inaktiven Zustand über. Der kontinuierliche Charakter der Aktivierungs-/Deaktivierungssemantik wird in dieser Darstellung besonders deutlich.

Die beiden Konzepte der Regelausführung sind bijektiv aufeinander abbildbar und können damit als gleich mächtig aufgefaßt werden. Das Feuern ist eine spezielle Art der Aktivierung/Deaktivierung. Eine Feuern-Regel entspricht einer Aktivierungs-/Deaktivierungsregel, deren Aktivierungs- und Aktivteil dem Aktionsteil der Feuern-Regel entspricht, und deren Deaktivie-

rungsteil leer ist. Eine Aktivierungs/Deaktivierungsregel läßt sich durch drei Feuerrregeln (je eine für den Aktivierungs-, den Aktiv- und den Deaktivierungsteil) simulieren. Näheres zu der Äquivalenz der Regelausführungssemantiken kann Schneider (1996b) entnommen werden.

## 2.4 Implementierungen

In Kapitel 1 wurden bereits die Entwicklung des SESAM-Systems und die Gründe für eine Neuimplementierung angedeutet. Diese Aspekte sollen im folgenden vertieft werden. Zugleich werden die Architekturen von SESAM-1 und SESAM-2 vergleichend vorgestellt.

### 2.4.1 SESAM-1

Das Pilotsystem SESAM-1, inzwischen bei der Version 1.2 angelangt, wurde nach drei vorausgegangenen Prototypen vollständig in Smalltalk-80 (Goldberg, Robson, 1990) implementiert. Dieses System ist per FTP<sup>1</sup> frei verfügbar. Durch die Online-Dokumentation des SESAM-1-Systems kann man sich aus der Sicht des Spielers (Dokument „Operating Instructions for SESAM Players“) und aus der Sicht des Modellierers (Dokument „Instructions for SESAM Model Builders“) über das SESAM-1-System informieren.

Die Architektur von SESAM-1 ist in Abbildung 2.8 dargestellt. Benutzer sind dabei als stilisierte Figuren dargestellt, Werkzeuge durch Rechtecke und Datenflüsse zwischen Werkzeugen durch Pfeile mit eingeschlossenen Ellipsen.

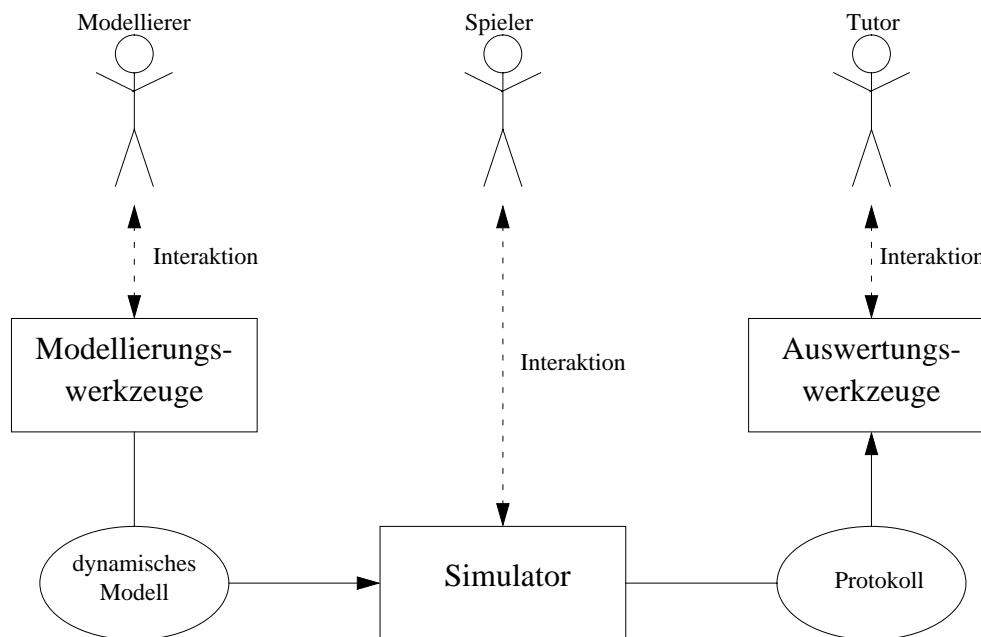


Abbildung 2.8: Architektur des SESAM-1-Systems

Dem Modellierer stehen als *Modellierungswerkzeuge* halbgraphische Editoren zur Entwicklung der drei Teilmodelle Schemamodell, Situationsmodell und Regelmodell zur Verfügung. Diese Teilmodelle lassen sich zu dynamischen Modellen verbinden und mit dem *Simulator* ausführen. Der Simulator interagiert mit dem Spieler, indem er dessen Eingaben entgegennimmt, für deren

1. unter <ftp://ftp.informatik.uni-stuttgart.de/pub/sesam/sesam1.1.tar.gz>

Umsetzung in der Modellsimulation sorgt und aus dem Modell kommende Ausgaben an den Spieler weiterleitet. Daneben führt der Simulator ein Protokoll über den Werteverlauf der Zustandsgrößen des Modells. Auf dieses Protokoll kann mit einem *Auswertungswerkzeug* zugegriffen werden, das vom Tutor bedient wird. Mit dem Auswertungswerkzeug können Zustandsgrößen des Modells nachverfolgt werden. Deren Entwicklung liefert Aufschluß über die Ursachen des erzielten Spielerfolgs (oder -mißerfolgs).

Der anfangs etwas chaotische Entstehungsprozeß von SESAM-1 führte dazu, daß die Implementierung unzureichend dokumentiert und auch schlecht wartbar wurde. Die Erfahrungen bei der Modellerstellung und Modellsimulation mit SESAM-1 führten dann auch zu geänderten Anforderungen an das SESAM-System. Insbesondere soll die Modellbeschreibungssprache von SESAM-1 modifiziert und um neue Konzepte erweitert werden. Deshalb entschloß sich das SESAM-Team, ein neues SESAM-System zu erstellen. Die Implementierung soll nicht mehr in Smalltalk-80, sondern in Ada 95 (Intermetrics, 1995) durchgeführt werden.

### 2.4.2 SESAM-Lite

Im Herbst 1995 implementierte Marcus Deininger einen Prototypen für das neue SESAM-2-System in Smalltalk-80. Dieser Prototyp besaß drei Aufgaben (Deininger, 1995): Er sollte

1. das prinzipielle Aussehen eines SESAM-Systems demonstrieren, dessen Simulator als Eingabe das Modell nicht in Form von Smalltalk-Objekten, sondern in einer rein textuellen Repräsentation (in einer Modellbeschreibungssprache formuliert) bekommt,
2. die zentralen Datenstrukturen von SESAM-2 skizzieren und
3. die zentralen Regel- und Simulationsmechanismen von SESAM extrahieren, da diese bisher nicht in expliziter Form vorlagen.

Der Prototyp führte das (für SESAM) neue Konzept der feuernenden Regeln ein, denn die Regeln in SESAM-1 besaßen Aktivierungs-/Deaktivierungssemantik. Die Erkenntnisse aus der Implementierung von SESAM-Lite flossen in die Diskussion über SESAM-2 ein. Es wurden jedoch nicht alle Konzepte übernommen.

### 2.4.3 SESAM-2

#### 2.4.3.1 Ziele der Neuimplementierung

In diesem Abschnitt werden die Zielsetzungen, die mit der Neuimplementierung des SESAM-Systems durch SESAM-2 verfolgt werden, zusammengefaßt.

- SESAM-2 soll alle für gut befundenen Konzepte von SESAM-1 aufgreifen und gleichzeitig erkannte Schwächen vermeiden.
- In SESAM-2 sollen neue Konzepte für die Modellbeschreibungssprache eingeführt werden, die sich aus den Erfahrungen mit SESAM-1 heraus als wünschenswert erwiesen haben.
- Alle Konzepte von SESAM-2 sollen vollständig und präzise spezifiziert und die Ergebnisse aller Kodierungstätigkeiten an SESAM-2 unter den Aspekten der Korrektheit und der Wartbarkeit geprüft werden. Diese qualitätssichernden Maßnahmen sollen die Wartbarkeitsproblematik, in die man mit SESAM-1 geraten war, zumindest mittelfristig vermeiden helfen.
- Die Semantik der Modellbeschreibungssprache in SESAM-2 soll explizit, vollständig und

eindeutig niedergelegt werden, um die Korrektheit der Implementierungen prüfen zu können.

- Das recht monolithische SESAM-1-System soll durch ein aus mehreren unabhängigen, miteinander kooperierenden Werkzeugen bestehendes System abgelöst werden. Das schlägt sich auch in der Architektur von SESAM-2 nieder. Diese ist im folgenden Abschnitt beschrieben.
- Die Implementierungssprache Smalltalk-80 wird zugunsten von Ada 95 aufgegeben. Das Problem bei der Verwendung von Smalltalk war, daß das gesamte SESAM-System aufgrund der Gegebenheiten in Smalltalk in einem Stück bearbeitet werden mußte und damit sehr unhandlich wurde.

Der Wechsel der Implementierungssprache schließt die Wiederverwendung von Code-Teilen des SESAM-1-System aus, was zunächst unnötigen Aufwand verursacht. Andererseits bietet sich die Chance, durch einen Neuanfang eine qualitativ bessere Implementierung zu erhalten, da das neue System keine „Altlasten“ mit sich herumschleppen muß. Aus diesem Grund gab es auch keinerlei Gedanken an eine Aufwärtskompatibilität von SESAM-2, die es ermöglicht hätte, Modelle für SESAM-1 in SESAM-2 auszuführen.

### 2.4.3.2 Architektur

Abbildung 2.9 zeigt die Architektur des SESAM-2-Systems. Im Vergleich zur Architektur von SESAM-1 in Abbildung 2.8 fällt auf, daß hier eine größere Anzahl von Werkzeugen miteinander interagieren.

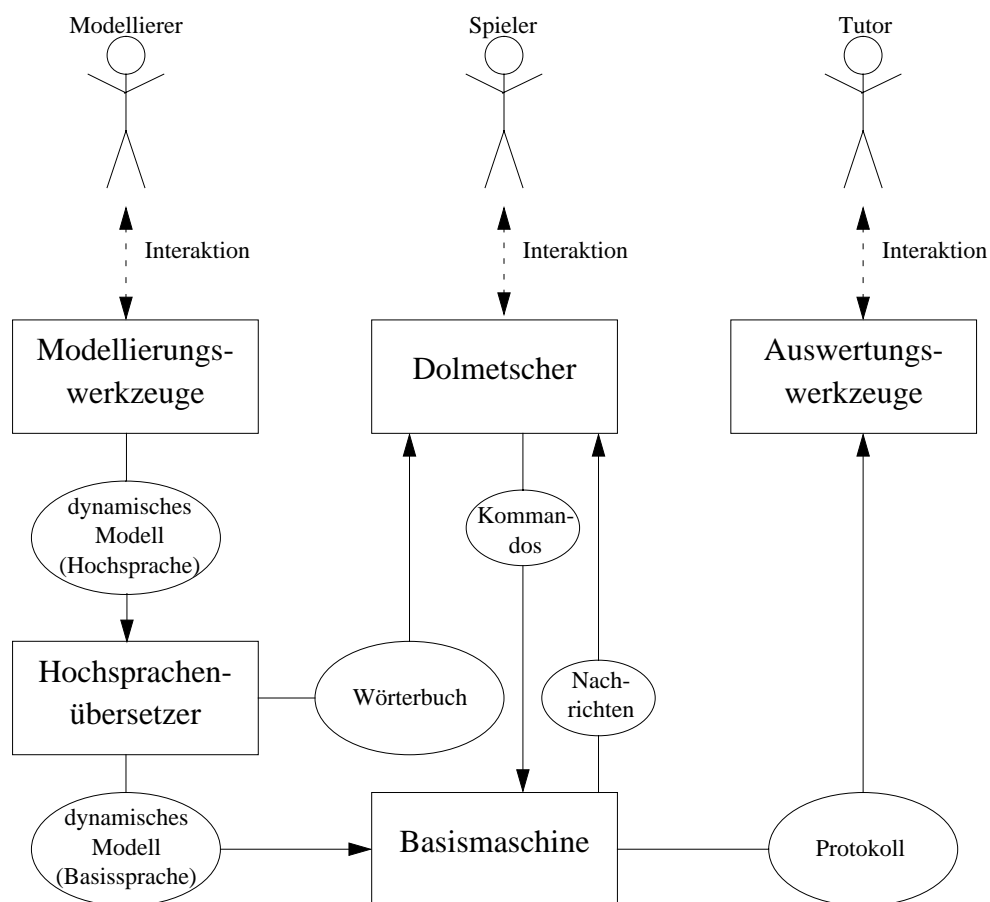


Abbildung 2.9: Architektur des SESAM-2-Systems

Das Konzept der *Modellierungswerkzeuge* ist auch in SESAM-2 als eigenständiger Bestandteil erhalten geblieben. Der Teil der Modellausführung wird jedoch stärker untergliedert. Es werden zwei verschiedene Ebenen der Modellbeschreibung unterschieden: Die Hochsprache und die Basissprache.

Die Hochsprache korrespondiert in etwa mit der Modellbeschreibungssprache von SESAM-1. Sie dient der Modellierung und bewegt sich in Bezug auf die tatsächliche Ausführung auf einem höheren Abstraktionsniveau als die Basissprache. Modellbeschreibungen in Hochsprache können mit dem *Hochsprachenübersetzer* in die Basissprache übersetzt werden.

Die Aufgabe der Basissprache ist es, mit möglichst wenig Konzepten eine universelle Beschreibungssprache für SESAM-Modelle zu sein. Ihr Ausführungsalgorithmus soll auch möglichst einfach sein. Die Unterschiede zwischen Hochsprache und Basissprache werden in Abbildung 2.10 zusammengefaßt.

	Hochsprache	Basissprache
Schwerpunkt	Modellentwicklung	Modellausführung
Sprachkonzepte	viele	wenige
Strukturiertheit	hoch	niedrig
Entwicklungskomfort	hoch	niedrig

**Abbildung 2.10: Unterschiede zwischen Hochsprache und Basissprache**

Die eigentliche Simulation des Modells wird durch die *Basismaschine* vorgenommen. Durch die Trennung von Hoch- und Basissprache können mehrere verschiedene Hochsprachen entwickelt werden, die alle auf der Basissprache aufbauen. Sie müssen sich nicht mit der Realisierung der Modellausführung befassen, da dies von der Basismaschine übernommen wird.

Die Interaktion mit dem Spieler wird durch den *Dolmetscher* durchgeführt, der ebenfalls neu ist. Die Kommunikation mit dem Benutzer geschieht auf rein textueller Basis in natürlicher Sprache. Der Dolmetscher nimmt die Eingaben des Spielers entgegen, verarbeitet sie und leitet sie in Form von Kommandos an die Basismaschine weiter. Die Basismaschine wiederum übergibt Ausgaben des Modells in Form von Nachrichten an den Dolmetscher, der die Nachrichten in benutzerverständlicher Form ausgibt. In Abschnitt 5.3.2 kann man den Dolmetscher im Einsatz sehen.

Der Dolmetscher bekommt zu einem Modell ein Wörterbuch, in dem spezifiziert ist, welche Kommandos das Modell versteht und welche Nachrichten es versenden kann. Zu jedem Kommando sind die möglichen Eingaben des Benutzers, die dieses Kommando auslösen sollen, genannt. Für jede Nachricht des Modells wird der Text angegeben, der beim Erhalt einer Nachricht dieses Typs ausgegeben werden soll.

Der Dolmetscher ist das bisher einzige realisierte Werkzeug im Rahmen des SESAM-2-Projekts. Näheres zum Dolmetscher und dem Wörterbuch kann der Diplomarbeit von André Spiegel (1995) entnommen werden.

Das Konzept der *Auswertungswerkzeuge* bleibt ebenfalls aus dem SESAM-1-System erhalten. Diese Werkzeuge bekommen als Eingabe ein Protokoll, das von der Basismaschine erzeugt wird und die Modellzustände während der Simulation enthält.

Die Aufgabenstellung dieser Arbeit umfaßt lediglich die Basismaschine. Daher wird im folgenden nur noch die Basismaschine und die zugehörige Modellbeschreibungssprache, die Basis-sprache, weiter betrachtet.

### 3 Die Basismaschine

*Siehe, ich lege [...] einen Grundstein, einen bewährten Stein, einen köstlichen Eckstein, der wohl gegründet ist.  
(Jesaja 28,16)*

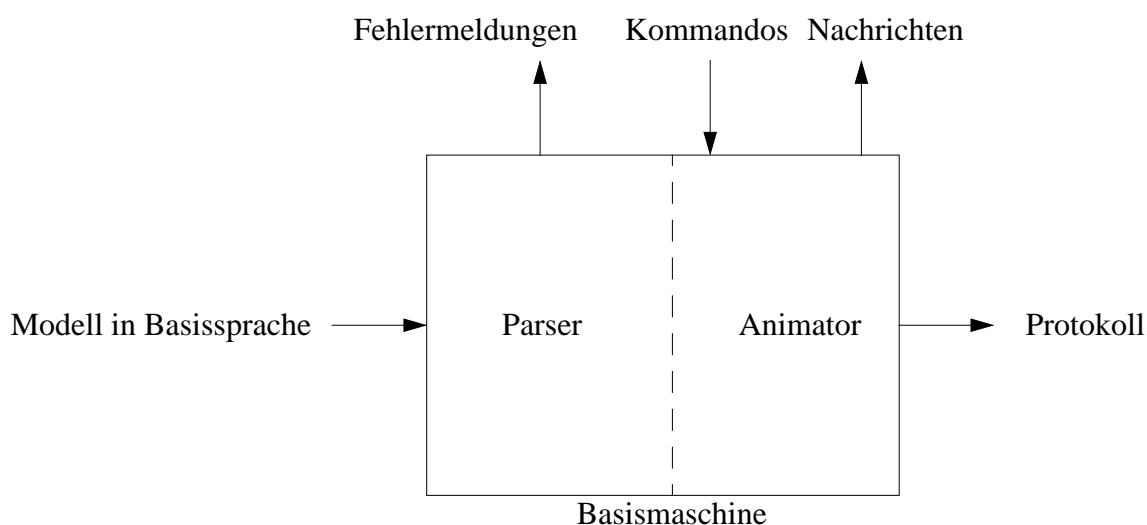
In diesem Kapitel wird die Aufgabe der Basismaschine genauer umrissen. Danach werden die Grundsätze der Konzeption der Basismaschine und wichtige Konzepte vorgestellt.

#### 3.1 Aufgabe

Die Basismaschine dient zur Ausführung eines regelbasierten, quantitativen Modells, das in einer Modellbeschreibungssprache (der Basissprache) vorliegt. Dabei führt sie folgende Aktivitäten aus:

- Der *Parser* der Basismaschine liest die Datei mit der Modellbeschreibung ein, analysiert sie auf syntaktische und semantische Fehler und erzeugt passende Fehlermeldungen. Ist das Modell fehlerfrei, wird es vom *Animator* ausgeführt. Der Anfangszustand des Modells wird hergestellt und es werden Folgezustände durch die Anwendung der Regeln auf den aktuellen Modellzustand erzeugt. Treten während der Simulation Laufzeitfehler auf, werden sie vom Animator gemeldet und die Simulation wird abgebrochen.
- Die Basismaschine kommuniziert mit einer natürlichsprachlichen Benutzungsoberflächen, dem Dolmetscher. Sie erhält von ihm die Kommandos des Benutzers und sendet die Ausgaben des Modells in Form von Nachrichten an ihn.
- Die Basismaschine legt jeden Modellzustand in einem Protokoll ab.

Die Schnittstellen der Basismaschine nach außen ergeben sich direkt aus den genannten Aufgaben. Abbildung 3.1 faßt die Schnittstellen zusammen. Die Darstellung ist im wesentlichen ein Ausschnitt von Abbildung 2.9, der Architektur von SESAM-2, um die Basismaschine herum.



**Abbildung 3.1: Schnittstellen der Basismaschine nach außen**

## 3.2 Grundsätze

*Make it as simple as possible, but no simpler.  
(Albert Einstein)*

Auf der Ebene der Konzeption steht der Entwurf der Basissprache im Vordergrund. Die Basismaschine als Ausführungswerkzeug der Basissprache tritt in den Hintergrund.

Die Basissprache kann als Programmiersprache für speziellen Anwendungen aufgefaßt werden. Daher kann die Basissprache auch nach den Entwurfskriterien für Programmiersprachen von Ghezzi und Jazayeri (1989) beurteilt werden. Ghezzi und Jazayeri unterscheiden bei ihren Entwurfskriterien die Bereiche Programmierbarkeit, Lesbarkeit und Zuverlässigkeit. Im folgenden sollen die Grundsätze des Sprachentwurfs der Basissprache bezogen auf diese drei Bereiche dargestellt werden.

### 3.2.1 Programmierbarkeit

Die Programmierbarkeit ist ein Maß dafür, inwieweit die Programmiersprache die Lösungsformulierung eines Problems unterstützt, ohne durch ihre Eigenschaften zusätzlichen Aufwand zu verursachen. Die Programmierbarkeit läßt sich in vier Unterkriterien klassifizieren. Diese sind Einfachheit, Ausdrucksstärke, Orthogonalität und Bestimmtheit. Die für die Basissprache wichtigsten Kriterien sind die ersten beiden.

**Einfachheit.** Einfachheit steht für die Leichtigkeit, mit der die Sprache erlernt und ihre Semantik verstanden werden kann. Die Zahl der Sprachkonstrukte und ihre Kombinierbarkeit spielen dabei eine wichtige Rolle.

Die Basissprache soll möglichst einfach sein. Deshalb wird eine möglichst geringe Anzahl von Sprachkonstrukten eingeführt, wodurch der Sprachumfang überschaubar bleibt. Der Ausführungsalgorithmus der Basissprache, der die dynamische Semantik festlegt, wird ebenfalls so einfach wie möglich gehalten.

**Ausdrucksstärke.** Die Ausdrucksstärke ist hoch, wenn eine Problemlösungsstrategie „natürlich“ in eine Programmstruktur abgebildet werden kann.

Die Ausdrucksstärke der Basissprache soll möglichst hoch sein. Die Struktur eines SESAM-Modells mit seinen typischen Bestandteilen (siehe Abschnitt 2.3.1) soll sich daher in der Basissprache wiederfinden. Außerdem sollen alle bereits bekannten Konzepte der Hochsprache möglichst einfach auf Konzepte der Basissprache abgebildet werden können.

**Orthogonalität.** Orthogonalität bedeutet, daß jede Kombination der Grundprimitive der Sprache möglich sein sollte.

Da es in der Basissprache so wenig Primitive wie möglich geben soll, hält sich auch die Zahl der Kombinationsmöglichkeiten im Rahmen. Orthogonalität soll im Sprachentwurf nur eine untergeordnete Rolle spielen.

**Bestimmtheit.** Die Bestimmtheit einer Programmiersprache steht für die Präzision, mit der die Syntax und die Semantik der Sprache definiert sind.

Um eine hohe Bestimmtheit zu erreichen, soll die Basissprache durch eine kontextfreie Grammatik für die Syntax und eine präzise formale Semantik beschrieben werden.

### 3.2.2 Lesbarkeit

Die Lesbarkeit eines Programms wird nicht nur durch den Programmierstil, sondern auch durch die Programmiersprache selbst bestimmt. Zu den Konzepten einer Sprache, die die Lesbarkeit erhöhen, zählen nicht nur Kommentare und geeignete Schlüsselwörter, sondern auch die Möglichkeit zur Abstraktion im Kleinen (Datenstrukturen, Unterprogramme) und im Großen (Module).

Die Basissprache wird voraussichtlich nicht zur Erstellung größerer Modelle herangezogen werden. Diese Aufgabe fällt eindeutig der Hochsprache zu. Deshalb kann in Zukunft von einer automatischen Generierung der Modellbeschreibungen in Basissprache durch einen Hochsprachenübersetzer ausgegangen werden. Deshalb spielt die Lesbarkeit nur eine untergeordnete Rolle.

Eine Modularisierung des Modells in Teilmodelle ist nicht vorgesehen. Dennoch sollte die textuelle Notation des Modells und seiner Bestandteile ein intuitives Verständnis ermöglichen. Deshalb orientiert sich die Syntax und die Semantik der Basissprache stark an Ada 95. Außerdem wurden möglichst sprechende Schlüsselwörter gewählt.

### 3.2.3 Zuverlässigkeit

Für die Zuverlässigkeit sind zwei verschiedene Kriterien entscheidend: die Prüfbarkeit der Korrektheit eines Programms und die Korrektheit der Implementierung des Programmausführungsmechanismus. Durch statische Prüfungen bei der Übersetzung können viele Fehler bereits vor der Programmausführung gefunden werden. Zusätzliche Laufzeitprüfungen helfen beim Finden weiterer Fehler.

Die Basissprache soll, wie Ada 95 mit seinem *strong typing*, ein strenges Typkonzept haben, das rigorose Typprüfung zur Übersetzungszeit erlaubt. Zusätzliche Prüfungen zur Laufzeit sollen helfen, Modellierungsfehler schneller und besser zu finden.

## 3.3 Konzepte

*Einen vollkommenen Entwurf hast du nicht erreicht, wenn du nichts mehr hinzufügen, sondern wenn du nichts mehr wegnehmen kannst.  
(Antoine de Saint-Excupéry)*

### 3.3.1 Vorgaben durch die Aufgabenstellung

Einige Konzepte der Basissprache sind bereits durch die Aufgabenstellung festgelegt worden. Diese Anforderungen werden im folgenden genannt.

- SESAM-Modelle sollen weiterhin als regelbasierte Modelle formuliert werden. Die Dreiteilung eines SESAM-Modells in Schemamodell, Situationsmodell und Regelmodell wird beibehalten.
- Die einzelnen Teilmodelle beinhalten die gewohnten Bestandteile (vgl. Abschnitt 2.3.1). Das Schemamodell setzt sich aus Entitätstypen und Relationstypen mit ihren Attributen zusammen. Das Situationsmodell besteht aus Entitäten und Relationen, während das Regel-

modell die Regeln mit Bedingungs- und Aktionsteil enthält.

- Ein SESAM-Modell wird auch weiterhin durch die systematische Anwendung von Regeln auf den aktuellen Modellzustand ausgeführt.

### 3.3.2 Annahmen über die Hochsprache

Alle Konzepte der Hochsprache sollen mit vertretbarem Aufwand in Konzepte der Basissprache übersetzbar sein. Deshalb bedarf es gewisser Grundannahmen über die Hochsprache, um die Basissprache entwerfen zu können. Die Grundannahmen für die Hochsprache ergeben sich aus Konzepten der Modellbeschreibungssprache in SESAM-1 und den Planungen für eine neue Hochsprache, die nun im Rahmen der Diplomarbeit von Bernd Schneider entsteht (siehe hierzu dessen Sprachbeschreibung in Schneider, 1996a). Ein konkreter Entwurf der Hochsprache lag zum Zeitpunkt der Konzeption der Basismaschine noch nicht vor. Folgende Annahmen wurden gemacht:

- Auch die Hochsprache nimmt die Trennung der drei Teilmodelle vor; eine Modellbeschreibung in der Hochsprache beschreibt ein spezifisches dynamisches Modell.
- Das Schemamodell besteht aus Entitäts- und Relationstypen, die Attribute haben.
- Das Situationsmodell besteht aus Instanzen der Entitäts- und Relationstypen des Schemamodells.
- Das Regelmodell beinhaltet Regeln mit Bedingungs- und Aktionsteil.
- Entitätstypen und Relationstypen können ihre Eigenschaften vererben. Dies entspricht einer Umsetzung des Generalisierungskonzepts des Entity-Relationship-Modells und war bereits in SESAM-1 möglich.
- Für die Attribute von Entitäts- und Relationstypen wird es Basistypen und darauf aufbauende Typkonstruktoren geben.

### 3.3.3 Wesentliche neue Konzepte

In diesem Abschnitt werden Konzepte der Basissprache zusammengestellt, die gegenüber den Modellbeschreibungssprachen bisheriger SESAM-Systeme neu sind. Der Schwerpunkt liegt hier nicht auf der ausführlichen Erklärung dieser Konzepte, da diese in Kapitel 4 genauer vorgestellt werden.

#### 3.3.3.1 Benutzerkommandos

Benutzerkommandos sind besondere Regeln, die zusätzlich noch formale Parameter haben können. Wenn der Benutzer im Modell eine bestimmte Aktion auslösen möchte, aktiviert er ein Benutzerkommando mit den notwendigen aktuellen Parametern. Benutzerkommandos können nur durch eine Benutzeraktion angestoßen werden, ansonsten dürfen sie nicht ausgeführt werden.

Beispielsweise könnte ein Entwickler mit einem Benutzerkommando `Hire(x)` eingestellt werden, dem beim Aufruf als Wert für den formalen Parameter `x` der einzustellende Entwickler mitgegeben wird. Das Benutzerkommando `Hire` nimmt daraufhin die notwendigen Veränderungen im Modellzustand vor.

Die Menge der Benutzerkommandos definiert gleichzeitig die Eingabeschnittstelle des Modells nach außen. Da der Dolmetscher die Benutzereingaben in Kommandos an die Basismaschine umsetzt, muß die Menge der Benutzerkommandos mit der Menge der Kommandos im Wörterbuch des Dolmetschers korrespondieren.

### 3.3.3.2 Prioritäten bei Regeln

Regeln haben Prioritäten. Je höher die Priorität einer Regel ist, desto früher wird sie bei der Regelausführung berücksichtigt. Auf diese Weise kann zum Beispiel eine Regel mit hoher Priorität die nötigen Voraussetzungen schaffen, die es einer Regel mit niedrigerer Priorität erlauben, innerhalb desselben Simulationsschritts ausgeführt zu werden. Die Idee der Einführung von Prioritäten stammt aus SESAM-Lite.

### 3.3.3.3 Regelausführungssemantik „Feuern“

Im Gegensatz zu SESAM-1, wo eine Regel aktiviert und deaktiviert wird, haben Regeln in der Basissprache Feuernsemantik (vgl. Abschnitt 2.3.2). Wenn der Bedingungsteil der Regel erfüllt ist, wird der Aktionsteil ausgeführt: sie feuert. Die Regel bleibt jedoch nicht aktiv, sondern wird sofort wieder inaktiv. Die Feuernsemantik von Regeln stellt in den meisten regelbasierten Sprachen wie zum Beispiel OPS5 (Krickhahn, Radig, 1987) den Normalfall dar. Sie wurde ebenfalls aus SESAM-Lite übernommen.

### 3.3.3.4 Nachrichten

Da der Dolmetscher als neue Benutzerschnittstelle fungieren soll, werden die Ausgaben des Modells durch eine spezielle Prozedur (`send_message`) realisiert. Diese wird mit einem Nachrichtenbezeichner und die notwendigen Parameter für die Nachricht aufgerufen. Der Dolmetscher generiert aus einer Nachricht und ihren Parametern anhand des Wörterbucheintrags der Nachricht einen natürlichsprachlichen Text.

Das Benutzerkommando `Hire` aus Abschnitt 3.3.3.1 stellt einen Entwickler ein. Um dem Benutzer eine Rückmeldung über den Erfolg seiner Aktion zu geben, soll eine Nachricht an ihn geschickt werden, die aussagt, daß der Entwickler eingestellt wurde. Also schickt das Modell dem Dolmetscher die Nachricht „`Hired(John)`“. Der Nachrichtenbezeichner ist `Hired`. Unter diesem Bezeichner schlägt der Dolmetscher im Wörterbuch nach und gibt den richtigen Text aus, wobei er den Namen des Entwicklers `John` in den Text einbaut.

### 3.3.3.5 Typverträglichkeit

In SESAM-1 gibt es die Möglichkeit der Vererbung zwischen Entitätstypen und auch zwischen Relationstypen. Dieses Konzept soll auch die neue Hochsprache besitzen (Schneider, 1996a, Kapitel 7 und 8), da es sich als nützlich erwiesen hat.

In der Basissprache wird keine Vererbungsnotation eingeführt. Stattdessen wird eine Möglichkeit vorgesehen, einen Typ zu einem anderen Typ als typverträglich zu deklarieren. Voraussetzung für eine solche Deklaration ist, daß der als typverträglich deklarierte Typ *B* mindestens dieselben Attribute wie der andere Typ *A* besitzt. Dann sind alle Zugriffe auf Attribute von *A* auch für *B* möglich.

Die Typverträglichkeit ist in vielen objektorientierten Programmiersprachen (zum Beispiel in Eiffel, vgl. Meyer, 1992) eine der Konsequenzen aus einer Vererbungsbeziehung. Erbt ein Typ  $B$  von einem Typ  $A$ , so ist  $B$  zu  $A$  automatisch typverträglich. Um die Vererbungsmöglichkeiten der Hochsprache nicht einzuschränken, wurde kein explizites Vererbungskonzept eingeführt. Die praktischen Konsequenzen der Vererbung sind stattdessen durch Typverträglichkeitsdeklarationen modellierbar (vgl. hierzu Abschnitt 4.2.2.3).

## 4 Die Basissprache

*Language shapes the way we think and determines what we can think about.*  
(B. L. Whorf)

In diesem Kapitel wird die Modellbeschreibungssprache der Basismaschine, die Basissprache, vorgestellt. Nach einer Übersicht über den Aufbau einer Modellbeschreibung (Abschnitt 4.1) in der Basissprache werden die wichtigsten Sprachbestandteile vorgestellt (Abschnitt 4.2 bis Abschnitt 4.4). Daran schließt sich die Beschreibung der Modelldynamik in Form eines Ausführungsalgorithmus an (Abschnitt 4.5). Zum Schluß werden die Unterschiede zur Modellbeschreibungssprachen von SESAM-1 und SESAM-Lite betrachtet (Abschnitt 4.6).

Die Angabe der Semantik von Sprachkonstrukten erfolgt informell. Eine formale Semantik der Basissprache kann der Sprachbeschreibung (Reißing, 1996) entnommen werden. Die formale Semantik stützt sich auf die Modellierung des Schemamodells und des Situationsmodells als Hypergraphen. Gearbeitet wird daneben mit Mengen und Relationen sowie Funktionen über diesen Mengen. Der verfolgte Ansatz orientiert sich an den Formalisierungen in Schneider (1994), weicht jedoch an manchen Stellen entscheidend ab.

Die Basissprache trägt den Namen BASE-2, eine Abkürzung für Basissprache für SESAM-2. BASE ist dabei als das englische Wort für Basis oder Fundament zu interpretieren.

### 4.1 Übersicht

Eine Modellbeschreibung in BASE-2 besteht aus

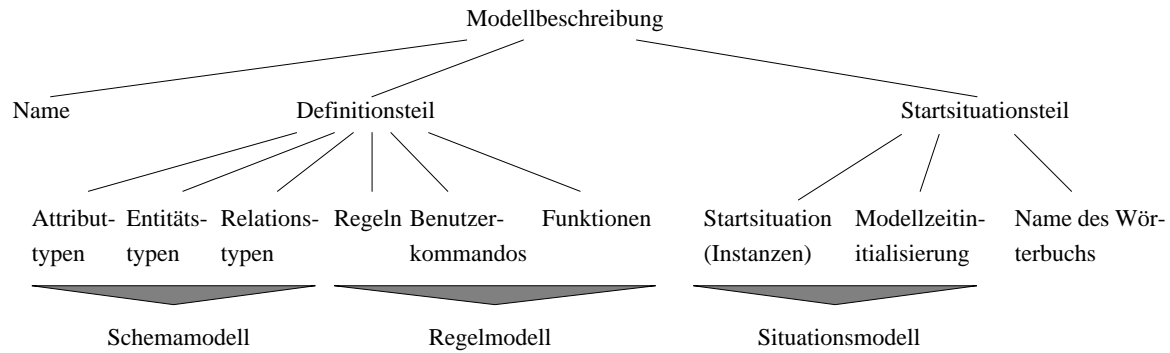
- einem Namen,
- einem Definitionsteil und
- einem Startsituationsteil.

Der Name des Modells dient der eindeutigen Identifikation, vor allem für die Daten des Protokolls. Im Definitionsteil werden das Schemamodell und das Regelmodell definiert. Im Startsituationsteil wird das Situationsmodell, das den Anfangszustand des Modells beschreibt, festgelegt. Zudem werden noch Initialisierungsinformationen für die Modellzeit und das Dolmetscher-Wörterbuch bereitgestellt. Abbildung 4.1 gibt einen graphischen Überblick über den Aufbau der Modellbeschreibung.

Der Definitionsteil umfaßt die Definitionen

- der Attributtypen,
- der Entitätstypen,
- der Relationstypen,
- der Regeln,
- der Benutzerkommandos und der
- benutzerdefinierten Funktionen.

Attributtypen, Entitätstypen und Relationstypen bilden zusammen das Schemamodell. Regeln, Benutzerkommandos und benutzerdefinierte Funktionen ergeben zusammen das Regelmodell.



**Abbildung 4.1: Aufbau einer Modellbeschreibung in BASE-2**

Der Startsituationsteil enthält

- eine Menge von Entitäten und Relationen,
- den Anfangswert der Modellzeit (das Startdatum),
- die Simulationsschrittweite und
- den Namen des Wörterbuchs für den Dolmetscher.

Durch diese Struktur kommen die drei Teilmodelle Schemamodell, Situationsmodell und Regelmodell nicht als solche zur Geltung, sie werden eher miteinander vermischt. Sie können jedoch aufeinanderfolgend notiert werden: Zuerst alle Definitionen des Schemamodells, dann alle Definitionen des Regelmodells und schließlich das Situationsmodell der Startsituation.

Auf der Ebene der Basissprache liegt der Schwerpunkt auf dem Bereich der Ausführung. Für diesen Aspekt ist die vorliegende Struktur gut geeignet, da ein einziger Durchlauf durch die Modellbeschreibung genügt, um alle notwendigen Informationen herauslesen zu können. Aus diesem Grund müssen Definitionen vor ihrer Verwendung in einer anderen Definition bereits eingeführt worden sein, wie das auch Ada 95 der Fall sein muß.

## 4.2 Schemamodell

Das Schemamodell setzt sich aus den Attributstypen, den Entitätstypen und den Relationstypen zusammen. Diese werden nun nacheinander näher erläutert.

### 4.2.1 Attributtypen

Die Attributtypen definieren die Menge von Typen, von denen Attribute im Schemamodell deklariert werden dürfen. Es gibt zwei verschiedene Arten von Attributtypen: Basistypen und Listentypen.

#### 4.2.1.1 Basistypen

Die Basistypen sind die vordefinierten Attributtypen. Im Sinne einer möglichst hohen Einfachheit der Sprache gibt es nur fünf vordefinierte Typen:

- Boolean,
- Integer,
- Real,

- String und
- Date.

Boolean ist ein Typ für die Wahrheitswerte `true` und `false`. Der numerische Typ `Integer` steht für eine Teilmenge der ganzen Zahlen. `Real` ist ein numerischer Typ, der eine Teilmenge der reellen Zahlen repräsentiert. Der Typ `String` steht für Zeichenketten über der Menge der ASCII-Zeichen. Der Typ `Date` repräsentiert gültige Zeitpunkte, die auf der Zeiteinheit Minuten basieren und aus Datum und Uhrzeit bestehen. Ein `Date`-Wert wird wie folgt notiert: zuerst kommt das Jahr, anschließend der Monat, der Tag, die Stunden und schließlich die Minuten. Beispielsweise sieht der 20.11.1995, 8:00 Uhr so aus:

```
1996/11/20/08:00
```

Die Basistypen `Boolean`, `Integer`, `Real` und `String` sind auch aus vielen anderen Programmiersprachen, wie zum Beispiel `Modula-2` (Wirth, 1988) oder `Ada 95` (Intermetrics, 1995), bekannt.<sup>1</sup> Sie besitzen die meisten der dort üblichen Operationen, und auch ihre Werte werden wie üblich notiert. Die vordefinierten Operationen über den Basistypen sind in der Sprachbeschreibung (Reißing, 1996) in ausführlicher Form zusammengestellt.

Ungewöhnlich ist lediglich der Typ `Date`, der jedoch für die Zeitaspekte des Modells benötigt wird. Die Modellzeit wird als Wert vom Typ `Date` repräsentiert (siehe Abschnitt 4.5.1).

#### 4.2.1.2 Listentypen

Die Liste ist der einzige Typkonstruktor in `BASE-2`. Mit Hilfe des Listenkonstruktors lassen sich Typen für Listen über Basistypen definieren. Diese Typen sind benannt, weil die Typäquivalenz für Attributtypen in `BASE-2` auf Namensäquivalenz<sup>2</sup> beruht.

Es wäre möglich gewesen, auf die Deklaration von benannten Listentypen zu verzichten und stattdessen mit unbenannten Listentypen (in den Attributdeklarationen) zu arbeiten. Da es bei Listentypen von derselben Struktur unterschiedlichste Semantiken geben kann, wurde wegen der besseren Lesbarkeit einer Benennung der Vorzug gegeben.<sup>3</sup>

Eine Deklaration einer Liste von Strings, die als `WordList` bezeichnet wird, sieht syntaktisch wie folgt aus:

```
type WordList is list of String;
```

Die Beschränkung der Typkonstruktoren auf den Listenkonstruktor hat einen Grund: die in der Hochsprache bisher vorgesehenen Datenstrukturen, z.B. `Stack`, `Queue`, `Set` oder `Array`, lassen sich auf die Liste abbilden (vgl. hierzu Reißing, 1996, Anhang F). Die einzige Ausnahme bildet der Verbund (`Record`), der auf der Ebene der Basissprache in seine Bestandteile aufgelöst wer-

---

1. `Real` heißt in `Ada 95` `Float`.

2. Namensäquivalenz bedeutet, daß zwei Objekte denselben Typ haben, wenn ihre Typen denselben Namen haben (Gegensatz: Strukturäquivalenz; vgl. Ghezzi, Jazayeri, 1989, S.199).

3. Dies kann die Lesbarkeit automatisch generierten Codes stark erhöhen. Listen können zur Simulation unterschiedlichster Datentypen, z.B. einer Warteschlange herangezogen werden. Durch die Vergabe eines entsprechenden Bezeichners für den Listentyp wird so die beabsichtigte Semantik des Typs besser erkennbar.

den muß, da eine Liste über verschiedenartigen Typen nicht erlaubt ist. Ein Attribut der Hochsprache von einem Record-Typ muß in einzelne Attribute (für jede Komponenten des Record-Typs eines) umgesetzt werden.

Ein Nachteil der Abbildung der meisten Datenstrukturen auf die Liste ist ein Effizienzverlust einiger Operationen auf diesen Datentypen. Das liegt daran, daß zum Zugriff auf ein beliebiges Listenelement zuerst eine sequentielle Suche in der Liste nötig ist. Eine entsprechende Implementierung der Liste kann dem entgegenwirken, indem die Zugriffsoperationen auf die Liste besonders effizient gestaltet werden.

Die Attributtypen finden vor allem in der Definition von Entitäts- und Relationstypen Verwendung, die im folgenden vorgestellt werden.

## 4.2.2 Entitätstypen

Entitätstypen sind Typen für die Objekte der abstrakten Welt, die durch das Modell definiert ist und in der die Dynamik des Modell abläuft. Diese Objekte, die sogenannten Entitäten, besitzen Attribute, deren Deklaration auf Typebene durchgeführt wird, während eine Wertzuweisung erst auf Objektebene möglich ist.

### 4.2.2.1 Deklaration

Entitätstypen haben einen (eindeutigen) Namen und bestehen aus einer optionalen Typverträglichkeitsklausel sowie den Attributdeklarationen. Die Deklaration eines Attributs setzt sich aus dem Attributnamen und dem Attributtyp zusammen.

Im folgenden Beispiel wird ein Entitätstyp `Person` definiert, der die Attribute `name` und `age` besitzt. Das Attribut `name` ist vom Typ `String`, das Attribut `age` vom Typ `Integer`.

```
entity type Person is
  attributes
    name: String;
    age: Integer;
end type;
```

Das genannte Beispiel besitzt keine Typverträglichkeitsklausel. Dieses Konzept wird im folgenden Abschnitt eingeführt.

### 4.2.2.2 Typverträglichkeit

Typverträglichkeit ist eine Beziehung zwischen Typen. Wenn ein Typ *A* zu einem Typ *B* typverträglich ist, kann ein Objekt vom Typ *A* anstelle eines Objekts vom Typ *B* verwendet werden, da *A* alle Eigenschaften von *B* zur Verfügung stellt. Jeder Typ ist zu sich selbst typverträglich.

Die Deklaration eines Entitätstyps `Developer`, der zu dem Typ `Person` typverträglich ist, sieht in BASE-2 so aus:

```
entity type Developer conforming Person is
  attributes
    name: String;
    age: Integer;
    experience: Real;
```

```

    motivation: Real;
    cost_per_day: Real;
end type;

```

In der conforming-Klausel wird der Entitätstyp `Developer` zum Typ `Person` als typverträglich deklariert. Er verfügt über dieselben Attribute wie der Entitätstyp `Person`, daher ist die Typverträglichkeitsdeklaration zu `Person` legal. Zusätzlich besitzt `Developer` noch drei weitere Attribute: ein Attribut `experience` vom Typ `Real`, das seine Erfahrung als Zahl zwischen 0 und 1 aufnehmen soll, ein Attribut `motivation` vom Typ `Real`, das seine Motivation, ebenfalls als Zahl zwischen 0 und 1, speichert, und ein Attribut `cost_per_day`, das die Personalkosten pro Tag für diesen Entwickler aufnimmt.

#### 4.2.2.3 Warum Typverträglichkeit?

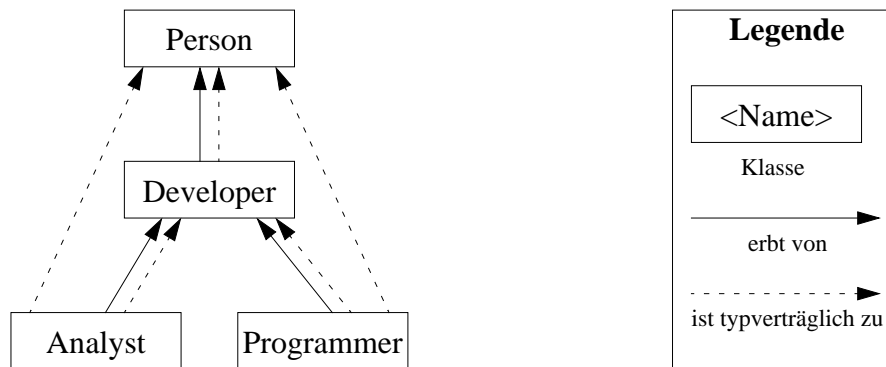
Hier soll motiviert werden, wofür dieses Konzept überhaupt benötigt wird. Wie bereits in Abschnitt 3.3.3.5 gesagt, gibt es in BASE-2 kein explizites Vererbungskonzept. Vererbung ist jedoch eine praktische Methode, Begriffshierarchien programmtechnisch umzusetzen. Die Spezialisierung eines Begriffs wird durch die Angabe des Oberbegriffs und der Verfeinerungen in Form neuer Attribute hinreichend beschrieben. Es ist nicht erforderlich, alle Attribute noch einmal neu aufzuschreiben.

Ein weiterer Vorteil der Vererbung ist, daß sich Verbesserungen in der Repräsentation des Oberbegriffs automatisch auf alle Unterbegriffe übertragen. Schließlich lassen sich die durch die Vererbung entstandenen Verwandtschaftsbeziehungen, die sogenannten Typverträglichkeitsbeziehungen, bei der Programmierung nutzen. So können zum Beispiel Regeln formuliert werden, die mit Oberbegriffen operieren und sich dann auch auf alle Unterbegriffe anwenden lassen.

**Zusammenhang zwischen Vererbung und Typverträglichkeit.** Wie das Konzept der Typverträglichkeit mit dem der Vererbung zusammenhängt, läßt sich an einem einfachen Beispiel verdeutlichen. Eine Klasse `Person` habe eine Unterklasse `Developer`, diese wiederum zwei Unterklassen: `Analyst` und `Programmer`. Die Unterklassen erben jeweils die Attribute ihrer direkten Vorgängerklasse. Dadurch erben sie (transitiv) alle Attribute ihrer Vorgängerklassen in der Vererbungshierarchie. Eine erbende Klasse kann höchstens Attribute hinzufügen, jedoch keine entfernen.

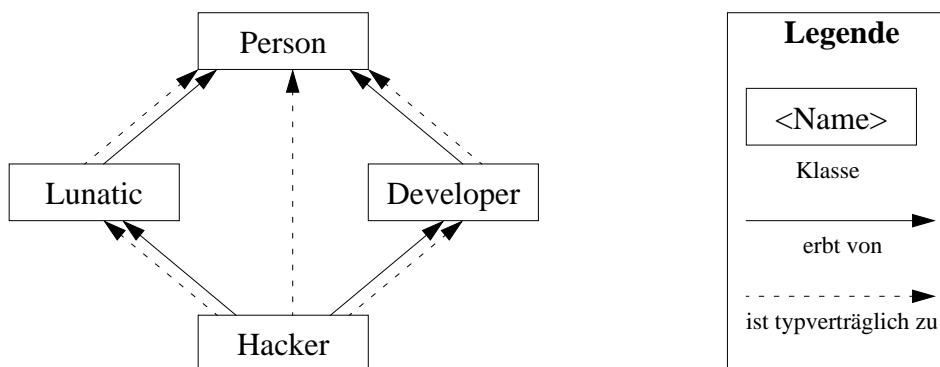
Die Typverträglichkeit sagt aus, daß anstelle eines Objekts vom Typ `Person` ein Objekt einer beliebigen Unterklasse verwendet werden darf, da diese durch die Vererbung dieselben Attribute besitzt. Die Typverträglichkeitsrelation ist damit die transitive Hülle der Vererbungsrelation. Abbildung 4.2 macht das graphisch deutlich.

**Vorteile.** Der Vorteil der Typverträglichkeitsnotation ist zum einen, daß bei der Typdefinition immer *alle* Attribute sichtbar sind und nicht erst mühsam durch Betrachten der Oberklassen zusammengesucht werden müssen. Zum anderen läßt sich damit auch Mehrfachvererbung modellieren. Die Klasse `Person` aus dem obigen Beispiel habe eine weitere Unterklasse `Lunatic`



**Abbildung 4.2: Zusammenhang zwischen Vererbung und Typverträglichkeit**

(Verrückter). Die Klasse Lunatic ist Oberklasse für die Klasse Hacker, die auch von Developer erbt. Damit ist Hacker sowohl zu Lunatic als auch zu Developer typverträglich. Diese Vererbungshierarchie ist in Abbildung 4.3 dargestellt.



**Abbildung 4.3: Mehrfachvererbung und Typverträglichkeit**

In BASE-2-Notation würde der Entitätstyp Hacker wie folgt aussehen:

```
entity type Hacker conforming Lunatic, Developer, Person is
  attributes
  ...
end type;
```

Die conforming-Klausel von Hacker enthält alle drei unmittelbaren und mittelbaren Vorgänger in der Vererbungshierarchie.

**Nachteile.** Die genannten Vorteile der Typverträglichkeitsnotation können auch als Nachteile interpretiert werden. Dadurch, daß immer alle Attribute eines Typs aufgeschrieben werden müssen, ist es notwendig, Veränderungen in der Oberklasse auch in allen Unterklassen vorzunehmen. Außerdem müssen bei einer Typverträglichkeitsdeklaration alle Oberklassen genannt werden, die direkte Oberklasse genügt nicht. Diese Nachteile fallen jedoch bei einer automatischen Generierung der Modellbeschreibung nicht ins Gewicht.

### 4.2.3 Relationstypen

Relationstypen sind Typen für die Beziehungen zwischen den Objekten der abstrakten Welt des Modells, also für Beziehungen zwischen Entitäten. Diese Beziehungen, die sogenannten Relationen, besitzen Rollen und Attribute.

#### 4.2.3.1 Deklaration

Relationstypen haben einen (eindeutigen) Namen und bestehen aus einer optionalen Typverträglichkeitsklausel sowie den Rollen und den Attributen. Die Deklaration einer Rolle setzt sich aus dem Rollennamen und dem Rollentyp, der ein Entitätstyp sein muß, zusammen. Durch die Rollen werden die Beziehungen zwischen den Entitäten modelliert. Da Beziehungen immer mindestens zwei Partner haben, müssen Relationstypen auch mindestens zwei Rollen besitzen. Die Attribute, die analog zu den Entitätstypen deklariert werden, drücken Eigenschaften der Beziehung aus.

Im folgenden Beispiel wird ein Relationstyp `TalksTo` (unterhält sich mit) definiert, der keine Attribute und zwei Rollen aufweist: die Personen, die sich miteinander unterhalten (`who` und `whom`).

```
relation type TalksTo is
  roles
    who: Person;
    whom: Person;
  end type;
```

#### 4.2.3.2 Typverträglichkeit

Auch zwischen Relationstypen können Typverträglichkeitsbeziehungen bestehen. Zusätzlich zu der Typverträglichkeit zwischen Entitätstypen, die auf gleichen Attributen beruht, kommen hier die Rollen ins Spiel. Die Rollen können auf der Ebene der Typverträglichkeit als spezielle, entitätswertige Attribute aufgefaßt werden. Deshalb muß ein Relationstyp, wenn er sich zu einem anderen Typ als typverträglich deklarieren will, auch dessen Rollen besitzen. Der Typ einer Rolle muß jedoch nicht exakt derselbe sein; auch dazu typverträgliche Entitätstypen sind erlaubt.

Im ersten Beispiel wird ein Relationstyp `TalksToAbout` eingeführt, der nur das zusätzliche Attribut `about` enthält und gleichzeitig illustriert, wie ein Relationstyp mit Attributen aussieht.

```
relation type TalksToAbout conforming TalksTo is
  roles
    who: Person;
    whom: Person;
  attributes
    about: String;
  end type;
```

Im zweiten Beispiel werden Rollen redefiniert, d.h. von einem zum ursprünglichen Typ typverträglichen Entitätstyp deklariert. Es wird ein spezieller Relationstyp `Interviews` für ein Bewerbungsgespräch eingeführt. Die Rolle `who`, deren Typ in `TalksTo` `Person` war, ist jetzt

vom Typ `Manager`, einem zu `Person` typverträglichen Typ. Die Rolle `whom`, deren ursprünglicher Typ ebenfalls `Person` war, ist nun vom Typ `Developer`, der auch zu `Person` typverträglich ist.

```
relation type Interviews conforming TalksTo is
roles
  who: Manager;
  whom: Developer;
end type;
```

Natürlich sind auch Kombinationen aus der Hinzufügung von Attributen und der Redefinition von Rollen möglich.

### 4.3 Regelmodell

Das Regelmodell bestimmt die Dynamik des Modells. Es werden Effekte auf dem Schemamodell definiert.

Effekte können zum einen modellintern ausgelöst werden. Diese Effekte werden durch Regeln modelliert. Zum anderen können Effekte von außen, durch den Benutzer, ausgelöst werden. Solche externen Effekte werden durch die Benutzerkommandos modelliert. Im folgenden werden die beiden Definitionsarten Regeln und Benutzerkommandos vorgestellt.

Zur Vereinfachung von Berechnungen in Zuweisungen und Bedingungen wurde BASE-2 noch um die Möglichkeit erweitert, Funktionen auf Attributtypen zu definieren. Diese benutzerdefinierten Funktionen werden im Anschluß eingeführt.

#### 4.3.1 Regeln

Regeln bestehen aus drei Teilen: dem Regelkopf, dem Bedingungsteil und dem Aktionsteil. Der Bedingungsteil legt fest, wann die Regel feuern darf und der Aktionsteil bestimmt, was die Regel tun soll, wenn sie feuert.

##### 4.3.1.1 Regelkopf

Der Regelkopf enthält einen (eindeutigen) Namen für die Regel, eine Priorität und einen Zeitverbrauch. Die Priorität ist eine ganze Zahl zwischen 0 und 1000. Sie legt fest, wann eine Regel in einem Simulationsschritt ausgeführt werden soll. Je höher die Priorität ist, desto bevorzugter wird die Regel behandelt (vgl. hierzu den Ausführungsalgorithmus in Abschnitt 4.5.2). Der Zeitverbrauch gibt an, wieviel Zeit in Minuten dem Projektleiter durch die Ausführung der Regel verloren geht.

Dieser Zeitverbrauch ist in Regeln meistens 0, da die meiste Zeit des Projektleiters durch die Ausführung von Benutzerkommandos verbraucht wird. Auf das Zeitkonzept in BASE-2 und die Auswirkungen des Zeitverbrauchs wird hier nicht eingegangen. Das wird in Abschnitt 4.5 nachgeholt.

Ein Beispiel für einen Regelkopf aus Name, Priorität und Zeitverbrauch sieht so aus:

```
rule EnhanceSpecification[1000] taking 0 is
...
```

Die Regel `EnhanceSpecification` hat die höchstmögliche Priorität<sup>1</sup> von 1000 und kostet den Projektleiter bei ihrer Ausführung 0 Minuten Zeit, also keine Zeit.

#### 4.3.1.2 Bedingungsteil

Der Bedingungsteil zerfällt in zwei Teile: den Strukturbedingungsteil und den Attributbedingungsteil.

**Strukturbedingungsteil.** Im Strukturbedingungsteil wird die Struktur angegeben, die in der aktuellen Situation vorgefunden werden muß, damit die Regel feuern kann. Die Struktur wird durch formale Entitäten und formale Relationen beschrieben.

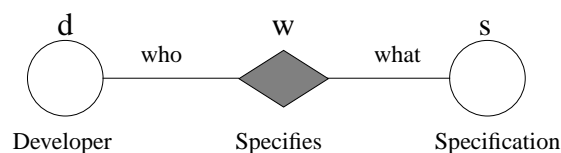
Die formalen Entitäten haben einen Namen und einen Entitätstyp. Die formalen Relationen sind ebenfalls benannt und haben einen Relationstyp sowie eine Rollenbelegung, die aus bereits deklarierten formalen Entitäten besteht. Die formalen Relationen beschreiben die Beziehungen, die zwischen den formalen Entitäten bestehen müssen.

Der Strukturbedingungsteil wird durch das Schlüsselwort `structure` eingeleitet. Der Strukturbedingungsteil der Regel `EnhanceSpecification`, deren Kopf bereits vorgestellt wurde, sieht so aus:

```
structure
  s: Specification;
  d: Developer;
  w: Specifies(d,s);
```

Gefordert wird eine Spezifikation `s`, ein Entwickler `d` und eine `Specifies`-Beziehung zwischen `d` und `s`. Auf diese Beziehung kann unter dem Namen `w` zugegriffen werden. Um die Strukturbedingung zu erfüllen, dürfen auch Instanzen typverträglicher Entitäts- und Relationstypen herangezogen werden. Also dürfte zum Beispiel für den Entwickler `d` auch eine Instanz des Entitätstyps `Programmer` verwendet werden, da `Programmer` zu `Developer` typverträglich ist.

Abbildung 4.4 stellt den Strukturbedingungsteil graphisch dar. Die graphische Notation ist dabei an die in Schneider (1994) verwendete angelehnt. Die formalen Entitäten werden durch Kreise, die formale Relation durch eine gefüllte Raute dargestellt. Unter den formalen Komponenten steht ihr Typ, oberhalb ihr Bezeichner. Die Rollen der formalen Relation sind durch Kanten repräsentiert, die mit dem Rollennamen beschriftet sind.



**Abbildung 4.4: Graphische Darstellung des Strukturteils**

Die Zuordnung einer Entität aus der aktuellen Situation zu einer formalen Entität nennt man *Bindung* der formalen Entität. Die Bindung einer formalen Entität muß mit einer Entität des geforderten Entitätstyps durchgeführt werden. Erlaubt sind darüber hinaus auch Entitäten eines

1. Die in BASE-2 verwendete Notation der Priorität in eckigen Klammern ist inspiriert durch Modula-2, wo Modulen auf dieselbe Weise eine Priorität zugeordnet werden konnte (vgl. Wirth, 1988, S. 124).

typverträglichen Typs. Dadurch, daß auch Entitäten typverträglichen Typs zugelassen sind, muß eine Regel nicht für alle Untertypen neu formuliert werden. Dies erhöht die Übersichtlichkeit des Modells gewaltig.

Der Begriff der Bindung kann auf den ganzen Strukturbedingungsteil erweitert werden, indem man zu jeder formalen Entität eine passende Entität in der aktuellen Situation findet und auch die formalen Relationen in der aktuellen Situation Entsprechungen für die gebundenen formalen Entitäten haben. Innerhalb einer Bindung darf dieselbe Entität nicht mehrfach gebunden werden, die formale Struktur wird also isomorph auf die aktuelle Situation abgebildet. Zu einer formalen Struktur kann es in einer Situation mehrere verschiedene Bindungen geben.

**Attributbedingungsteil.** Damit eine Regel wirklich feuern kann, muß nicht nur eine Bindung für die formale Struktur gefunden werden können. Darüber hinaus können noch Bedingungen über die Attribute der formalen Entitäten und Relationen formuliert werden, die ebenfalls erfüllt sein müssen.

Eine Bindung, deren Attributbedingungsteil erfüllt ist, wird als *gültige Bindung* bezeichnet. Eine Regel mit einer gültigen Bindung ist eine *Regelinstanz*.

Zu einer Regel kann es in der aktuellen Situation durchaus mehrere Regelinstanzen geben. So könnten zum Beispiel bei der vorgestellten Regel `EnhanceSpecification` mehrere Entwickler gleichzeitig an derselben Spezifikation arbeiten, wodurch es für jeden Entwickler eine eigene Regelinstanz gäbe.

Der Attributbedingungsteil wird durch das Schlüsselwort `constraints` eingeleitet. Zu dem oben vorgestellten Strukturbedingungsteil sieht der Attributbedingungsteil wie folgt aus:

```
constraints
  s.function_points < s.max_function_points;
```

Die Regel darf erst dann ausgeführt werden, wenn die gebundene Spezifikation `s` diese Bedingung erfüllt, d.h. wenn die tatsächlich enthaltenen Anforderungen (`function_points`) die Anzahl der Anforderungen des Kunden (`max_function_points`) noch nicht erreicht hat. Ansonsten gäbe es an der Spezifikation nichts mehr zu verbessern, die Ausführung der Regel wäre sinnlos.

Der Attributbedingungsteil darf auch Bedingungen enthalten, die eigentlich Aussagen über die Struktur machen. Hier spielen drei vordefinierte Funktionen eine Rolle, die jeweils Boolesche Werte liefern:

- Die Funktion `exists` erhält als Parameter ein Relationsliteral. Ein Relationsliteral wird notiert wie eine formale Relation. Es besteht aus einem Relationstyp und einer Rollenbelegung durch formale Entitäten. Wenn es eine Relation in der aktuellen Situation vom bezeichneten Relationstyp gibt, deren Rollen die angegebenen Werte besitzen, liefert `exists` wahr.  
Ein Beispiel für die Verwendung von `exists` findet sich im Benutzerkommando `Hire` im nächsten Abschnitt.
- Die Funktion `conforms` erhält als Parameter eine Entität und einen Entitätstyp bzw. eine Relation und einen Relationstyp. Sie prüft, ob der Typ der Entität bzw. Relation zu dem Entitätstyp bzw. dem Relationstyp typverträglich ist.

- Die Funktion `is_a` erhält dieselben Parameter wie `conforms`, prüft jedoch, ob der Typ der Entität bzw. Relation mit dem Entitätstyp bzw. dem Relationstyp identisch ist.

Mit Hilfe der Funktionen `conforms` und `is_a` können Bindungen aussortiert werden, die Instanzen nicht gewollter typverträglicher Typen enthalten. Sollen zum Beispiel für eine formale Entität `p` vom Typ `Person` alle Personen, die keine Entwickler sind, ausgewählt werden, läßt sich das mit der Bedingung

```
not conforms(p, Developer)
```

realisieren. Diese Bedingung sortiert alle Entitäten aus, die vom Typ `Developer` oder einem zu `Developer` verträglichen Typ sind.

### 4.3.1.3 Aktionsteil

Der Aktionsteil einer Regel gibt an, was bei einer Regelinstanz passieren soll, wenn sie ausgeführt wird. Er besteht aus einer Menge von Aktionen und ist ähnlich wie ein Ada-95-Block (vgl. Intermetrics, 1995, Abschnitt 5.6) aufgebaut. Zuerst können lokale Variable, deren Typ ein Attributtyp sein muß, deklariert werden. Diese müssen initialisiert werden, um Probleme mit uninitialisierten Variablen von vornherein auszuschließen.

An die Deklaration der lokalen Variablen schließt sich eine Folge von Befehlen an, die sich in die bereits in Abschnitt 2.3.1 eingeführten Kategorien gliedern lassen:

- Attributwerten verändern,
- neue Entitäten erzeugen,
- neue Relationen erzeugen,
- vorhandener Entitäten (einschließlich aller Verbindungen) entfernen oder
- vorhandene Relationen entfernen.

Hinzu kommt noch die Möglichkeit des Versendens von Nachrichten an den Benutzer, die mit der Prozedur `send_message` durchgeführt wird (vgl. dazu Abschnitt 3.3.3.4).

Die Veränderung von Attributwerten der formalen Entitäten und Relationen geschieht durch eine einfache Zuweisung des gewünschten neuen Werts. Der Wert kann sich dabei auch aus einem Ausdruck berechnen, muß jedoch denselben Typ haben wie das Attribut, dem er zugewiesen wird. Um zum Beispiel das Alter des Entwicklers `d` zu erhöhen, könnte man schreiben:

```
d.age := d.age + 1;
```

Die Erzeugung neuer Entitäten und Relationen geschieht mit der `create`-Anweisung, die in Abschnitt 4.4.1 und Abschnitt 4.4.2 vorgestellt werden wird.

Das Entfernen von Entitäten und Relationen wird mit der `delete`-Anweisung durchgeführt. Diese entfernt zusammen mit einer Entität automatisch alle Relationen, in denen die Entität in mindestens einer Rolle enthalten ist. Dies ist notwendig, weil – zumindest in SESAM-Modellen – eine Beziehung zu einer nicht existenten Entität nicht auftreten darf. Um zum Beispiel den Entwickler `d` zu entfernen, schreibt man:

```
delete(d);
```

Dann werden auch alle Relationen in der aktuellen Situation, die `d` enthalten, entfernt.

Aktionen können auch bedingt oder wiederholt ausgeführt werden. Dies wird durch die *if*- bzw. die *while*-Anweisung realisiert. Ihr Aussehen entspricht exakt dem Ada-95-Vorbild (vgl. Intermetrics, 1995, Abschnitte 5.3 und 5.5).

```
-- if-Anweisung:
if <Bedingung> then
    ...
else    -- Der else-Teil ist optional
    ...
end if;

-- while-Anweisung:
while <Bedingung> loop
    ...
end loop;
```

Der Aktionsteil der bereits teilweise vorgestellten Regel *EnhanceSpecification* wird nun im Gesamtzusammenhang präsentiert:

```
rule EnhanceSpecification[1000] taking 0 is
structure    -- Strukturbedingungsteil
  s: Specification;
  d: Developer;
  w: Specifies(d,s);
constraints  -- Attributbedingungsteil
  s.function_points < s.max_function_points;
-- Aktionsteil:
declare    -- Vereinbarung lokaler Variablen:
  new_function_points : Real :=
    (s.max_function_points - s.function_points)*0.1
    * d.experience * d.motivation * w.intensity;
begin    -- Aktionen
  s.function_points :=
    min(s.function_points + new_function_points,
        s.max_function_points);
end rule;
```

Zuerst wird eine lokale Variable *new\_function\_points* deklariert, die mit dem geplanten Zuwachs der beschriebenen Anforderungen des Kunden in der Spezifikation initialisiert wird. Dieser beträgt im günstigsten Fall 10% der fehlenden Anforderungen (daher der Faktor 0,1). Dieser Zuwachs kann jedoch noch durch mangelnde Erfahrung des Entwicklers, seine Motivation und die Intensität der Bearbeitung gemindert werden. Diese Faktoren sind jeweils Zahlen zwischen 0 und 1, wobei 1 das Optimum darstellt. Der so berechnete Zuwachs wird im Rumpf des Aktionsteils zu den bisherigen Anforderungen hinzugeschlagen. Dabei wird durch die *min*-Funktion sichergestellt, daß die maximale Anzahl der Anforderungen nicht überschritten wird. Die Funktion *min* ist eine benutzerdefinierte Funktion, die in Abschnitt 4.3.3 eingeführt wird.

### 4.3.2 Benutzerkommandos

Die Benutzerkommandos wurden in Abschnitt 3.3.3.1 bereits kurz vorgestellt. Benutzerkommandos dienen dazu, Anweisungen des Benutzers in Änderungen in der aktuellen Situation umzusetzen.

Benutzerkommandos sind vom Aufbau her den Regeln sehr ähnlich. Sie bestehen aus einem Kommandokopf, einem Bedingungsteil und einem Aktionsteil. Sie verfügen jedoch nicht über Prioritäten, da sie nur auf Anfrage des Benutzers hin ausgeführt werden und dann nicht mit anderen Regeln oder Benutzerkommandos konkurrieren müssen.

Benutzerkommandos können formale Parameter haben, die – analog zu formalen Parametern von Prozeduren – im Kommandokopf deklariert werden. Formale Parameter können von einem Entitätstyp oder von einem Basistyp sein. Über diese Parameter wird dem Benutzerkommando zusätzliche Information vom Benutzer mitgegeben.

Die formalen Parameter, die von einem Entitätstyp sind, können als bereits gebundene formale Entitäten des Strukturbedingungsteils aufgefaßt und auch als solche verwendet werden. Sie dürfen also zum Beispiel im Strukturbedingungsteil als Rollenbelegung formaler Relationen auftreten.

Da ein Benutzerkommando einen Strukturbedingungsteil besitzen kann, ist es je nach aktueller Situation möglich, mehrere verschiedene Bindungen eines Benutzerkommandos zu finden. Analog zu den Regeln spricht man von *Benutzerkommandoinstanzen*, wenn diese Bindungen gültig sind, d.h. die Attributbedingungen erfüllen. Ausgeführt werden alle Benutzerkommandoinstanzen, soweit das möglich ist (vgl. hierzu Abschnitt 4.5.2).

Ein Benutzerkommando `LetSpecify`, mit dem der Benutzer einen Entwickler dazu veranlassen kann, sich mit der Spezifikation zu beschäftigen, sieht so aus:

```
command LetSpecify(whom: Developer) taking 125 is
structure
  s: Specification;
  p: Project;
  m: IsMemberOf(whom,p);
constraints
  not exists(Specifies(whom,s));
begin
  create relation Specifies with
    who := whom;
    what := s;
    intensity := 1;
  end create;
  send_message( Specifies, whom );
end command;
```

Der Entwickler `whom` ist der formale Parameter dieses Benutzerkommandos. Zusätzlich wird im Strukturbedingungsteil die Spezifikation `s` und das Projekt `p` gebunden. Das Projekt `p` wird benötigt, um die `IsMemberOf`-Beziehung zwischen dem Entwickler und dem Projekt ebenfalls in den Strukturbedingungsteil aufnehmen zu können. Schließlich sollten nur Entwickler spezifizieren, die auch im Projekt mitarbeiten.

Im Attributbedingungsteil wird noch überprüft, ob der Entwickler nicht schon mit Spezifizieren beauftragt worden ist. Dies wird durch eine `Specifies`-Beziehung zwischen dem Entwickler und der Spezifikation ausgedrückt. Würde diese Kante existieren, bräuchte das Kommando

nicht ausgeführt werden. Die Nichtexistenz von Beziehungen kann, obwohl es sich eigentlich um eine Strukturbedingung handelt, nicht im Strukturbedingungsteil geprüft werden, da dort nur existente Entitäten und Relationen gebunden werden können.

Im Aktionsteil wird eine `Specifies`-Beziehung erzeugt, die den Entwickler als an der Spezifikation arbeitend ausweist. Danach wird mit `send_message` eine Nachricht an den Benutzer geschickt, daß der Entwickler `whom` die Arbeit an der Spezifikation aufgenommen hat; der Dolmetscher macht aus dieser Nachricht einen verständlichen natürlichsprachlichen Text.

### 4.3.3 Benutzerdefinierte Funktionen

Um komplizierte oder mehrfach auftretende Berechnungen kapseln zu können, wurde in BASE-2 die Möglichkeit vorgesehen, Funktionen definieren zu können. Diese haben attribbutypwertige Parameter und liefern ein Ergebnis von einem Attribbutyp. Sie sind nebenwirkungsfrei, da eine Manipulation der aktuellen Situation in einer Funktion nicht möglich ist.

Funktionen bestehen aus einem (eindeutigen) Namen, formalen Parametern, einem Rückgabetyt und dem Funktionsrumpf, in dem der Rückgabewert berechnet wird. Der Funktionsrumpf kann lokale Variable haben, die auch von einem Attribbutyp sein müssen. Das Aussehen einer solchen Funktion ähnelt sehr stark dem Ada-95-Vorbild. Als Beispiel wird die Funktion `min` vorgestellt, die von der Regel `EnhanceSpecification` in Abschnitt 4.3.1 in ihrem Aktionsteil verwendet wurde.

```
function min(a: Real, b: Real) return Real is
  -- returns the minimum of a and b
begin
  if a > b then
    return b;
  else
    return a;
  end if;
end function;
```

Die Funktion `min` erhält zwei Parameter `a` und `b` vom Typ `Real` und gibt ein Ergebnis vom Typ `Real` zurück. Im Rumpf wird anhand eines Vergleichs zwischen `a` und `b` entschieden, welcher von beiden der kleinere Wert ist und mit der `return`-Anweisung zurückgegeben wird.

Falls bei der Ausführung bis zum Erreichen des Endes einer Funktion keine `return`-Anweisung angetroffen wurde, wird ein Laufzeitfehler ausgelöst, da eine Funktion immer definiert sein, also einen Rückgabewert besitzen muß.

## 4.4 Situationsmodell (Startsituationsteil)

Der Startsituationsteil einer Modellbeschreibung in BASE-2 korrespondiert in etwa mit einem Situationsmodell. Dieses wird noch um weitere Initialisierungen ergänzt, die in Abschnitt 4.4.3 besprochen werden. Abschnitt 4.4.1 und Abschnitt 4.4.2 befassen sich mit der Erzeugung von Entitäten und Relationen. Die Startsituation, die aus einer Menge von Entitäten und Relationen besteht, wird nämlich durch eine Reihe von Erzeugungsanweisungen repräsentiert.

#### 4.4.1 Erzeugung von Entitäten

Um eine Entität erzeugen zu können, muß man den Typ der gewünschten Instanz und die Initialisierungswerte für *alle* Attribute angeben. Die Erzeugung einer Entität bedarf darüber hinaus der Angabe eines externen Namens. Unter diesem Namen ist die Entität nach außen bekannt. Der Dolmetscher verwendet diesen Namen in der Kommunikation mit dem Benutzer und mit der Basismaschine. Der Aufbau einer create-Anweisung für eine Entität wird am besten an einem Beispiel deutlich.

```
create entity John: Developer aka "John Bankmiller" with
  name := "John Bankmiller";
  age := 35;
  experience := 0.94;
  motivation := 0.5;
  cost_per_day := 600;
end create;
```

Diese create-Anweisung erzeugt einen Entwickler, auf den unter dem lokalen Bezeichner `John` zugegriffen werden kann. Der Entwickler ist nach außen unter dem externen Namen „John Bankmiller“ bekannt. Der externe Name wird in der `aka`<sup>1</sup>-Klausel vergeben. Der interne Name, der Wert des Attributs `name`, entspricht dem externen. John ist 35 Jahre alt, ziemlich erfahren, allerdings nur mittelmäßig motiviert, und aufgrund seiner Erfahrung auch recht teuer (600 DM pro Tag).

Der Bezeichner für die erzeugte Entität (im Beispiel: `John`) ist notwendig, um erzeugte Entitäten noch für die Initialisierung von Rollen später erzeugter Relationen verwenden zu können. Dies findet im Beispiel des folgenden Abschnitts statt.

#### 4.4.2 Erzeugung von Relationen

Die Erzeugung von Relationen geschieht weitgehend analog zu der von Entitäten. Hinzu kommt die Initialisierung *aller* Rollen durch bereits erzeugte Entitäten. Relationen besitzen keinen externen Namen, da sie keine eigenständigen Objekte der abstrakten Welt des Modell sind. Außerdem werden sie bei der Erzeugung keinem Bezeichner zugeordnet, da auf sie nicht mehr zugegriffen werden muß.

Das folgende Beispiel erzeugt für den Entwickler John Bankmiller die Projektzugehörigkeitsbeziehung in Form einer `IsMemberOf`-Relation.

```
create relation IsMemberOf with
  who := John;
  what := DasProjekt;
end create;
```

`DasProjekt` ist dabei der Bezeichner des ebenfalls bereits erzeugten Projekts (vgl. hierzu das Beispiel im folgenden Abschnitt).

---

1. `aka` ist eine vor allem im Internet gebräuchliche Abkürzung für „also known as“

### 4.4.3 Sonstige Initialisierungen

Zu Beginn des Startsituationsteils einer Modellbeschreibung werden noch andere Initialisierungen vorgenommen. Das sind der Anfangswert der Modellzeit (das Startdatum) in Form eines Date-Literals, die Simulationsschrittweite in Minuten und der Name des Wörterbuchs.

Bei dem folgenden Beispiel eines Startsituationsteils handelt es sich um eine gekürzte Version des Startsituationsteils des Beispielprogramms im Anhang. Es wurden nur drei Erzeugungsanweisungen aufgenommen. Die Modellzeit wird dem 20.11.1995, 8:00 Uhr initialisiert, die Simulationsschrittweite auf 60 Minuten gesetzt und als zu verwendendes Wörterbuch das Wörterbuch „demo“ festgelegt. Näheres zur Modellzeit und dem zugrundeliegenden Zeittyp Date kann Abschnitt 4.5.1 entnommen werden.

```
begin at 1995/11/20/08:00 timestep 60 using "demo"

  create entity DasProjekt: Project aka "ScheSch" with
    name := "ScheSch";
    customer := "Suebia";
    budget := 250000;
  end create;

  create entity John: Developer aka "John Bankmiller" with
    name := "John Bankmiller";
    age := 35;
    experience := 0.94;
    motivation := 0.5;
    cost_per_day := 600;
  end create;

  create relation IsMemberOf with
    who := John;
    what := DasProjekt;
  end create;

end model;
```

## 4.5 Modellausführung

In diesem Abschnitt wird beschrieben, wie die Ausführung von BASE-2-Modellen aussieht. Zuerst wird noch auf die Modellzeit eingegangen.

### 4.5.1 Modellzeit

Die Modellzeit wird im Animator intern verwaltet. Eine explizite Manipulation der Modellzeit durch Aktionen innerhalb des Modells ist, von der Initialisierung im Startsituationsteil abgesehen, nicht möglich. Die aktuelle Modellzeit kann innerhalb des Modells durch die vordefinierte Funktion `current_date` abgefragt werden.

Die Modellzeit ist ein Wert des Basistyps Date (vgl. Abschnitt 4.2.1.1). Fortgeschaltet wird die Modellzeit durch einen Simulationsschritt; die Größe der Erhöhung wird durch die Simulationsschrittweite bestimmt. Die Simulationsschrittweite wird im Startsituationsteil gesetzt und in Minuten angegeben.

## 4.5.2 Ausführungsalgorithmus

### 4.5.2.1 Aufgabe

Aufgabe des Ausführungsalgorithmus ist es, die dynamische Semantik eines BASE-2-Programms festzulegen. Wichtig ist in diesem Zusammenhang die Festlegung, wann welche Regeln in welcher Reihenfolge angewendet und wann Benutzerkommandos ausgeführt werden. Zudem muß die Auswirkung des Zeitverbrauchs von Regeln und Benutzerkommandos auf die Modellzeit definiert werden.

### 4.5.2.2 Informelle Darstellung

Auf eine präzise formale Darstellung des Algorithmus wurde hier zugunsten einer leichter verständlichen informellen Definition verzichtet. Dennoch wird mit formalen Bezeichnern als abkürzende Schreibweise gearbeitet.  $t$  bezeichne die Modellzeit,  $\Delta t$  die Simulationsschrittweite.  $c$  ist eine Zählgröße für verbrauchte Zeit, die noch nicht in Simulationsschritten umgesetzt worden ist. Ein Simulationsschritt ist immer dann notwendig, wenn  $c \geq \Delta t$  gilt, d.h. wenn der Simulationsbedarf  $c$  die Simulationsschrittweite erreicht oder übersteigt.

Abbildung 4.5 zeigt eine graphische Darstellung des Ausführungsalgorithmus als Ablaufdiagramm. Aktionen sind durch Rechtecke, bedingte Verzweigungen durch Rauten und Kontrollfluß durch Pfeile dargestellt.

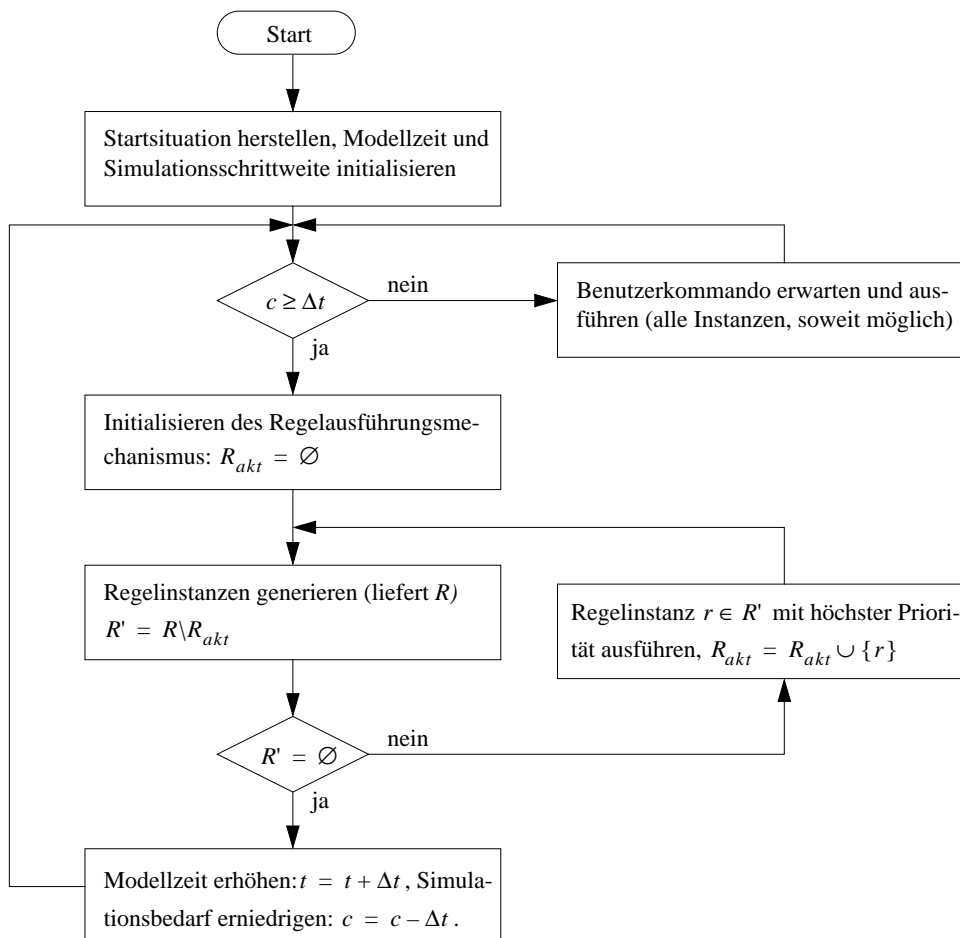


Abbildung 4.5: Graphische Darstellung des Ausführungsalgorithmus

**Startsituation herstellen (Modellinitialisierung).** Zuerst wird die Startsituation hergestellt, indem die create-Anweisungen des Startsituationsteils von oben nach unten ausgeführt werden. Die Modellzeit wird mit dem Startdatum initialisiert, die Simulationsschrittweite gesetzt und der Dolmetscher mit dem Wörterbuch gestartet. Der Simulationsbedarf  $c$  wird mit 0 initialisiert. Nach dieser Modellinitialisierung beginnt die eigentliche Modellausführung.

**Benutzerkommando erwarten.** Da zunächst kein Simulationsschritt erforderlich ist ( $c$  ist 0 und damit kleiner  $\Delta t$ ), wird auf ein Benutzerkommando gewartet. Trifft ein Kommando ein, werden alle möglichen Instanzen anhand der aktuellen Situation erzeugt und in der Reihenfolge der Generierung ausgeführt – sofern der Bedingungsteil zum Zeitpunkt der Ausführung noch erfüllt ist. Dies muß nicht unbedingt der Fall sein, da die Ausführung einer Benutzerkommandoinstanz Veränderungen in der aktuellen Situation vornimmt, die sich auf die Gültigkeit anderer Instanzen auswirken können.

Für jede ausgeführte Benutzerkommandoinstanz wird der Simulationsbedarf  $c$  um den Zeitverbrauch des Benutzerkommandos erhöht. Solange der Simulationsbedarf am Ende der Ausführung eines Benutzerkommandos die Simulationsschrittweite nicht erreicht, wird auf ein weiteres Benutzerkommando gewartet.

**Initialisieren des Regelausführungsmechanismus.** Ist der Simulationsbedarf groß genug, wird ein Simulationsschritt durchgeführt. In einem Simulationsschritt werden so lange Regeln ausgeführt, bis ein stabiler Zustand erreicht wurde, in dem keine weitere Regel mehr feuern kann. Dies wird durch die innere Schleife realisiert. Zuerst wird die Menge der bereits ausgeführten Regelinstanzen  $R_{akt}$  mit der leeren Menge initialisiert. Die Speicherung bereits ausgeführter Regelinstanzen dient dazu, die mehrfache Ausführung derselben Regelinstanz in einem Simulationsschritt zu verhindern. Andernfalls kann nicht garantiert werden, daß die Regelausführung terminiert, da sich zum Beispiel zwei Regelinstanzen immer gegenseitig die Voraussetzungen für ein Feuern schaffen könnten.

**Regelinstanzen generieren.** Anschließend werden auf der Grundlage der aktuellen Situation alle möglichen Regelinstanzen  $R$  generiert und von diesen alle diejenigen entfernt, die bereits ausgeführt wurden. Übrig bleibt die Menge  $R' = R \setminus R_{akt}$ . Ist diese Menge leer, konnte keine neue Regelinstanz mehr gefunden werden, der Regelausführungsmechanismus terminiert. Ist  $R'$  nicht leer, wird die Regelinstanz mit der höchsten Priorität ausgewählt. Bei Regeln gleicher Priorität entscheidet der Name der Regel über die Reihenfolge der Ausführung: die Regel mit dem lexikographisch kleineren Namen wird bevorzugt. Innerhalb von Instanzen derselben Regel entscheidet die Reihenfolge der Generierung über die Ausführungsreihenfolge.

**Regelinstanz ausführen.** Die ausgewählte Regelinstanz  $r$  wird ausgeführt, d.h. die Befehle des Aktionsteils werden sequentiell abgearbeitet. Anschließend wird  $r$  der Menge der ausgeführten Regelinstanzen hinzugefügt ( $R_{akt} = R_{akt} \cup \{r\}$ ) und der Simulationsbedarf wird um den Zeitverbrauch der Regel erhöht. Dann wird die nächste auszuführende Regelinstanz bestimmt, indem  $R'$  neu berechnet wird.

Es wäre auch denkbar, alle Instanzen aus  $R'$  in der Reihenfolge ihrer Prioritäten auszuführen (sofern der Bedingungsteil noch erfüllt ist), und erst danach  $R'$  neu zu berechnen. In diesem Fall könnte es jedoch passieren, daß eine Regelinstanz mit niedrigerer Priorität ausgeführt wird, obwohl auch eine Regelinstanz mit höherer Priorität ausgeführt werden kann. Dies ist dann der

Fall, wenn eine Regelinstanz bei ihrer Ausführung die Voraussetzungen für die Ausführung einer anderen Regelinstanz schafft. Aus diesem Grund wird die Berechnung der Regelinstanzen nach der Ausführung einer einzigen Regelinstanz von neuem ausgeführt.

**Modellzeit fortschalten.** Ist die Regelausführung abgeschlossen, wird der Simulationsschritt durch ein Weiterschalten der Simulationszeit  $t$  um einen Simulationsschritt und eine Verringerung des Simulationsbedarfs  $c$  um einen Simulationsschritt abgeschlossen.

### 4.5.3 Beispiel einer Modellausführung

Der Ausführungsalgorithmus soll nun an einem kleinen Beispiel verdeutlicht werden. Das Modell beinhalte die vorgestellte Regel `EnhanceSpecification` und das Benutzerkommando `LetSpecify`. Hinzu kommen die Regel `DecreaseBudget`, die die täglichen Kosten für einen Entwickler vom Projektbudget abzieht und ein Benutzerkommando `Hire`, das dazu dient, einen Entwickler einzustellen.

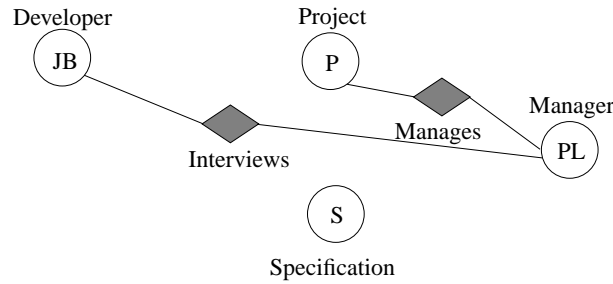
```
rule DecreaseBudget[0] taking 0 is
structure
  p: Project;
  d: Developer;
  m: IsMemberOf(d,p);
begin
  p.budget := p.budget - d.cost_per_day/24;
  -- Faktor 1/24 zur Umrechnung von Tagen auf Stunden
end rule;

command Hire(whom: Developer) taking 60 is structure
  p: Manager;
  j: Project;
  t: Interviews(p,whom);
  m: Manages(p,j);
constraints
  not exists(IsMemberOf(whom,j));
begin
  create relation IsMemberOf with
    who := whom;
    what := j;
  end create;
  send_message( Hired, whom );
  delete(t);
end command;
```

#### 4.5.3.1 Ausgangssituation

Die Ausgangssituation sei wie in Abbildung 4.6 dargestellt.

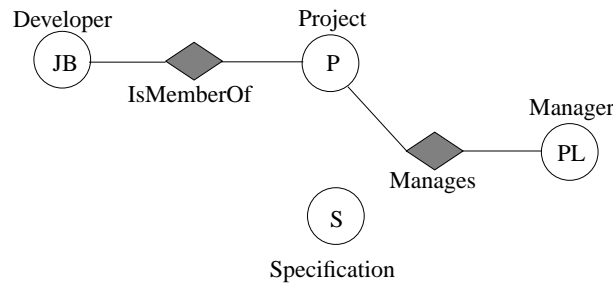
Der bereits bekannte Entwickler John Bankmiller (hier als JB abgekürzt) ist bisher nicht Mitglied des Projekts (P), befindet sich aber bereits mit dem Projektleiter (PL) in einem Bewerbungsgespräch (ausgedrückt durch die `Interviews`-Beziehung). Die Spezifikation (S) wird bisher nicht bearbeitet.



**Abbildung 4.6: Ausgangssituation des Beispiels**

#### 4.5.3.2 Erster Simulationsschritt

Durch die Ausführung des Benutzerkommandos `Hire` wird John ins Projekt eingestellt. Dies wird durch die neue `IsMemberOf`-Beziehung repräsentiert. Gleichzeitig wird die `Interviews`-Beziehung entfernt. Die neue Situation ist in Abbildung 4.7 dargestellt.



**Abbildung 4.7: Situation nach der Einstellung**

Durch die Ausführung der Benutzerkommandoinstanz mit John als einzustellendem Entwickler wird der Zeitverbrauch des Benutzerkommandos, in diesem Fall 60 Minuten, wirksam. Der Simulationsbedarf  $c$  beträgt nun 60 Minuten. Die Simulationsschrittweite beträgt ebenfalls 60 Minuten. Deshalb ist jetzt ein Simulationsschritt erforderlich.

Die Menge der bereits ausgeführten Regelinstanzen  $R_{akt}$  wird mit der leeren Menge initialisiert. Wie man an der Situation in Abbildung 4.7 sieht, kann die Regel `EnhanceSpecification` jedoch nicht gebunden werden, da es keine `Specifies`-Beziehungen gibt. Von der Regel `DecreaseBudget` kann eine Instanz gebildet werden, da John nun Projektmitglied ist. Diese Instanz wird ausgeführt, da sie als einzige Instanz auch die Instanz mit der höchsten Priorität ist. Sie wird in  $R_{akt}$  gespeichert und ihr Zeitbedarf von 0 Minuten zu  $c$  hinzugezählt. Abbildung 4.8a zeigt die Menge  $R'$ , nachdem zum ersten Mal alle Regelinstanzen ermittelt wurden. Regelinstanzen werden hier durch einen Kasten repräsentiert, dessen erster Teil den Regelnamen, der zweite die Priorität und der dritte die Bindung des Strukturteils enthält.

Anschließend werden wieder alle möglichen Regelinstanzen erzeugt. Es kommen aber keine neuen Instanzen hinzu. Die Menge der neuen Benutzerkommandos  $R'$  ist deshalb leer (siehe Abbildung 4.8b). Der Regelausführungsmechanismus wird daraufhin abgebrochen. Die Modellzeit wird erhöht und der Simulationsbedarf  $c$  um einen Simulationsschritt erniedrigt, er beträgt nun wieder 0 Minuten. Deshalb wird kein weiterer Durchlauf durch den Regelausführungsmechanismus vorgenommen, sondern wieder auf ein Benutzerkommando gewartet.

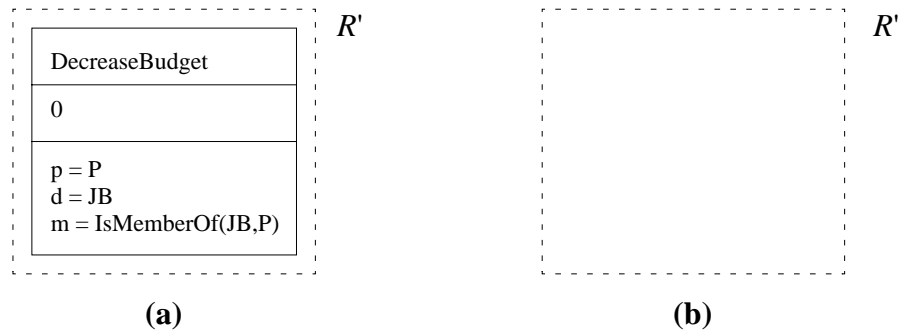


Abbildung 4.8: Regelinstanzen im ersten Simulationsschritt

### 4.5.3.3 Zweiter Simulationsschritt

Jetzt trifft das Kommando `LetSpecify` mit dem Parameter John Bankmiller ein. Dieses Benutzerkommando stellt eine `Specifies`-Beziehung zwischen der Spezifikation und dem Entwickler her. Das Ergebnis sieht man in Abbildung 4.9.

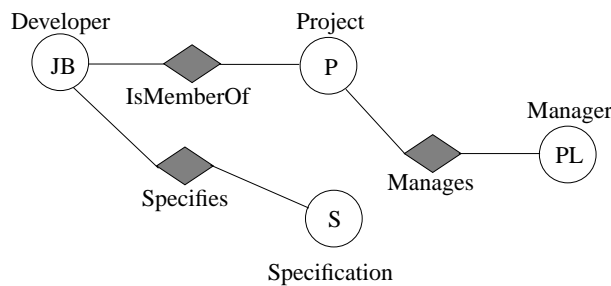


Abbildung 4.9: Situation nach Ausführung von `LetSpecify`

Das Benutzerkommando `LetSpecify` hat einen Zeitverbrauch von 125 Minuten. Der Simulationsbedarf  $c$  erhöht sich also auf 125 Minuten, weshalb ein Simulationsschritt durchgeführt werden muß.

Jetzt kann die Regel `EnhanceSpecification` gebunden werden. Es ergibt sich eine Regelinstanz mit John als Entwickler. Außerdem gibt es wieder dieselbe Instanz von `DecreaseBudget` wie im letzten Simulationsschritt. Die beiden Instanzen sind in Abbildung 4.10a dargestellt. Die Instanz von `EnhanceSpecification` hat die höchste Priorität und wird ausgeführt. Anschließend wird sie in  $R_{akt}$  gespeichert. Da der Zeitbedarf der Regel 0 ist, wird  $c$  nicht verändert.

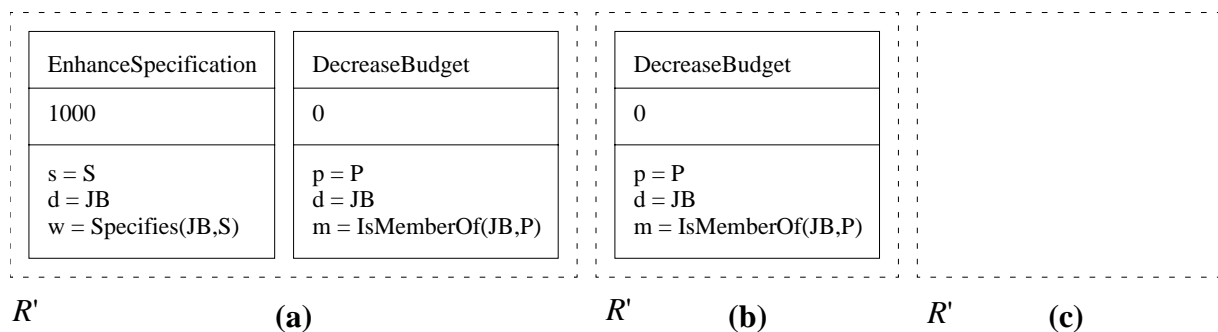


Abbildung 4.10: Regelinstanzen im zweiten Simulationsschritt

Die folgende Suche nach Regelinstanzen liefert dieselben Instanzen wie bei der vorhergehenden. Da die Instanz von `EnhanceSpecification` bereits ausgeführt wurde, ist diesmal die Instanz von `DecreaseBudget` die einzige (Abbildung 4.10b). Diese Instanz wird – wie im ersten Simulationsschritt – ausgeführt. Anschließend gibt es keine neuen Instanzen mehr, so daß der Regelausführungsmechanismus abgebrochen wird (Abbildung 4.10c).

Die Simulationszeit wird weitergeschaltet und  $c$  auf 65 Minuten reduziert. Dieser Wert macht einen erneuten Simulationsschritt erforderlich, der dasselbe tut wie der gerade beschriebene. Danach wird wieder auf das nächste Benutzerkommando gewartet, da der restliche Simulationsbedarf von 5 Minuten kleiner als die Simulationsschrittweite ist.

## 4.6 Vergleich mit SESAM-1 und SESAM-Lite

In diesem Abschnitt werden wichtige Unterschiede zwischen den Konzepten von BASE-2 und den Modellierungssprachen der Vorgängersysteme SESAM-1 und SESAM-Lite herausgearbeitet. Es wird dabei eine Grundkenntnis dieser Systeme vorausgesetzt.

### 4.6.1 Entitäts- und Relationstypen

Abstrakte Entitätstypen und Relationstypen gibt es auf der Ebene der Basissprache nicht, dieses Konzept kann jedoch problemlos auf Hochsprachenebene eingeführt werden. Abstrakte Typen der Hochsprachen werden dann auf normale Typen in BASE-2 abgebildet.

Die Möglichkeit zur Deklaration von Default-Werten bei der Entitätstypdeklaration ist auf Basissprachen-Ebene nicht eingeführt worden, da dies lediglich der Bequemlichkeit dient und auf Hochsprachenebene leicht realisiert werden kann. Außerdem können in Relationstypen keine Default-Werte für Rollen angegeben werden, da es im Schemamodell keine Entitäten gibt, die man zur Initialisierung verwenden könnte. Auf diese Weise entsteht ein unästhetisches Ungleichgewicht zwischen Rollen ohne Default-Werten und Attributen mit Default-Werten.

### 4.6.2 Events

In SESAM-1 und in SESAM-Lite gibt es neben den Entitätstypen und Relationstypen im Schemamodell noch die Eventtypen. Diese beschreiben Events (Ereignisse, vgl. hierzu Schneider, 1994, Abschnitt 4.3.3). Es werden exogene Events und endogene Events unterschieden.

Exogene Events sind Ereignisse, die außerhalb des Modells erzeugt werden und dann auf das Modell einwirken. Sie dienen dazu, die Auswirkungen von Benutzeraktionen zu modellieren. In BASE-2 werden die exogenen Events durch die Benutzerkommandos ersetzt.

Endogene Events sind Ereignisse, die im Modell selbst erzeugt werden. Sie dienen vor allem dazu, Ereignisse, die irgendwann in der Zukunft eintreten sollen, zu modellieren. Mit Hilfe von endogenen Events lassen sich Effekte erzeugen, die dem Grundsatz, daß der Folgezustand des Modells sich direkt aus dem Vorgängerzustand berechnet, zuwiderlaufen.

Endogene Events werden häufig dazu herangezogen, eine Art Kontrollfluß zwischen zwei Regeln zu modellieren: Wenn eine Regel ausgeführt wird, soll das dazu führen, daß eine zweite Regel ausgeführt wird. Das läßt sich bequem durch das Absetzen eines Events in der ersten Regel und das Warten auf diesen Event in der zweiten Regel realisieren.

Die durch den Versand und den Empfang von Events definierte Reihenfolge der Ausführung kann in der Basissprache durch Prioritäten realisiert werden. Je höher die Priorität einer Regel ist, desto früher wird sie bei der Regelausführung berücksichtigt und kann die nötigen Voraussetzungen schaffen, die es einer weiteren Regel mit niedrigerer Priorität erlauben, innerhalb desselben Simulationsschrittes ausgeführt zu werden.

### 4.6.3 Regeln

Regeln in BASE-2 unterscheiden sich in zwei Aspekten von denen in SESAM-1. Wie schon in Abschnitt 2.4 behandelt wurde, war die Regelausführungssemantik in SESAM-1 die Aktivierungs-/Deaktivierungssemantik. Bereits in SESAM-Lite war zu einer Feuernsemantik übergegangen worden, da es sich um das einfachere Konzept handelt. BASE-2 hat die Feuernsemantik übernommen.

Auch der zweite Unterschied wurde bereits in SESAM-Lite eingeführt: die Prioritäten. Auf die Prioritäten wurde bereits im obigen Abschnitt in Zusammenhang mit endogenen Events eingegangen.

### 4.6.4 Externe Namen von Entitäten

In SESAM-1 wurde der externe Name durch die Vergabe eines Attributs `name` an alle Entitätstypen realisiert. Der Wert dieses Attributs war dann der externe Name. Dieses Konzept stellte jedoch eher eine Durchmischung des Attributkonzepts mit dem Konzepts des externen Namens dar, weshalb die beiden Konzepte in BASE-2 strikt getrennt wurden.

SESAM-Lite arbeitet mit sogenannten Schlüsselattributen. In jedem Entitätstyp wird ein Attribut als Schlüsselattribut gekennzeichnet, der Wert dieses Attributs wird dann als externer Name verwendet. Zum Beispiel würde man für den Entitätstyp `Developer` als Schlüsselattribut das Attribut `name` wählen. Ein solches als Schlüsselattribut geeignetes Attribut muß es jedoch nicht bei jedem Entitätstyp von vornherein geben, so daß schließlich in BASE-2 das Konzept der expliziten Vergabe externer Namen eingeführt wurde.

### 4.6.5 Modellzeit

Die Modellzeit in BASE-2 beruht auf dem Basistyp `Date`, der einen 24-Stunden-Tag hat. In Software-Projekten wird in der Regel nicht rund um die Uhr gearbeitet, so daß sich die Simulation eigentlich auf die Arbeitszeiten beschränken sollte. Dies wurde in SESAM-1 so realisiert, daß ein Tag nur acht Stunden hatte. Eine feste Vorgabe von Arbeitszeiten durch eine entsprechende Definition des Zeittyps (z.B. von 8:00 bis 16:00 Uhr) erscheint jedoch nicht sinnvoll, da eine Modellierung von Überstunden in diesem Fall nicht direkt möglich ist. Deshalb wurde der Zeittyp in BASE-2 nicht beschränkt. Die Arbeitszeiten müssen durch den Modellierer umgesetzt werden, indem er zum Beispiel Entwickler als an- bzw. abwesend kennzeichnet und ihr Anwesenheitsverhalten modelliert.

Eine direkte Manipulation der Modellzeit – in diesem Fall ein Weiterschalten bis zum nächsten Morgen – könnte hier sinnvoll sein, da auch der Projektleiter nicht rund um die Uhr arbeiten und daher nicht zu jeder Zeit Aktivitäten ausführen kann. Dies wird durch den momentan definierten Ausführungsmechanismus eines BASE-2-Modells jedoch nicht berücksichtigt. Die Idee der Modellzeitmanipulation wird in Abschnitt 6.2.1.3 wieder aufgegriffen.

## 5 Realisierung der Basismaschine

*Was ihr macht, das macht recht; es kostet nicht mehr Zeit, als ihr dazu braucht, um es schlecht zu machen.*

*(Johann Jakob Sulzer)*

In diesem Kapitel wird die im Rahmen der Arbeit durchgeführte Realisierung der Basismaschine beschrieben. Zuerst wird in Abschnitt 5.1 auf den Entwurf der Basismaschine eingegangen. Daran schließt sich in Abschnitt 5.2 die auf den Entwurf aufbauende Implementierung an, die allerdings nur Teile des Entwurfs umfaßt. Abschließend wird in Abschnitt 5.3 ein kleiner Beispiellauf der Basismaschine gezeigt.

### 5.1 Entwurf

Im Entwurf der Basismaschine ging es vor allem darum, eine passende Architektur für das Programm zu erstellen, das ein BASE-2-Modell einlesen und ausführen kann. Wie bereits in Abbildung 3.1 angedeutet wurde, kann man die Aufgabe der Basismaschine in zwei Teile teilen:

- Zuerst muß das BASE-2-Modell eingelesen und auf seine statische Korrektheit überprüft werden. Diese Aufgabe fällt dem *Parser* zu.
- Anschließend muß das Modell ausgeführt werden, bis der Benutzer einen Abbruch der Simulation veranlaßt oder ein Laufzeitfehler auftritt. Diese Aufgabe übernimmt der *Animator*. Der Animator bedient sich bei der Modellausführung des Dolmetschers als Benutzeroberfläche.

Im Laufe des Entwurfs wurden zwei verschiedene Ansätze zur Realisierung dieser Aufgaben entwickelt. Diese heißen „Interpreter-Ansatz“ und „Code-Generator-Ansatz“. Der Code-Generator-Ansatz wurde für die Umsetzung in der Implementierung ausgewählt.

In Abschnitt 5.1.1 wird der Interpreter-Ansatz vorgestellt, in Abschnitt 5.1.2 folgt der Code-Generator-Ansatz. Die beiden Ansätze werden anschließend in Abschnitt 5.1.3 verglichen und die Auswahl des Code-Generator-Ansatzes begründet.

Abschnitt 5.1.4 enthält eine Beschreibung des gewählten Regelausführungsalgorithmus. Dabei liegt der Schwerpunkt auf dem Algorithmus zur Bestimmung von Regel- und Benutzerkommandoinstanzen, weil dieser durch die Sprachbeschreibung von BASE-2 nicht festgelegt ist.

#### 5.1.1 Interpreter-Ansatz

Beim Interpreter-Ansatz wird die Basismaschine als ausführender Übersetzer für die Basissprache realisiert. Ein solcher Übersetzer, zum Beispiel für die Programmiersprache Basic<sup>1</sup>, wird als Interpreter bezeichnet. Im Unterschied zu einem imperativen Basic-Programm, das zeilenweise eingelesen und ausgeführt werden kann, ist es bei einem regelbasierten BASE-2-Programm erforderlich, zuerst das gesamte Programm einzulesen. Erst dann stehen alle

---

1. Basic ist eine Abkürzung für „Beginners' All purpose Symbolic Instruction Code“. Basic ist eine imperative Sprache, die bereits 1965 entwickelt wurde und eine weite Verbreitung gefunden hat. Zur Verarbeitung von Basic-Programmen werden vornehmlich Interpreter eingesetzt (Duden Informatik, 1993). Näheres zu Basic kann Kemeny, Kurtz (1985) entnommen werden.

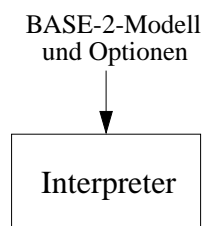
erforderlichen Informationen zur Programmausführung zur Verfügung. So kann zum Beispiel die Startsituation erst dann hergestellt werden, wenn das Schemamodell bekannt ist. Andernfalls kann nicht geprüft werden, ob alle Attribute und Rollen korrekt initialisiert wurden.

### 5.1.1.1 Arbeitsweise

Ein BASE-2-Programm muß als Textdatei vorliegen, um vom Interpreter verarbeitet werden zu können. Der Interpreter erhält als Eingabe den Namen dieser Datei und einige Optionen, mit denen die Art der Verarbeitung beeinflusst werden kann. So gibt es zum Beispiel eine sogenannte Debug-Option, die den Interpreter veranlaßt, seine Verarbeitungsschritte an den Benutzer zu melden.

Der Interpreter verarbeitet seine Eingaben, die sogenannten Argumente, liest das Programm aus der Datei vollständig ein, speichert die notwendigen Informationen über das Modell und führt dann das Modell aus.

Abbildung 5.1 zeigt eine graphische Darstellung des Interpreter-Ansatzes. Der Interpreter erhält als Eingabe ein BASE-2-Modell sowie mögliche Optionen. Eventuelle Fehlermeldungen des Parsers sowie die zur Modellausführung erforderliche Kommunikation mit dem Dolmetscher wurden in dieser Darstellung weggelassen.



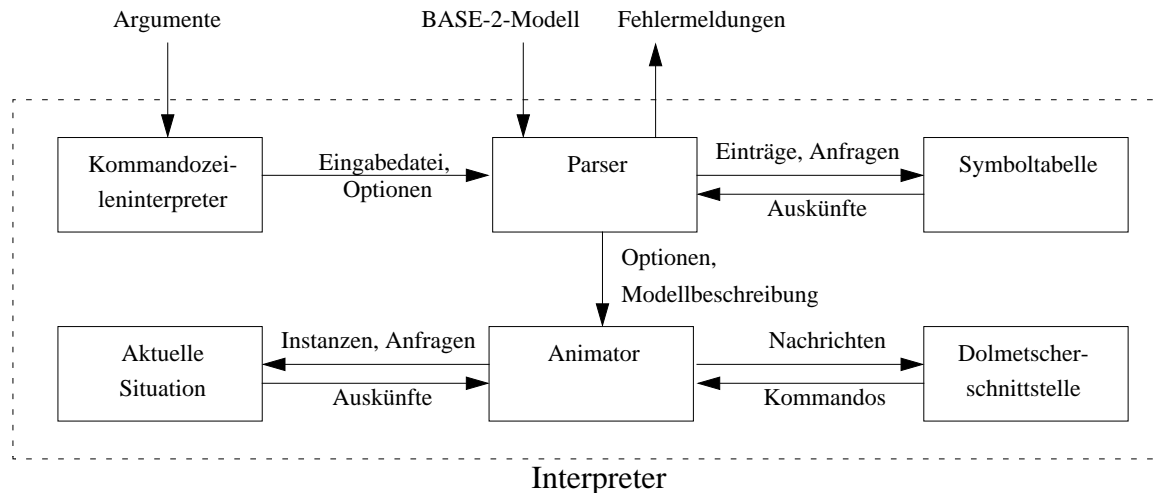
**Abbildung 5.1: Arbeitsweise des Interpreteransatzes**

### 5.1.1.2 Architektur

Der Interpreter wird nun in seine Hauptbestandteile zerlegt. Die erforderlichen Komponenten zur Datenhaltung und zur Verarbeitung werden identifiziert und ihre Schnittstellen festgelegt. Abbildung 5.2 zeigt die sich ergebende Architektur des Interpreters.

In diesem Bild werden die Komponenten zur Datenhaltung und zur Verarbeitung durch Rechtecke dargestellt, Datenflüsse zwischen den Komponenten durch beschriftete Pfeile in der Richtung des Datenflusses.

Die Argumente werden von einem *Kommandozeileninterpreter* analysiert und in aufbereiteter Form an den Parser weitergereicht. Der *Parser* öffnet die Eingabedatei und liest das BASE-2-Modell ein. Falls bei der Programmanalyse ein Fehler gefunden wird, meldet er ihn an den Benutzer. Für die einzelnen Definitionen des Modells erzeugt der Parser Symboltabelleneinträge, die einer Verwaltungskomponente, der *Symboltabelle*, übergeben werden. Für die Durchführung semantischer Prüfungen richtet der Parser dann Anfragen an die Symboltabelle, die ihm die gewünschten Auskünfte liefert. Eine solche Auskunft kann zum Beispiel sein, ob ein bestimmter Entitätstyp bereits definiert wurde, oder welchen Typ ein bestimmtes Attribut eines Relationstyps hat.



**Abbildung 5.2: Architektur des Interpreters**

Ist das Modell aus Sicht des Parsers fehlerfrei, übergibt er die Optionen und eine Modellbeschreibung dem Animator. Die Modellbeschreibung besteht aus beschreibenden Datenstrukturen für die Definitionen und den Startsituationsteil.

Der *Animator* erzeugt zuerst die Startsituation, indem er den Startsituationsteil ausführt und die Instanzen der Entitäts- und Relationstypen einer Verwaltungskomponente, der *aktuellen Situation*, übergibt. Diese verwaltet auch die Modellzeit. Der Name des Wörterbuchs wird der Dolmeterschnittstelle übergeben. Die *Dolmeterschnittstelle* startet daraufhin den Dolmetscher, nimmt von ihm Kommandos entgegen und leitet Nachrichten des Modells an ihn weiter.

Während der Modellsimulation erzeugt der Animator anhand der Regelbeschreibungen und den Auskünften der aktuellen Situation die Regelinstanzen und führt sie aus. Bei Eingang eines Benutzerkommandos generiert er ebenfalls in Zusammenarbeit mit der aktuellen Situation alle möglichen Instanzen dieses Benutzerkommandos und führt die Instanzen aus.

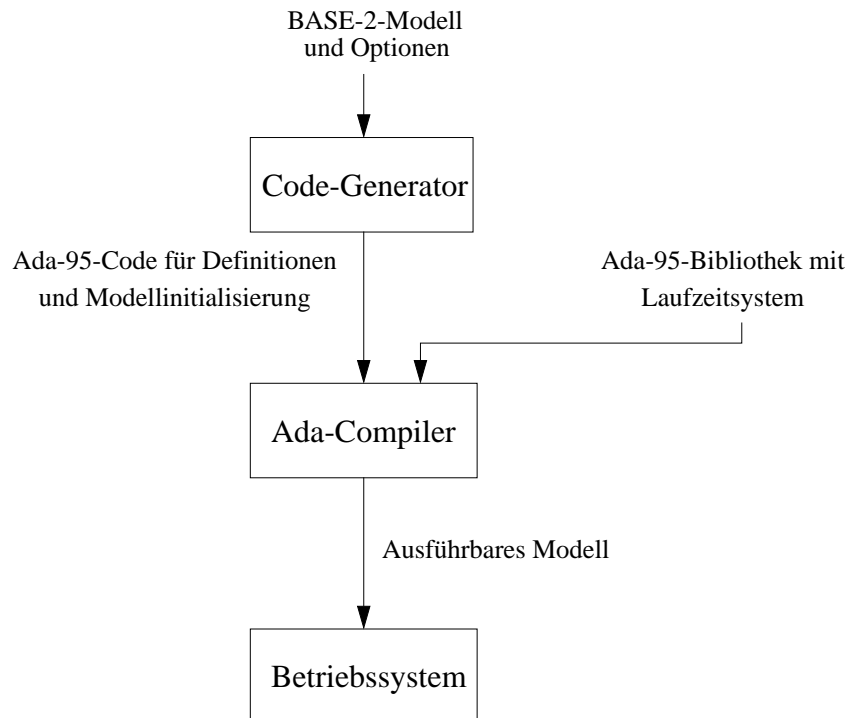
### 5.1.2 Code-Generator-Ansatz

Der Code-Generator-Ansatz geht im Vergleich zum Interpreteransatz anders vor. Der Parser generiert aus der Modellbeschreibung Ada-95-Code, der anschließend kompiliert und ausgeführt werden kann. Auf diese Weise erhält man ein ausführbares Programm, das die Modellanimation realisiert. Da dieses Vorgehen auch der Arbeitsweise eines Compilers entspricht, der eine Sprache in eine andere übersetzt, könnte man auch von einem Compiler-Ansatz sprechen. Schließlich werden Programme in Basissprache in Ada 95 übersetzt.

#### 5.1.2.1 Arbeitsweise

Abbildung 5.3 zeigt eine schematische Übersicht der Arbeitsweise des Code-Generator-Ansatzes.

Das BASE-2-Modell wird von einem *Code-Generator* in Ada-95-Code übersetzt. Dieser Code wird zusammen mit einer Ada-95-Bibliothek übersetzt, die das Laufzeitsystem des Modells enthält. Das Laufzeitsystem umfaßt alle notwendigen Komponenten zur Modellausführung. Das



**Abbildung 5.3: Arbeitsweise des Code-Generator-Ansatzes**

Ergebnis der Übersetzung durch den *Ada-Compiler* ist ein ausführbares Programm, das durch das *Betriebssystem* ausgeführt werden kann. Das ausführbare Modell bedient sich bei der Ausführung des Dolmetschers, der in dieser Darstellung (wie auch in Abbildung 5.1) weggelassen wurde.

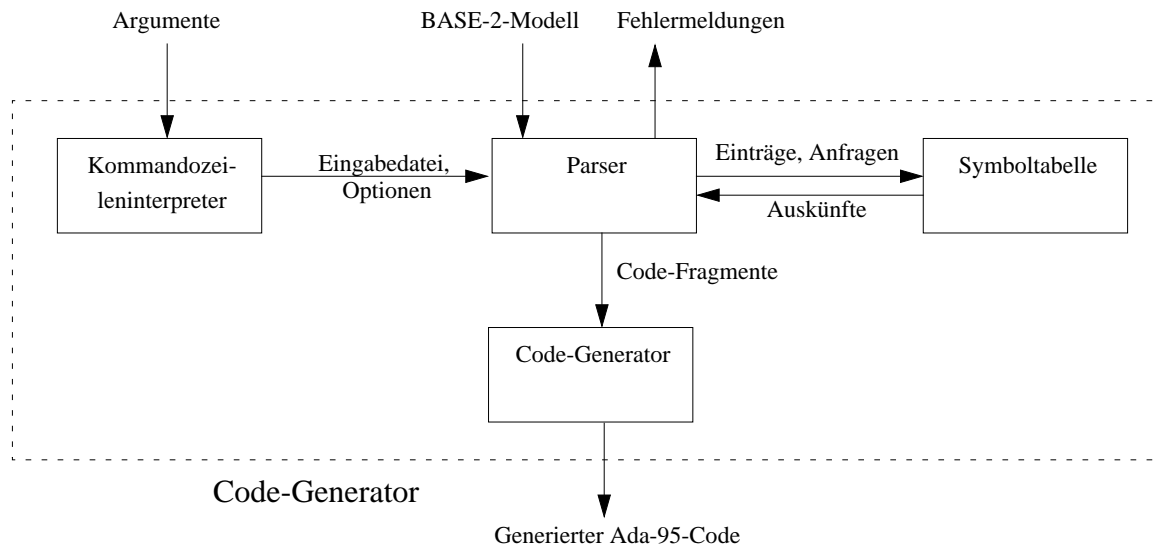
Theoretisch wäre es denkbar, den gesamten Ada-95-Code für das ausführbare Modell vom Code-Generator erzeugen zu lassen. Dies hätte jedoch die Komplexität der Implementierung des Code-Generators stark erhöht. Zudem wäre die Übersetzungszeit für den Ada-Code höher. Durch die Ablage des Laufzeitsystems in einer Bibliothek kann es vorkompiliert werden und muß bei der Übersetzung des generierten Codes nur noch hinzugebunden werden.

Die Trennung des für das Modell spezifisch generierten Codes und des allgemeinen Laufzeitsystems macht jedoch eine vermittelnde Schnittstelle zwischen den beiden Teilen erforderlich. Auf die Aufgaben dieser Schnittstelle, des sogenannten *Generated Code Interface*, wird bei der Beschreibung der Architektur des ausführbaren Modells im folgenden Abschnitt genauer eingegangen.

### 5.1.2.2 Architektur

Die Architektur des Code-Generator-Ansatzes zerfällt in zwei Teile. Der erste Teil umfaßt den Code-Generator, der zweite das ausführbare Modell.

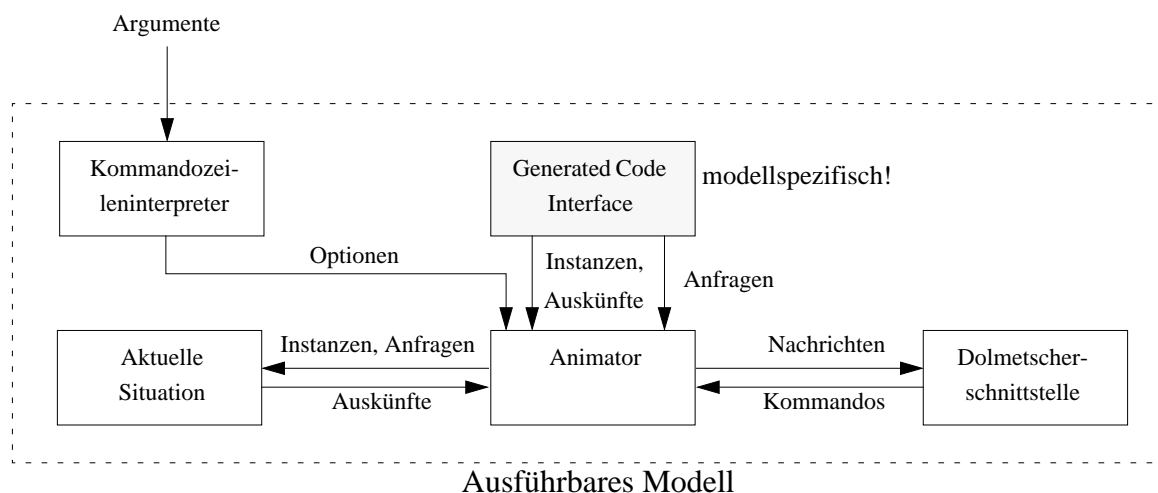
**Code-Generator.** Der Code-Generator übernimmt ähnliche Aufgaben wie die obere Ebene der Architektur des Interpreter-Ansatzes in Abbildung 5.2. Die Architektur des Code-Generators ist in Abbildung 5.4 schematisch dargestellt.



**Abbildung 5.4: Architektur des Code-Generators**

Aus der Architektur des Interpreter-Ansatzes übernommen wurden die drei Komponenten Kommandozeileninterpreter, Parser und Symboltabelle. Der *Parser* erzeugt jedoch keine interne Modellbeschreibung, sondern übergibt Code-Fragmente für die einzelnen Definitionen und den Startsituationsteil an die Code-Generator-Komponente. Für jede Definition des Modells wird ein Ada-95-Typ erzeugt, für den Startsituationsteil eine Folge von Initialisierungsbefehlen. Der *Code-Generator* bettet diese Code-Fragmente in passende Ada-95-Module<sup>1</sup> ein und schreibt diese Module in Dateien. Außerdem erzeugt er die Schnittstelle zum generierten Code, das Generated Code Interface, die ebenfalls in eine Datei geschrieben wird.

**Ausführbares Modell.** Der vom Code-Generator erzeugte modellspezifische Code wird mit dem Laufzeitsystem zusammengebunden. Dadurch ergibt sich ein ausführbares Modell, dessen Architektur Abbildung 5.5 zeigt.



**Abbildung 5.5: Architektur des ausführbaren Modells**

1. Für jede Definitionsart wird ein Modul mit festem Namen erzeugt. Die Initialisierungsbefehle werden in einer Initialisierungsprozedur festen Namens zusammengefaßt.

Da dem ausführbaren Modell Optionen übergeben werden können, verfügt es ebenfalls über einen *Kommandozeileninterpreter*. Dieser gibt die Optionen weiter an den Animator. Die Ebene des Animators in diesem Architekturbild ähnelt wiederum der analogen Ebene in der Architektur des Interpreter-Ansatzes in Abbildung 5.2. Die Komponenten aktuelle Situation, Animator und Dolmeterschnittstelle kommen auch hier vor und übernehmen dieselben Aufgaben wie beim Interpreter-Ansatz.

Der *Animator* benötigt zur Ausführung eines Benutzerkommandos und zur Anwendung der Regeln in einem Simulationsschritt Informationen über die Benutzerkommandos und die Regeln. Diese Auskünfte liefert ihm das *Generated Code Interface* (GCI) in Form von Deskriptoren. Das GCI wird modellspezifisch durch den Code-Generator erzeugt, weshalb es in der Abbildung hervorgehoben ist. Zur Erzeugung von Regel- und Benutzerkommandoinstanzen übergibt der Animator eine Beschreibung der Instanz an das GCI, das ihm daraufhin eine echte Instanz zurückliefert. Diese Instanz kann dann auf die Gültigkeit ihrer Bindung untersucht und gegebenenfalls ausgeführt werden kann.

Weil der aus den Definitionen generierte Code lediglich aus Ada-95-Typen und mit diesen verknüpften Operationen besteht, kommt er in der Abbildung nicht als eigenständige Komponente(n) vor.

### 5.1.2.3 Beispiel für generierten Code

Hier soll anhand einer Regel vorgestellt werden, wie die Umsetzung in Ada-95-Code aussieht. Für eine Regel wird sowohl ein gleichnamiger Ada-95-Typ als auch ein Regeldescriptor angelegt. Der Deskriptor enthält den Namen und eine Beschreibung des Strukturteils einer Regel und ist später über das GCI abrufbar. Die Deskriptoren dienen dem Algorithmus zur Bindung von Regeln (dem Regel-Matching-Algorithmus, vgl. Abschnitt 5.1.4.1) zur Ermittlung potentieller Regelinstanzen (siehe oben).

Der Ada-95-Typ selbst hat folgende Bestandteile:

- Die formalen Komponenten des Strukturteils werden in Attribute passenden Typs übersetzt.
- Der Zeitbedarf der Regel kann über die Methode `delta_c` abgefragt werden.
- Der Bedingungsteil wird in eine Methode `is_valid` übersetzt, die aufgrund der Attributbelegung für die formalen Komponenten einen Booleschen Wert für die Gültigkeit der Bindung zurückliefert.
- Der Aktionsteil wird in eine Methode `execute` übersetzt. Die dabei benötigten lokalen Variablen des Aktionsteils werden als lokale Variablen der Methode realisiert.

Die Prioritäten der Regeln werden nicht übernommen. Stattdessen werden vor der abschließenden Code-Generierung die Regeln anhand ihrer Prioritäten – bei gleicher Priorität anhand des Namens – sortiert und ihre Typdeskriptoren in einer Liste gespeichert. Der Deskriptor der Regel mit der höchsten Priorität steht dabei vorne in der Liste, da diese Regel bei der Instanziierung bevorzugt betrachtet werden soll. Durch die Reihenfolge in der Liste ist also die Prioritätsinformation erhalten geblieben. Diese Liste wird ebenfalls vom GCI verwaltet und ist damit dem Animator zugänglich.

An der Regel `EnhanceSpecification`, die hier noch einmal wiedergegeben wird, soll der entstandene Ada-95-Code demonstriert werden. Die Regel lautet:

```

rule EnhanceSpecification[1000] taking 0 is
structure
  s: Specification;
  d: Developer;
  w: Specifies(d,s);
constraints
  s.function_points < s.max_function_points;
declare
  new_function_points : Real :=
    (s.max_function_points - s.function_points)*0.1
    * d.experience * d.motivation * w.intensity;
begin
  s.function_points :=
    min(s.function_points + new_function_points,
        s.max_function_points);
end rule;

```

Der erzeugte Ada-95-Code wird im folgenden vorgestellt. Der Ada-95-Typ wird als Unterklasse der abstrakten Oberklasse `Rule` deklariert und hat Attribute für die formalen Komponenten des Strukturteils. Der Typ eines Attributs ist ein Zeiger auf Objekte des Typs der formalen Komponente, da die Objekte bei einer Bindung referenziert und nicht kopiert werden sollen.

```

type EnhanceSpecification is new Base2.Rule with
  record
    s: Specification_Ptr;
    d: Developer_Ptr;
    w: Specifies_Ptr;
  end record;

```

Danach folgen die Methoden des Typs. Die Methode `delta_c` für den Zeitverbrauch gibt 0 zurück, da der Zeitverbrauch der Regel 0 Minuten beträgt.

```

function delta_c(X: in EnhanceSpecification) return Natural is
begin
  return 0;
end;

```

Die Methode `is_valid`, die den Attributbedingungsteil modelliert, enthält alle Bedingungen. Sie liefert wahr, wenn alle Bedingungen erfüllt sind. Da es mehrere Bedingungen geben kann, ist es praktisch, die Bedingungen in der Form einer `if`-Abfrage nacheinander abzufragen.

```

function is_valid(X: in EnhanceSpecification) return Boolean is
begin
  if not X.s.function_points < X.s.max_function_points then
    return FALSE;
  end if;
  return TRUE;
end;

```

Die Methode `execute` schließlich realisiert den Aktionsteil der Regel. Die Deklaration der lokalen Variable `new_function_points` wird in eine lokale Variable der Methode übersetzt, die Aktionen kommen in den Rumpf der Methode.

```

procedure execute (X: in out EnhanceSpecification) is
  new_function_points : Real :=
    (X.s.max_function_points - X.s.function_points)*0.1
    * X.d.experience * X.d.motivation * X.w.intensity;
begin
  X.s.function_points :=
    min(X.s.function_points + new_function_points,
        X.s.max_function_points);
end;

```

Der formale Parameter X in den Methoden ist in Ada 95 notwendig. Auf diese Weise wird die Zugehörigkeit von Methoden zu einem Typ ausgedrückt. In den meisten anderen objektorientierten Programmiersprachen, zum Beispiel in C++ (Stroustrup, 1992), werden Methoden lokal zum Typ deklariert und können deshalb direkt auf die Attribute zugreifen.

### 5.1.3 Vergleich der Ansätze

Jeder der beiden Ansätze hat seine eigenen Vor- und Nachteile. Die folgenden beiden Abschnitte arbeiten die Vorteile und Nachteile des Code-Generator-Ansatzes gegenüber dem Interpreter-Ansatz heraus.

#### 5.1.3.1 Vorteile des Code-Generator-Ansatzes

Die Vorteile des Code-Generator-Ansatzes gegenüber dem Interpreter-Ansatz sind:

- Viele Funktionen des Laufzeitsystems müssen nicht vom Ausführungsmechanismus des Modells realisiert werden, sondern werden vom Ada-Laufzeitsystem übernommen. Dazu zählt nicht nur die Ausführung der Aktionsteile, sondern auch die Instanziierung von Komponententypen, Regeln und Benutzerkommandos. Die Implementierung von Funktionen ist mühelos, da eine direkte Abbildung auf Ada-Funktionen vorgenommen werden kann.
- Die Ausführung des Modells in Maschinencode statt durch einen Interpreter steigert die Geschwindigkeit des Animators. Insbesondere der viel Zeit benötigende Vorgang der Suche nach passenden Regelinstanzen in der aktuellen Situation kann dadurch beschleunigt werden.
- Der Aufbau einer internen Modellbeschreibung während der Übersetzung kann entfallen, da der BASE-2-Code direkt in Ada-95-Code umgesetzt werden kann.
- Das fertige Modell kann zusammen mit dem Dolmetscher als ausführbares Programm distribuiert werden. Alternativ kann auch der Ada-95-Code des ausführbaren Modells weitergegeben werden. Der Code-Generator wird zur Ausführung nicht mehr benötigt.

#### 5.1.3.2 Nachteile des Code-Generator-Ansatzes

Die Nachteile des Code-Generator-Ansatzes gegenüber dem Interpreter-Ansatz sind:

- Beim Interpreter-Ansatz wird ein geschlossenes System entwickelt. Beim Code-Generator-Ansatz muß hingegen auf mindestens zwei Ebenen gedacht werden. Zum einen muß der Code-Generator selbst entwickelt werden. Zum anderen will das ausführbare Modell und der dafür generierte Code durchdacht sein. Dies könnte zu Schwierigkeiten insbesondere in der Wartung der Basismaschine führen, da der Ansatz höhere Ansprüche an das Abstraktionsvermögen des wartenden Entwicklers stellt und eine längere Einarbeitung erfordert.

- Beim Interpreter-Ansatz findet nur ein Übersetzungsvorgang statt, beim Code-Generator kommt ein weiterer hinzu: die Übersetzung des generierten Codes durch den Ada-Compiler. Bei der Übersetzung des generierten Codes dürfen keine Fehlermeldungen auftreten, da diese Meldungen dem Benutzer in der Regel völlig unverständlich sein werden. Die Korrektheit des generierten Codes muß daher sehr hoch sein.
- Ein weiterer daraus resultierender Nachteil des Code-Generator-Ansatzes ist die voraussichtlich längere Wartezeit zwischen der Änderung eines BASE-2-Modells und dem Beginn seiner Ausführung.

Ich habe den Code-Generator-Ansatz wegen der oben genannten Vorteile ausgewählt. Insbesondere die Tatsache, daß viel Funktionalität des Laufzeitsystems nicht selbst implementiert werden muß, war verlockend. Ein Grundgedanke der prototypischen Implementierung ist ja, möglichst wenig selbst zu implementieren, sondern möglichst viele bereits vorhandene Bausteine in Form von Modulen oder Software-Werkzeugen zu verwenden.

Welche guten und schlechten Erfahrungen die Implementierung des Code-Generator-Ansatzes mit sich brachte, ist dem Abschnitt über die Implementierung, Abschnitt 5.2, zu entnehmen.

#### 5.1.4 Regelausführungsalgorithmus

In diesem Abschnitt werden der Regelausführungsalgorithmus und ein Teilalgorithmus von ihm, der Regel-Matching-Algorithmus, genauer vorgestellt und auf ihre Terminierung und Komplexität untersucht.

Der grobe Ablauf des Regelausführungsalgorithmus ist bereits durch die Sprachbeschreibung festgelegt. Sein Aussehen kann in Abschnitt 4.5.2 nachgeschlagen werden. Der Regelausführungsmechanismus geht wie folgt vor: Es werden so lange alle Regelinstanzen erzeugt und jeweils diejenige neue Instanz mit der höchsten Priorität ausgeführt, bis keine neuen Instanzen mehr gefunden werden können.

Die Vorgehensweise zur Ermittlung aller möglichen Regelinstanzen *einer* Regel in der aktuellen Situation, das sogenannte *Regel-Matching*, ist in der Sprachbeschreibung nicht festgelegt. Deshalb wird hier der in der Implementierung verwendete Regel-Matching-Algorithmus beschrieben und auf Terminierung sowie auf seine Zeit- und Platzkomplexität untersucht.

Es wird im folgenden davon ausgegangen, daß die Regeln des Regelmodells sortiert in einer Liste vorliegen. Die Regel mit der höchsten Priorität steht dabei am Anfang der Liste. Um die Regelinstanz mit der höchsten Priorität in der aktuellen Situation zu ermitteln, werden die Regeln in der Listenreihenfolge sukzessive abgearbeitet

Für jede Regel wird der Regel-Matching-Algorithmus angewandt, bis eine Regelinstanz gefunden werden kann. Es wird immer die Regelinstanz mit der höchsten Priorität gefunden und ausgeführt, da die Regeln in der Reihenfolge ihrer Prioritäten betrachtet werden.

##### 5.1.4.1 Regel-Matching-Algorithmus

Der gewählte Algorithmus läßt sich treffend als naiver ad-hoc-Algorithmus charakterisieren. Aus der Sicht der Zeit- und Platzkomplexität dürfte er als relativ ineffizient gelten.

Um eine Bindung für den Strukturteil einer Regel zu finden, wird zuerst der Strukturbedingungsteil der Regel in die Menge der formalen Entitäten und die Menge der formalen Relationen aufgeteilt.

Die formalen Entitäten werden durchnummeriert und nacheinander abgearbeitet. In der aktuellen Situation werden alle Entitäten daraufhin untersucht, ob sie auf die gerade betrachtete formale Entität passen. Kann eine passende Entität gefunden werden, wird sie als bereits verwendet markiert<sup>1</sup> und die Suche wird rekursiv für die nächste formale Entität fortgesetzt.

Falls auch die letzte formale Entität in der Aufzählung gebunden werden kann, hat man eine *Bindung der formalen Entitäten* gefunden. Nun muß geprüft werden, ob die formalen Relationen ebenfalls gebunden werden können, d.h. ob die geforderten Beziehungen zwischen den gebundenen Entitäten in der aktuellen Situation bestehen. Ist dies der Fall, hat man eine *Bindung der formalen Struktur* gefunden.

Die Bindung wird auf ihre Gültigkeit geprüft, indem der Attributbedingungsteil ausgewertet wird. Ist die Bindung gültig, wird sie gespeichert und das Suchverfahren wird fortgesetzt. Am Ende hat man alle gültigen Bindungen der Regel für die aktuelle Situation ermittelt.

Die gefundenen gültigen Bindungen werden gegen die bereits ausgeführten Regelinstanzen abgeglichen. Bereits ausgeführte Regelinstanzen werden entfernt. Bleibt mindestens eine Instanz übrig, wird diejenige ausgeführt, die als erste gefunden wurde.

**Terminierung.** Der Algorithmus zur Bestimmung aller gültigen Bindungen terminiert, da in jedem Schritt die noch zu bearbeitende Aufgabe kleiner wird. Da sowohl die Menge der formalen Entitäten als auch die Menge der Entitäten in der aktuellen Situation endliche Mengen sind, ist dies auch für die Menge der möglichen Bindungen der Fall. Deshalb wurden nach endlich vielen Schritten alle Möglichkeiten betrachtet, der Algorithmus terminiert.

**Zeitkomplexität.** Sei  $e$  die Anzahl der formalen Entitäten der Regel und  $n$  die Anzahl der Entitäten der aktuellen Situation. Dann benötigt man zur Ermittlung aller gültigen Bindungen

$$(n)_e = \frac{n!}{e!} = (n \cdot (n-1) \cdot \dots \cdot (n-e+1))$$

Schritte. Für die  $i$ -te der  $e$  formalen Entitäten werden nämlich die  $(n-i+1)$  verbleibenden Entitäten der aktuellen Situation betrachtet ( $1 \leq i \leq e$ ). Da das Verfahren rekursiv für jede formale Entität ausgeführt wird, multiplizieren sich die Schritte. Die anschließende Prüfung der formalen Relationen und gegebenenfalls der Gültigkeit kann dagegen als konstanter Faktor betrachtet werden und wird daher nicht genauer ausgeführt. Man kann damit die Zeitkomplexität mit  $O((n)_e)$  abschätzen.

**Platzkomplexität.** Die Platzkomplexität des Regel-Matching-Algorithmus beträgt ebenfalls  $O((n)_e)$ . Im schlimmsten Fall gibt es  $(n)_e$  gültige Bindungen, die gespeichert werden müssen. Bei diesem Fall ist jede denkbare Bindung auch gültig. Im Normalfall wird die Zahl der gültigen Bindungen jedoch viel niedriger liegen.

---

1. Die Markierung ist notwendig, da die Bindung isomorph zur Strukturbedingung sein soll. Eine Entität der aktuellen Situation darf nicht mehrfach gebunden werden, ansonsten ist die Isomorphieforderung verletzt.

### 5.1.4.2 Optimierungen

Die Zeitkomplexität läßt sich durch geeignetere Datenstrukturen für die aktuelle Situation verbessern. Die aktuelle Situation könnte zum Beispiel zu jedem Entitätstyp immer alle passenden Instanzen verwalten. Damit würde die Suche nach Bindungen für formale Entitäten beschleunigt werden. Für jede formale Entität müßten dann nicht alle Entitäten der aktuellen Situation betrachtet werden, sondern nur die Entitäten mit dem passenden Typ.

Die Platzkomplexität läßt sich einfach optimieren, indem bereits nach der Ermittlung einer gültigen Bindung überprüft wird, ob die zugehörige Regelinstanz schon ausgeführt wurde. Wenn dies nicht der Fall ist, hat man die nächste auszuführende Regelinstanz gefunden und kann den Suchvorgang abbrechen. Der Platzbedarf des Regel-Matching-Algorithmus ist dann konstant. Diese Änderung wirkt sich auch positiv auf die Zeitkomplexität aus, da nicht immer alle möglichen Bindungen untersucht werden müssen.

Darüber hinaus sind natürlich weiter spezialisierte Algorithmen möglich. Hier kann man sich auch die Erfahrungen aus dem Bereich der Graphgrammatik-Interpreter zunutze machen. Ein Graphgrammatik-Interpreter muß zur Ausführung von graphgrammatikbasierten Programmen ein ganz ähnliches Problem der Bindung formaler Strukturen lösen: die Bindung der linken Seiten von Graphgrammatikproduktionen in einem Graphen.

### 5.1.4.3 Komplexität des Regelausführungsalgorithmus

Auf der Grundlage der Komplexität des Regel-Matching-Algorithmus kann nun die Komplexität des Regelausführungsalgorithmus abgeschätzt werden. Dieser ermittelt ja so lange die nächste auszuführende Regelinstanz, bis keine neue mehr gefunden werden kann.

Sei  $n$  hier die *maximale* Anzahl der Entitäten in der Situation während eines Simulationsschritts. Diese Zahl ist prinzipiell nicht vorhersagbar, da ein Modell während seiner Ausführung beliebig viele Entitäten erzeugen kann. In der Regel kann  $n$  jedoch für ein festes Modell nach oben abgeschätzt werden. Sei  $r$  die Anzahl der Regeln und  $e$  die *maximale* Zahl von formalen Entitäten in den Strukturteilen der Regeln. Dann gibt es maximal  $r \cdot (n)_e$  Regelinstanzen. Dies ist der Fall, wenn jede Entität an jede formale Entität jeder Regel gebunden werden kann und alle diese Bindungen gültig sind.

Da in jedem Durchlauf genau eine Regelinstanz ausgeführt wird, müssen also  $r \cdot (n)_e$  Durchläufe absolviert werden. Der maximale Zeitbedarf zur Ermittlung der nächsten auszuführenden Regelinstanz beträgt  $(r \cdot (n)_e)$ , da im schlimmsten Fall alle möglichen Regelinstanzen untersucht werden müssen. Insgesamt ergibt sich eine Zeitkomplexität von  $O(r^2 \cdot (n)_e^2)$ . Man sieht hier, daß durch Einsatz eines optimierten Regel-Matching-Algorithmus in Zusammenhang mit geeigneten Datenstrukturen zur Verwaltung der aktuellen Situation viel Zeit eingespart werden kann. Außerdem könnte Information aus den vorhergehenden Schritten zur Ermittlung der nächsten auszuführenden Regelinstanz verwendet werden.

Die Platzkomplexität entspricht der Anzahl der Regelinstanzen, kann also durch  $O(r \cdot (n)_e)$  abgeschätzt werden. Dies kommt daher, daß alle ausgeführten Regelinstanzen (in  $R_{akt}$ ) gespeichert werden müssen, um eine mehrfache Ausführung einer Instanz zu vermeiden.

## 5.2 Implementierung

In diesem Abschnitt wird auf die Implementierung der Basismaschine im Rahmen dieser Arbeit eingegangen. Abschnitt 5.2.1 stellt die Rahmenbedingungen der Implementierung vor. Abschnitt 5.2.2 umreißt den Umfang der Implementierung. Abschnitt 5.2.3 befaßt sich mit einigen Problemen, die in der Implementierung auftraten. Eine Bewertung der Implementierung folgt abschließend in Abschnitt 5.2.4.

### 5.2.1 Rahmenbedingungen

Die Implementierung erfolgte in der Programmiersprache Ada 95 auf den Rechnern der Abteilung Software Engineering. Dies sind eine DEC-Station 3100 mit dem Betriebssystem Ultrix 4.3 und eine Sun Sparc Station mit dem Betriebssystem Solaris. Da in Ada 95 portabel programmiert werden kann, sollte die Basismaschine ohne Änderungen am Code auf beiden Rechnern übersetzbar und ausführbar sein.

Als Entwicklungsumgebung wurden folgende Werkzeuge verwendet:

- gnat, der Ada 95 Compiler der GNU Software Foundation<sup>1</sup>, in der Version 3.01,
- gdb, der Debugger der GNU Software Foundation mit rudimentärer Ada-95-Unterstützung, in der Version 4.15.1.gnat.1.10,
- aflex und ayacc, ein Scanner- und ein Parser-Generator für Ada, in der Version 1.4a,
- emacs, ein umfangreicher Editor der GNU Software Foundation, der einen speziellen Ada-95-Modus zur Erfassung und Bearbeitung von Ada-95-Programmen besitzt, in der Version 19.30.1, und
- das Versionsverwaltungssystem RCS (Resource Control System), das in den emacs integriert worden ist.

Da es sich bei den Produkten der GNU Software Foundation und bei aflex und ayacc um für Privatleute und Bildungseinrichtung kostenlos nutzbare Software handelt, sind hohe Ansprüche an die Zuverlässigkeit und Korrektheit der Programme nicht immer gerechtfertigt. Die Programme befinden sich jedoch in einem kontinuierlichen Verbesserungsprozeß, bei dem durch die Anwender entdeckte und gemeldete Mängel und Fehler in der Regel schnell behoben werden. Insbesondere gnat befindet sich noch stark in der Entwicklung, wobei in der Version 3.01 bereits der größte Teil der Funktionalität realisiert ist, die ein Ada-95-Compiler laut dem Ada 95 Reference Manual (Intermetrics, 1995) besitzen muß.

### 5.2.2 Umfang

Da für die Implementierung nur die begrenzte Zeit von fünf Wochen zur Verfügung stand (vgl. hierzu Abschnitt 6.1.1.1), mußte der Umfang der Implementierung des Entwurfs eingeschränkt werden. Es sollte ein prototypisches Minimalsystem implementiert werden, das die Realisierbarkeit und die Brauchbarkeit der Basissprache demonstrieren kann.

---

1. GNU ist eine Abkürzung für „GNU is Not Unix“. Die GNU Software Foundation hat sich zum Ziel gesetzt, für Unix eine umfangreiche Werkzeugpalette zu entwickeln und diese kostenlos zur Verfügung zu stellen. Unterstützt wird dies durch die kostenlose Mitarbeit hunderter Entwickler weltweit.

Bereits vor Beginn der Implementierung wurde entschieden, die Listentypen nicht zu realisieren. Außerdem sollte auf die benutzerdefinierten Funktionen verzichtet werden, deren Realisierung sich schließlich als so einfach erwies, daß sie doch implementiert wurden. Schließlich sollten auch die meisten vordefinierten Funktionen wie zum Beispiel die in Abschnitt 4.3.1.3 erwähnten Funktionen `exists`, `conforms` und `is_a` nicht implementiert werden.

Aus Schwierigkeiten während der Implementierung ergaben sich weitere Einschränkungen. Diese werden im folgenden zusammen mit ihrer Ursache vorgestellt.

### 5.2.3 Probleme

Die Schwierigkeiten, die im Laufe der Implementierung auftraten, lassen sich grob in zwei Kategorien einteilen. Zum Teil handelt es sich um Fehler im Entwurf, zum anderen um Probleme mit den Werkzeugen der Entwicklungsumgebung, die nicht immer wie erwartet arbeiteten.

#### 5.2.3.1 Fehler im Entwurf

Der Entwurf war an manchen Stellen nicht bis zum Ende durchdacht. Das zeigte sich insbesondere im Bereich der Code-Generierung. Ein besonders auffälliges Beispiel soll hier herausgegriffen werden, da es zu einer großen Einschränkung der Implementierung führte.

Aufgrund der strengen Typprüfung von Ada 95 stellte sich die Realisierung des conforming-Konzepts mit dem verfolgten Ansatz der Erzeugung von Ada-95-Typen aus den Definitionen als unmöglich heraus. Ursache hierfür ist die Anwendbarkeit der Typverträglichkeiten im Strukturbedingungsteil einer Regel oder eines Benutzerkommandos. Die formalen Komponenten des Strukturbedingungsteils einer Regel werden in Attribute des für die Regel generierten Ada-Typs umgesetzt (vgl. hierzu Abschnitt 5.1.2.3). Durch die statische Typprüfung von Ada 95 ist es unmöglich, einem solchen Attribut ein Objekt anderen Typs zuzuweisen.

Angenommen, eine Regel hätte in ihrem Strukturteil eine formale Entität `x` vom Typ `Person`. Der generierte Typ hat dann ein Attribut `x` vom generierten Ada-95-Typ `Person`, das bei der Instanziierung dann die gebundene Entität aus der aktuellen Situation aufnehmen soll. Nach der Semantik von BASE-2 ist es erlaubt, an eine formale Entität vom Typ `Person` eine Entität vom Typ `Developer` zu binden, da `Developer` zu `Person` typverträglich ist. Das Ada-95-Objekt vom Ada-Typ `Developer` kann jedoch nicht dem Attribut `x` des Ada-Typs für die Regel zugewiesen werden, da es sich nicht um ein Objekt vom Ada-Typ `Person` handelt.

Dieses Problem wäre einfach lösbar, wenn es in Ada 95 das Konzept der Typverträglichkeitsdeklaration gäbe. Das ist aber nicht der Fall. Stattdessen gibt es dort einfache Vererbung, aus der sich Typverträglichkeitsbeziehungen ergeben.

Es wäre denkbar, die Typverträglichkeiten in BASE-2 in Vererbungsbeziehungen in Ada 95 umzusetzen. Dies würde allerdings eine umfangreiche Analyse der Typverträglichkeitsbeziehungen erforderlich machen. Außerdem lassen sich die Typverträglichkeitsbeziehungen, die sich aus einer Mehrfachvererbung ergeben, nicht durch einfache Vererbung modellieren.

Alternativ wäre es auch denkbar gewesen, das Typverträglichkeitskonzept von BASE-2 durch ein Konzept der einfachen Vererbung zu ersetzen. Dann wären die Vererbungen direkt in Ada 95 übersetzbar gewesen. Diese Änderung der Basissprache hätte jedoch bedeutet, einen Teil der Ausdrucksfähigkeit des Typverträglichkeitskonzepts, insbesondere die Mehrfachvererbung, pragmatischen Überlegungen zu opfern.

Deshalb wurde beschlossen, in dieser Implementierung die Bindung formaler Komponenten im Strukturbedingungsteil einer Regel oder eines Benutzerkommandos auf Objekte des geforderten Typs zu beschränken. Typverträglichkeiten spielen also bei der Bindung des Strukturbedingungsteils keine Rolle mehr. Dies stellt eine gravierende Einschränkung der Basissprache dar, da damit die Anwendung des conforming-Konzepts praktisch wegfällt.

### 5.2.3.2 Werkzeuge der Entwicklungsumgebung

In diesem Abschnitt sollen die Erfahrungen mit den oben genannten Werkzeugen wiedergegeben werden. Außerdem werden Einschränkungen der Implementierung genannt, die sich aus den Eigenheiten der Werkzeuge ergaben.

**gnat.** Die meisten Probleme ergaben sich in Zusammenhang mit dem Ada-95-Compiler, was auch daran liegen dürfte, daß er erst kürzlich den Stand der vollen Ada-95-Unterstützung erreicht hat.

Für die Code-Fragmente des Parsers wurde ein abstrakter Datentyp implementiert, der auf dem vordefinierten Paket `Ada.Strings.Unbounded` (Intermetrics, 1995, A.4.5) für Strings beliebiger Länge aufbaut. Bei der Compilierung und auch zur Laufzeit traten seltsame Fehler auf, so daß schließlich das Paket `Ada.Strings.Bounded` (Intermetrics, 1995, A.4.4) mit einer maximalen String-Länge von 10.000 Zeichen verwendet werden mußte. Dies bedeutet, je nach Implementierung dieses Pakets<sup>1</sup>, eine gewaltige Speicherverschwendung, da zum Beispiel für einen String mit zehn Zeichen 10.000 Byte verbraucht werden können. Der gewaltige Speicherbedarf des Code-Generators deutet darauf hin, daß dies auch der Fall ist.

Ein weiterer Compiler-Fehler trat auf, als versucht wurde, Ada-95-Typen zu übersetzen, die aus Entitätstypen mit Real-Attributen erzeugt worden waren. Dieser Fehler war ein echter Compiler-Fehler. Deshalb mußte auf den Basistyp `Real` als Attributtyp völlig verzichtet werden. Die Folgeversion `gnat 3.03` wird – der Aussage eines beteiligten Entwicklers zufolge – diesen Fehler beheben.

Ein Linker-Fehler schließlich machte die Absicht zunichte, die Basismaschine auf beiden Rechnern der Abteilung Software Engineering lauffähig zu machen. Ein während der Entwicklung plötzlich auftretender und nicht behebbare Fehler beim Link-Vorgang auf der DEC Station machte es notwendig, sich auf die Sparc Station zu beschränken, wo ein anderer Linker installiert ist.

---

1. Der Implementierungsratschlag (*Implementation Advice*) im Ada 95 Reference Manual zu dem Paket `Ada.Strings.Bounded` legt nahe, es nicht mit dynamischer Allokation zu implementieren. Daher ist es durchaus möglich, daß für jedes Objekt vom Typ `Bounded_String` Speicherplatz für einen String maximaler Länge allokiert wird.

**gdb.** Der GNU Debugger wurde eigentlich für gcc, den C-Compiler des GNU-Projekts, entwickelt. Ein Ada-Programm spiegelt deshalb vor, ein C-Programm zu sein. Auf diese Weise kann es mit dem gdb untersucht werden. Die Angaben des Debuggers über die Namen aufgerufener Funktionen stellten sich des öfteren als falsch heraus, während der Verweis auf die Zeilennummer in der Quelldatei immer korrekt war. Dies deutet auf eine fehlerhafte Symboltabellenerzeugung durch den Linker hin.

**aflex und ayacc.** Mit diesen beiden Werkzeugen gab es, abgesehen von der etwas kryptischen Scanner- und Parserbeschreibungssprache, keine Probleme. Die Parserbeschreibungsdatei und der daraus generierte Ada-Code waren allerdings zum Schluß aufgrund ihrer Größe kaum mehr handhabbar.

**emacs.** Der Editor an sich arbeitet sehr zuverlässig. Der Ada-95-Modus hingegen weist noch einige Fehler auf, besonders was die automatische Einrückung von Zeilen angeht. Hier ist inzwischen eine neue Version veröffentlicht worden, die diese Fehler beheben soll. Der Diskussion im Internet ist jedoch zu entnehmen, daß immer noch Fehler enthalten sind.

**RCS.** Das RCS arbeitete immer einwandfrei. Der Umgang mit dem RCS wurde durch die Integration in den emacs stark vereinfacht.

#### 5.2.4 Bewertung

Die Implementierung der Basismaschine mit den genannten Einschränkungen war erfolgreich. Erfolgreich heißt hier jedoch nur, daß das Programm vorführbar ist. Ein Demonstrationsmodell, aus dem das Beispielmmodell im Anhang abgeleitet wurde, ließ sich problemlos übersetzen und ausführen.

Aufgrund der prototypischen Implementierung wurden kaum systematische Modul- und Systemtests durchgeführt, die unter Umständen noch weitere Mängel der Implementierung aufgezeigt hätten. Es kann also davon ausgegangen werden, daß die Implementierung noch bisher unentdeckte Mängel aufweist. Alle bekannten Mängel wurden dagegen behoben.

Das Demonstrationsmodell verfügt über nur zwei Regeln. Außerdem ist die Startsituation mit etwa zehn Instanzen verhältnismäßig klein. Dadurch ist die Übersetzungs- und die Ausführungsgeschwindigkeit des Modells noch annehmbar. Experimente mit einem Modell mit etwa 50 Regeln haben gezeigt, daß die Implementierung bereits dort an ihre Grenzen stößt. Der Code-Generator benötigt zur Übersetzung dieses Modells etwa 40 MB (!) Speicher. Im Laufe der Übersetzung bekommt das Betriebssystem immer größere Schwierigkeiten, den wachsenden Speicherbedarf zu befriedigen, wodurch der Code-Generator immer stärker verlangsamt wird. Bei etwa 60 Regeln ist die endgültige Grenze erreicht: Dem Code-Generator geht während der Code-Erzeugung der Speicher aus.

Führt man das Modell mit den 50 Regeln nach der Übersetzung aus, macht sich auch die Ineffizienz des Regelausführungsalgorithmus deutlich bemerkbar. Die Regeln wurden so gewählt, daß sie alle nacheinander einmal feuern können. Das Modell wird sehr langsam, bis es schließlich nach einigen Simulationsschritten aus Speichermangel abgebrochen wird. Der Speichermangel dürfte vor allem durch Speicherlecks in der Implementierung verursacht werden.

## 5.3 Beispiellauf

In diesem Abschnitt soll die Übersetzung und der Ablauf des Beispielmodells (siehe Anhang B) aus Benutzersicht gezeigt werden. Zu diesem Zweck wird das Beispielmodell vom Code-Generator übersetzt und anschließend ausgeführt. Während der Ausführung werden ein paar Benutzerkommandos eingegeben.

### 5.3.1 Übersetzung

Der Aufruf des Code-Generators lautet

```
cg -d -n beispiel.b2
```

Als Argumente werden die Debug-Option `-d`, die Nicht-Ausführungsoption `-n` und der Name der Modellbeschreibungsdatei `beispiel.b2` dem Code-Generator übergeben.

Der Code-Generator meldet sich daraufhin mit der Ausgabe

```
Base-2 Code Generator Version 1.0 (c) 1996 by Ralf Reissing
```

Daran schließen sich eine Menge Debug-Meldungen an, die im folgenden auszugsweise angegeben sind.

```
DEBUG: BASE-2 Code Generator starting ...
DEBUG: Debug is on
DEBUG: No execution is on
DEBUG: Generating code for entity type project ...
DEBUG: Generating code for entity type project ...

[...]

DEBUG: Generating code for command letspecify ...
DEBUG: Generating code for command letspecify ...
DEBUG: Input file successfully read
DEBUG: Writing code ...
DEBUG: Compiling model ...
DEBUG: BASE-2 Code Generator terminated.
```

Die Debug-Ausgaben liefern Informationen, welche Bearbeitungsstufe gerade durchlaufen wird.

Der Code-Generator erzeugt die notwendigen Ada-95-Dateien und stößt daraufhin den Übersetzungsprozeß an. Wäre die Nichtausführungsoption `-n` nicht angegeben worden, hätte er anschließend das ausführbare Modell gestartet.

Die Code-Erzeugung einschließlich der Übersetzung hat 52 Sekunden gedauert, wobei davon 15 Sekunden echte Rechenzeit waren (gemessen mit dem Unix-Werkzeug `time` auf dem Rechner `madrid`).

### 5.3.2 Ausführung

Der Name des erzeugten Programms entspricht dem Modellnamen. Das Modell wird nun über die Kommandozeile mit dem Aufruf

```
beispiel -d
```

gestartet. Auch das ausführbare Modell versteht die Debug-Option `-d` und reagiert darauf mit ausführlichen Meldungen über das, was gerade vor sich geht. Als Beispiel ein kleiner Auszug aus den Meldungen zu Beginn der Ausführung des Modells.

```
DEBUG: BASE-2 model starting ...
DEBUG: Debug is on
DEBUG: Setting time step to 60
DEBUG: Setting model time to 1995/11/20/08:00
DEBUG: Starting Interpreter with dictionary demo
DEBUG: Sent to si: message Scenario
DEBUG: Creating entity, type Project aka MOHN-2000
DEBUG: Creating entity, type Manager aka Peter Leiter
DEBUG: Creating relation, type Manages
[...]
```

Zuerst wird die Modellzeit initialisiert. Dann wird der Dolmetscher gestartet. Dieser öffnet ein Fenster, in das erst einmal eine Willkommensmeldung ausgegeben wird. Die Basismaschine sendet dem Dolmetscher (si, kurz für SESAM Interpreter) die Nachricht `Scenario`, woraufhin eine Beschreibung des Spielszenarios ausgegeben wird. Anschließend werden die Komponenten der Startsituation erzeugt, wovon hier nur drei als Beispiel angegeben sind.

Die ausgegebene Beschreibung des Spielszenarios lautet:

```
Welcome to SESAM.

SCENARIO
=====

You are working for Softex, a medium-sized software house.
As you did a very good job as project leader assistant during
the last software project, you are now appointed project leader.

Dr. Smith, Vice President, explains your mission:
The SUEBIA insurance company needs a check-accountance system as
soon as possible. Softex and Suebia have agreed on a fix price of
400.000,- DM. Your project budget is 250.000,- DM. Hardware is a
HAL 9000 machine.

The contract specifies the following details:

- The program has to collect check data from remote Suebia
  locations, compile open and payed checks, print weekly and
  monthly reports, and allow for data exchange over the
  Internet. Online checking of total check traffic must also be
  possible.

- The system must be operational by June 30, 1995.

Dr. Smith passes four closed envelopes to you, containing
information about Andrea Sevensleep, John Bankmiller, Katharina
Koolhaas and Martin Huthmacher. They have been associated with
Softex for several years. These four people are available
immediately, and you can hire them for your project. You may
also hire up to five additional developers.

The current date is December 1, 1994.
```

Nun ist der Spieler an der Reihe. Das Beispielmmodell kennt nur drei Benutzerkommandos, nämlich `Interview`, `Hire` und `LetSpecify`. Diese drei Kommandos sollen nun demonstriert werden. In der Anfangssituation des Beispielmmodells hat das Projekt ein Mitglied, John Bankmiller. Dieser hat zur Zeit keine Aufgaben.

Zunächst will der Spieler eine weitere Entwicklerin einstellen. Deshalb führt er ein Vorstellungsgespräch mit Andrea Sevensleep, einer der Entwickler, die Dr. Smith in der einführenden Meldung als verfügbar genannt hat (Benutzereingaben sind fett hervorgehoben).

```
interview andrea
```

```
ok
```

```
The interview with Andrea Sevensleep confirms your first impression.
```

```
She is a little bit nervous, but she explains she wants to work in your team.
```

Die Antwort des Dolmetschers ist ein `ok`, mit dem er signalisiert, daß er diese Eingabe erkannt und der Basismaschine ein Kommando geschickt hat. Die Basismaschine akzeptiert das Kommando und führt das Benutzerkommando `Interview` aus. Dieses sendet in seinem Aktionsenteil mit `send_message` eine Nachricht an den Benutzer. Diese Nachricht setzt der Dolmetscher in den Text um, der der Benutzereingabe folgt.

Als nächstes wird Andrea Sevensleep eingestellt. Dies ist nur nach einem vorhergehenden Vorstellungsgespräch möglich.

```
hire andrea
```

```
ok
```

```
Andrea Sevensleep starts working. She comes into your office room. You give her a lot of information about your firm and her tasks. And now, you can use one more developer.
```

```
Andrea Sevensleep is glad to start working and therefore she is motivated to do her best.
```

Auch hier reagiert das Modell mit einer Ausgabe im Aktionsteil des Benutzerkommandos `Hire`. Eine solche Ausgabe ist bei jedem Benutzerkommando sinnvoll, um anzuzeigen, daß auch etwas passiert ist.

Nun will der Spieler Andrea spezifizieren lassen.

```
let andrea specify
```

```
ok
```

```
Andrea Sevensleep starts to design the specification.
```

```
"This is rather difficult. Such a specification is very complex."
```

```
Andrea Sevensleep says: "I really like to specify. You can talk to many people. But sometimes they themselves don't know what they want..."
```

Das Projektmitglied John Bankmiller ist bisher untätig. Das soll sich ändern:

```
let john specify
```

```
ok
```

```
John Bankmiller starts to design the specification.  
"This is rather difficult. Such a specification is very complex."  
  
John Bankmiller says: "I really like to specify. You can talk to  
many people. But sometimes they themselves don't know what they  
want..."
```

Die letzten beiden Ausgaben des Modells sind von demselben Benutzerkommando erzeugt worden und unterscheiden sich doch. Der externe Name des Entwicklers ist jeweils in die Ausgabe eingebaut worden. Diese Anpassung der Nachricht des Modells nimmt der Dolmetscher vor.

Katharina Kohlhaas ist auch eine verfügbare Entwicklerin. Der Versuch, sie spezifizieren zu lassen, scheitert jedoch. Sie ist kein Projektmitglied. Dies ist jedoch eine Bedingung im Strukturteil des Benutzerkommandos `LetSpecify`. Deshalb kann das Benutzerkommando nicht instanziiert werden. Es gibt also auch keine Ausgaben des Modells, obwohl der Dolmetscher die Eingabe mit `ok` akzeptiert hat. Die semantische Korrektheit von Eingaben kann der Dolmetscher nicht überprüfen, da ihm die dazu notwendige Information fehlt.

```
let kathy specify  
ok
```

Nun wird das Spiel beendet. Dazu dient das `quit`-Kommando des Dolmetschers.

```
quit  
Thank you for playing SESAM.
```

Der Dolmetscher verabschiedet sich und schließt sein Ein-/Ausgabefenster. Die letzten Debug-Ausgaben beweisen, daß sich Dolmetscher und Basismaschine über das Ende der Modellanimation absprechen:

```
DEBUG: Received from si: quit  
DEBUG: Sent to si: quit  
DEBUG: BASE-2 model terminated.
```

Das ausführbare Modell terminiert gemeinsam mit dem Dolmetscher.

## 6 Zusammenfassung und Ausblick

*Perfection is achieved only on the point of collapse.*

*(C. N. Parkinson)*

In diesem Kapitel soll zunächst der Projektverlauf der Arbeit zusammengefaßt und bewertet werden. Anschließend werden in Form eines Ausblicks Verbesserungen und Erweiterungen der Basismaschine angedacht.

### 6.1 Zusammenfassung

#### 6.1.1 Projektverlauf

Das Projekt der Konzeption und Realisierung der Basismaschine für SESAM-2 wurde nach einem Phasenplan durchgeführt. Dazu wurde das Projekt in fünf Phasen eingeteilt:

- Projektplanung,
- Konzeption und Spezifikation,
- Entwurf,
- Implementierung und Test sowie
- Bericht.

Die Literatursuche und die Materialsammlung für den Bericht waren phasenübergreifende Aktivitäten.

Die fünf Phasen werden im folgenden Abschnitt näher erläutert. Daran schließt sich ein Abschnitt an, der den tatsächlich benötigten Aufwand mit dem geplanten Aufwand vergleicht und die erzielten Ergebnisse zusammenstellt. Zum Schluß wird noch auf den entstandenen Code der Basismaschine eingegangen.

##### 6.1.1.1 Projektphasen

**Planung.** In der Projektplanungsphase wurden die Rahmenbedingungen des Projekts in einem Projektplan niedergelegt. Dazu gehörte auch eine Planung des Vorgehens durch die Aufstellung eines Phasenplans. Zuerst wurden die Phasen festgelegt. Anschließend wurden für jede Phase die durchzuführenden Aktivitäten bestimmt und der Zeitbedarf abgeschätzt. Dann wurden die Phasen zeitlich eingeplant. Abbildung 6.1 zeigt den entstandenen Zeitplan.

Über der Zeitachse, auf der die einzelnen Phasenstart- und endtermine markiert sind, sind die Sollzeiträume in Form von grau unterlegten Balken aufgetragen. Ein Quadrat entspricht dabei einer Woche. Die Istzeiträume sind in Form weiß unterlegter Balken unter den Sollwerten angegeben. Der Projektplan wurde nach der ersten Erstellung nicht mehr verändert, da keine Anpassung an den tatsächlichen Projektverlauf vorgenommen werden mußte. Soll- und Istwerte stimmen völlig überein. Obwohl damit die geplanten Termine eingehalten wurden, entsprechen die Aufwandsverhältnisse der einzelnen Phasen nicht ganz dem Plan. Dies wird in Abschnitt 6.1.1.2 noch behandelt werden.

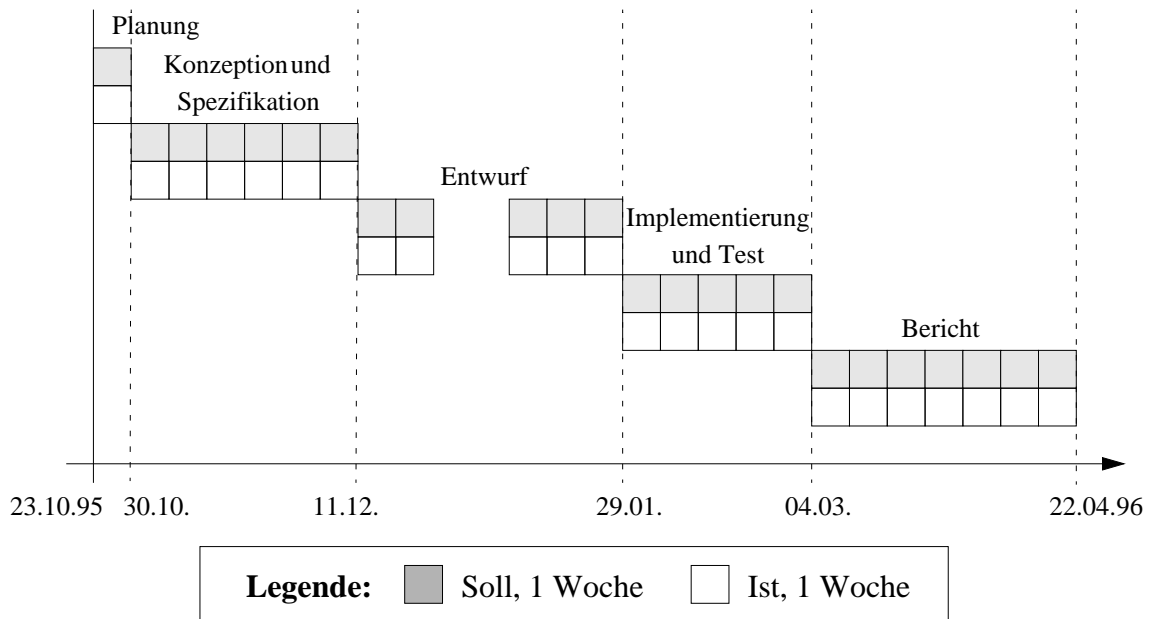


Abbildung 6.1: Zeitplan

**Konzeption und Spezifikation.** In dieser Phase sind die Konzepte der Basismaschine und der Basissprache entwickelt und in einer Spezifikation niedergelegt worden. Dazu bedurfte es zuerst einer Anforderungsanalyse, die sich aus einer Soll- und einer Istanalyse zusammensetzte. In der Istanalyse wurden bisherige Konzepte aus SESAM-1 und SESAM-Lite zusammengetragen und bewertet. Die Sollanalyse umfaßte die geplanten Konzepte einer neuen Hochsprache, die bewährte Konzepte früherer SESAM-Systeme bewahren und nachteilige Konzepte eliminieren sollte. Aus der Anforderungsanalyse konnten dann die Konzepte der Basissprache BASE-2 abgeleitet werden, indem eine Minimalsprache entworfen wurde, auf die alle Hochsprachenkonzepte abgebildet werden konnten. Diese Konzepte wurden in eine formale Beschreibung von BASE-2 umgesetzt, die Bestandteil der Spezifikation der Basismaschine wurde. Außerdem wurden Testfälle für den Systemtest festgelegt.

Nach dem vorläufigen Abschluß der Sprachbeschreibung wurde BASE-2 im Rahmen eines Mitarbeiterkolloquiums den Mitarbeitern der Abteilung Software Engineering vorgestellt und diskutiert. Einige Änderungsvorschläge der Zuhörer flossen anschließend in den Sprachentwurf ein.

**Entwurf.** In der Entwurfsphase wurden mögliche Architekturen für die Basismaschine entwickelt (vgl. Abschnitt 5.1). Anschließend wurde eine Architektur ausgewählt und in einen Modulentwurf umgesetzt. Die Spezifikationen der Module des Modulentwurfs wurden als Paketspezifikationen in Ada 95 notiert.

In die Entwurfsphase fiel der Zwischenbericht zu dieser Arbeit. Außerdem wurde der Entwurf durch einen zweiwöchigen Urlaub über die Weihnachtsferien unterbrochen.

**Implementierung und Test.** Die Implementierungsphase umfaßte die teilweise Implementierung des Entwurfs in einem Prototypen, wobei einige Änderungen am Entwurf als notwendig erkannt und durchgeführt wurden.

In dieser Phase war der Zeitdruck am höchsten, wie den unten aufgeführten Statistiken entnommen werden kann. Die angewandte Vorgehensweise läßt sich am besten als „quick and dirty“ charakterisieren. Für eine prototypische Implementierung ist das jedoch nicht weiter schlimm, da es dort primär um die Untersuchung der Realisierbarkeit von Konzepten geht und nicht um die Produktion qualitativ hochwertige Software.

**Bericht.** In der letzten Phase wurde dieser Bericht verfaßt. Er trägt die Ergebnisse der vorhergehenden Phasen zusammen. Am Ende der Berichtsphase wurde der Abschlußvortrag vorbereitet.

### 6.1.1.2 Aufwand und Ergebnisse

Während der einzelnen Phasen wurde der Aufwand (in Stunden) erfaßt, um so ein realistischeres Bild der Qualität des Projektplans zu erhalten, als das durch einen reinen Terminvergleich möglich wäre. Als Aufwand zählt die reine Arbeitszeit. Pausen und Zeiten für An- und Abfahrt zur Arbeitsstelle wurden nicht aufgenommen. Der tatsächliche Aufwand liegt schätzungsweise bei 133% des ausgewiesenen Aufwands.

In Tabelle 1 wird der erfaßte Aufwand jeder Phase zusammen mit den erzielten Ergebnissen aufgeführt. Der Umfang von Dokumenten wird in DIN-A4-Seiten angegeben, der Umfang von Code in Simple Lines of Code (SLOC). Bei der Implementierungsphase wird zuerst der Code für die Basismaschine und dann der Code für Testfälle angegeben. Auf den Code der Implementierung und das Maß SLOC wird in Abschnitt 6.1.1.3 noch genauer eingegangen.

Phase	Aufwand (h)	Ergebnisse
Planung	10	• Projektplan, 5 Seiten
Spezifikation	153	• Sprachbeschreibung, 77 Seiten • Spezifikation, 16 Seiten • Spezifikation der Testfälle, 5 Seiten • Kolloquiumsvortrag, 14 Folien
Entwurf	85	• Entwurfsdokumentation, 18 Seiten • Modulspezifikationen, 1694 SLOC • Zwischenvortrag, 8 Folien
Implementierung	180	• Überarbeitete Entwurfsdok., 29 Seiten • Ada-Code, 14955+3054 SLOC
Bericht	170	• Bericht, 100 Seiten • Abschlußvortrag, 12 Folien
<b>Gesamt</b>	598	• 227 Seiten, 18009 SLOC, 34 Folien

**Tabelle 1: Aufwand und Ergebnisse der einzelnen Phasen**

Tabelle 2 zeigt einen Vergleich zwischen der geplanten und der tatsächlichen Aufwandsverteilung der fünf Phasen. Die geplante Aufwandsverteilung ergibt sich aus dem Anteil der geplanten Wochen pro Phase an der Gesamtbearbeitungszeit von 24 Wochen. Die Ist-Aufwandsverteilung errechnet sich als Anteil des Aufwands einer Phase am Gesamtaufwand. Diese Aufwandsdaten können Tabelle 1 entnommen werden.

Phase	Dauer (Wochen)	Aufwandanteil Soll (%)	Aufwandsanteil Ist (%)
Planung	1	4,2	1,7
Spezifikation	6	25,0	25,6
Entwurf	5	20,8	14,2
Implementierung	5	20,8	30,1
Bericht	7	29,2	28,4

**Tabelle 2: Soll- und Ist-Aufwandsverteilung**

Es fällt auf, daß sich der Aufwand zuungunsten der Entwurfs- und zugunsten der Implementierungsphase verschoben hat. Trotz der Beschränkung auf ein Minimalsystem hatte die Implementierung des Prototyps einen Zeitbedarf, der an der Grenze des für eine Diplomarbeit mit konzeptionellem Schwerpunkt vertretbaren lag.

Zusammengenommen liegen die beiden Phasen allerdings ungefähr im Plan. Angesichts der Fehler im Entwurf (vgl. Abschnitt 5.2.3.1), die während der Implementierung gefunden wurden, kann das Ergebnis so interpretiert werden, daß Aufwand, der eigentlich in der Entwurfsphase hätte investiert werden müssen, in die Implementierungsphase verschoben wurde.

In der Spezifikationsphase kam während der Bearbeitung die Anforderung der Erstellung einer *formalen* Spezifikation hinzu. Der für eine Formalisierung erforderliche zusätzliche Aufwand war im Plan nicht vorgesehen war. Trotzdem hat sich der Aufwandsanteil der Spezifikationsphase gegenüber dem Plan nicht signifikant erhöht. Die Umstellung auf eine formalisierte Darstellung der Semantik von BASE-2 hat wesentlich zur Präzision der Sprachbeschreibung beigetragen. Im Laufe der Formalisierung stellten sich nämlich einige Sprachkonstrukte als bisher ungenügend genau beschrieben heraus.

In Tabelle 3 wird der Aufwand der Phasen in Aufwand pro Woche umgerechnet. Diese Normalisierung des Aufwands dient dazu, die Arbeitsbelastung in den einzelnen Phasen vergleichbar zu machen.

Damit erhält man ein realistischeres Bild. Der Aufwand in der Implementierungsphase ist mit 36 Stunden pro Woche am höchsten, in der Planungsphase mit 10 Stunden pro Woche am niedrigsten. Der niedrige Aufwand der Planungsphase täuscht etwas, da in dieser Woche neben der Projektplanung eine intensive Einarbeitung in die bereits bestehenden SESAM-Systeme stattfand. Der Aufwand hierfür ist der Spezifikationsphase zugerechnet worden.

Der Aufwand in der Spezifikations- und der Berichtsphase entspricht etwa dem durchschnittlichen Aufwand von etwa 25 Stunden pro Woche.

Phase	Dauer (Wochen)	Aufwand (h)	Aufwand pro Woche (h)
Planung	1	10	10
Spezifikation	6	153	25,5
Entwurf	5	85	17
Implementierung	5	180	36
Bericht	7	170	24,3
<b>Gesamt</b>	24	598	24,9

**Tabelle 3: Aufwand pro Woche**

Insgesamt zeigt sich, daß der Projektplan im großen und ganzen eingehalten wurde. Dies recht gute Abschätzung der tatsächlichen Aufwandsverhältnisse durch den Plan (siehe Tabelle 2) ist meines Erachtens eher zufällig.

### 6.1.1.3 Untersuchung des Codes

Tabelle 4 zeigt die Zusammensetzung des Codes für die Basismaschine. Der Code für Testfälle im Rahmen des Modultest wurde nicht aufgenommen, da er größtenteils mit einem Code-Generator für Testfälle erzeugt wurde. Das hätte das Bild stark verzerrt.

Die Codezeilen werden in drei Kategorien unterschieden:

- Kommentarzeilen: Zeilen, die ausschließlich Kommentar enthalten,
- Leerzeilen: Zeilen, die keine Zeichen enthalten, und
- Code-Zeilen: die übrigen Zeilen.

Ein Befehl, zum Beispiel ein umfangreicher Prozeduraufruf, kann sich über mehrere Code-Zeilen erstrecken. Deshalb kann man noch alle Zeilen zählen, in denen ein Semikolon vorkommt. Dieses schließt in Ada 95 die meisten Befehle ab. Auf diese Weise erhält man eine minimale Anzahl der echten Code-Zeilen, während die Code-Zeilenzahl eine Obergrenze für die echten Code-Zeilen darstellt.

Neben der Unterscheidung in die genannten Zeilenkategorien läßt sich der Code auch in die Kategorien „generiert“ und „nicht generiert“ unterteilen. Generiert bedeutet hier, daß der Code von den Scanner- und Parsergeneratoren aus Beschreibungsdateien erzeugt wurde. Diese Beschreibungsdateien sind in der gesamten Code-Zahl nicht enthalten. In der Spalte „von Hand“ sind nicht generierter Code und Code für die Beschreibungsdateien des generierten Codes zusammengefaßt. Diese Zeilen sind diejenigen, die insgesamt von Hand eingegeben wurden.

Kategorie	gesamt	generiert	nicht gen.	von Hand
Kommentarzeilen	3536	1217	2319	2426
Leerzeilen	3065	1054	2011	2212
Code-Zeilen	8354	3511	4843	6232
<b>Gesamt</b>	14955	5782	9173	10870
Zeilen mit Semikolon	3875	1072	2083	3478

**Tabelle 4: Code-Zeilenverteilung (ohne Testfälle)**

Für eine Betrachtung der Produktivität sollen zunächst die gesamten Zeilen betrachtet werden. Bei einer Dauer der Implementierungsphase von 5 Wochen à 5 Tagen ergibt sich eine Produktivität von etwa 600 Zeilen pro Tag. Umgerechnet auf Code-Zeilen entspricht das einer Produktivität von etwa 330 Code-Zeilen pro Tag. Hier macht sich die Produktivitätssteigerung durch die Verwendung von Code-Generatoren bemerkbar. Betrachtet man die Zeilen, die von Hand eingegeben wurden, ergeben sich Produktivitäten von etwa 435 Zeilen pro Tag oder etwa 250 Code-Zeilen pro Tag. Es ergibt sich somit durch den Einsatz von Code-Generatoren auf den ersten Blick eine Produktivitätssteigerung um etwa ein Drittel. Diese Zahl ist jedoch mit Vorsicht zu genießen, da die Struktur des generierten Codes und die des von Hand verfaßten Codes nicht ganz vergleichbar sind.

## 6.1.2 Bewertung der Ergebnisse

### 6.1.2.1 BASE-2

Die im Rahmen der Arbeit entworfene Basissprache BASE-2 scheint insgesamt realisierbar und auch für die angestrebten Aufgaben brauchbar zu sein. Der Umfang der Implementierung umfaßt jedoch nur einen Teil der Basissprache, so daß keine abschließenden Aussagen gemacht werden können.

Hoare (1973) hat folgende Aussage gemacht (zitiert nach Ghezzi, Jazayeri, 1989, S. 479):

*Der Entwickler einer Sprache sollte mit vielen Alternativmöglichkeiten aus anderen Sprachen vertraut sein und ein Gespür dafür haben, jeweils die beste Wahl zu treffen und alle sich gegenseitig ausschließenden Möglichkeiten beiseite zu lassen. Er muß in der Lage sein, unvermeidbare kleinere Inkonsistenzen oder Überlappungen zwischen verschiedenen Entwurfsmerkmalen durch einen guten ingenieurmäßigen Entwurf auszugleichen. Er muß eine klare Vorstellung davon haben, welchen Anwendungsbereich und welchen Zweck seine neue Sprache verfolgt und wie weit sie im Hinblick auf Größe und Komplexität gehen darf. ...*

*Eins darf er nicht tun: unerprobte eigene Ideen einbringen. Seine Aufgabe ist Konsolidierung, nicht Innovation.*

Ich habe mich bemüht, diesem Ideal so nahe wie möglich zu kommen. Dabei half auch die Kenntnis unterschiedlichster Programmiersprachen, aus denen sich Konzepte entlehnen ließen. Folgt man dem Gedanken im letzten Absatz, stößt man aber unweigerlich auf das conforming-Konzept von BASE-2 (vgl. Abschnitt 4.2.2.2), das in den Verdacht kommt, eine solche neue, unerprobte Idee zu sein. Meines Erachtens handelt es sich beim conforming-Konzept nicht um

ein neues Konzept, sondern um eine – nach meiner Kenntnis – neue Notation für ein bekanntes, altbewährtes Konzept. Trotzdem handelt es sich um eine Neuerung, und damit um ein Risiko. Im Sinne der obigen Aussage von Hoare hätte man besser darauf verzichten und stattdessen mit Vererbung arbeiten sollen. Mir erschien jedoch der Flexibilitätsvorteil des conforming-Konzepts groß genug, um die Neuerung zu rechtfertigen.

### 6.1.2.2 Implementierung der Basismaschine

Die Brauchbarkeit der vorliegenden Implementierung der Basismaschine muß als recht niedrig eingestuft werden. Dies liegt an den Einschränkungen der Implementierung, an der unzureichenden Code-Qualität und der mangelnden Effizienz.

Klassifiziert als experimenteller Prototyp kann sich die Implementierung dagegen sehen lassen, da das erstellte Programm die gewünschte Funktionalität bietet und damit die Brauchbarkeit von BASE-2 demonstriert werden kann. Außerdem kann die vorliegende Implementierung durchaus als Grundlage einer Neuimplementierung verwendet werden (vgl. hierzu Abschnitt 6.2.3).

## 6.2 Ausblick

### 6.2.1 Erweiterungen von BASE-2

In diesem Abschnitt werden Ideen zusammengefaßt, die bisher nicht in BASE-2 integriert wurden. Zum größeren Teil stellen diese Ideen Erweiterungen dar, die zu mehr Entwicklungskomfort und zu besserer Lesbarkeit der Modellbeschreibungen führen würden. Der Rest betrifft notwendige Bestandteile, die bei der Konzeption außer acht gelassen wurden.

#### 6.2.1.1 Vererbung

Anstelle von Vererbung unter Entitäts- und Relationstypen ist in BASE-2 das Typverträglichkeitskonzept (siehe Abschnitt 4.2.2.3) eingeführt worden. Sollte sich herausstellen, daß die Flexibilität dieses Konzepts nicht benötigt wird, kann es zugunsten einer Einfachvererbung fallen gelassen werden.

Die Modellbeschreibungssprache von Bernd Schneider (1996a), die erste bisher entworfene Hochsprache, wird nur Einfachvererbung besitzen. Von daher wäre eine Übersetzung in BASE-2 einfacher, wenn BASE-2 ebenfalls über Einfachvererbung verfügen würde.

#### 6.2.1.2 Nicht-Strukturen

Im Strukturbedingungsteil von Regeln können Strukturen formuliert werden, die in der aktuellen Situation vorhanden sein müssen, damit die Regel ausgeführt werden darf. Für manche Anwendungen ist es zweckmäßig, auch das Fehlen bestimmter Strukturen zur Voraussetzung der Ausführung einer Regel zu machen. Eine solche Struktur, die nicht vorhanden sein soll, wird Nicht-Struktur genannt.

In BASE-2 ist es möglich, mit Hilfe des `exists`-Prädikats unerwünschte Relationen zwischen formalen Entitäten zu erkennen. Dazu nimmt man die Bedingung

```
not exists(<relation>);
```

in den Attributbedingungsteil auf. Dieses Verfahren funktioniert jedoch nicht für Nicht-Strukturen, die auch Entitäten enthalten.

Theoretisch ist ein Verfahren zur Erkennung von Nicht-Strukturen möglich, das auf den vorhandenen Sprachkonstrukten der Basissprache BASE-2 aufbaut (vgl. hierzu Reißing, 1996, Anhang F). Die dazu notwendigen Erweiterungen eines Modells sind jedoch sehr kompliziert und für einen Leser der Modellbeschreibung kaum mehr verständlich, so daß es meines Erachtens notwendig ist, eine eigene Notation einzuführen, wenn sich ein dringender Bedarf nach der Formulierung von Nicht-Strukturen ergeben sollte.

### 6.2.1.3 Manipulation der Modellzeit

Bisher ist es in BASE-2 nicht möglich, die Modellzeit explizit zu verändern. Die einzige Möglichkeit zur Fortschaltung der Modellzeit besteht in der Ausführung einer Regel oder eines Benutzerkommandos mit einem Zeitverbrauch größer 0.

In SESAM-1 gab es die Möglichkeit, durch ein spezielles Kommando zum nächsten Tag der Simulation fortzuschreiten. Dies müßte in einem BASE-2-Modell durch ein Benutzerkommando realisiert werden, das durch eine besondere Manipulation der aktuellen Situation eine Regel anstößt. Diese „Fortschaltregel“ verbraucht so lange Zeit, bis der nächste Tag erreicht ist. Damit während des Fortschaltens keine Benutzerkommandos akzeptiert werden, muß der Simulationsbedarf  $c$  über der Simulationsschrittweite gehalten werden. Der Zeitverbrauch der Fortschaltregel und des Benutzerkommandos wird also zweckmäßigerweise auf die Simulationsschrittweite gesetzt.

Bei der hier skizzierten Lösung sind noch nicht alle Details durchdacht, sie scheint jedoch bereits jetzt recht kompliziert zu sein. Besser wäre es, bereits in der Basissprache eine Anweisung zu haben, die ein Fortschreiten der Simulation ohne Benutzereingriff bis zu einem gewünschten Zeitpunkt veranlaßt. Diese Anweisung könnte dann in dem Benutzerkommando zur Zeitweitschaltung aufgerufen werden. Der Ausführungsalgorithmus muß dann allerdings um einen speziellen Modus erweitert werden, in dem keine Benutzerkommandos akzeptiert werden und grundsätzlich sofort der nächste Simulationsschritt vorgenommen wird, bis die gewünschte Zeit erreicht oder überschritten wurde.

### 6.2.1.4 Abfragbarkeit der Simulationsschrittweite

Ein weiterer Aspekt in Zusammenhang mit der Modellzeit ist die Frage, ob es nicht notwendig ist, neben der aktuellen Modellzeit auch die Simulationsschrittweite durch eine vordefinierte Funktion abfragbar zu machen.

Dies hätte den Vorteil, daß bei der Formulierung von Veränderungen von Attributen die Simulationsschrittweite als Faktor einfließen kann. Ein solches Vorgehen ist in Sprachen zur Beschreibung diskreter Simulationsmodelle allgemein üblich. Auf diese Weise kann nämlich die Simulationsschrittweite verändert werden, ohne das Modell anpassen zu müssen.

Die Verfügbarkeit der Simulationsschrittweite über eine vordefinierte Funktion dient jedoch nur dem Komfort des Modellierers, da eine automatische Umsetzung der Modellanpassung auf die gewählte Simulationsschrittweite durch den Hochsprachenübersetzer vorgenommen werden kann. Dies ist möglich, da es sich zum Zeitpunkt der Übersetzung einer Modellbeschreibung in die Basissprache bei der Simulationsschrittweite um eine Konstante handelt.

### 6.2.1.5 Protokoll des Spiels

Bei der bisherigen Definition von BASE-2 wurde die Erzeugung des Protokolls völlig ausgelassen. Das Protokoll wird zur Auswertung eines Spiels benötigt. Es soll dazu die Werte aller oder ausgewählter Attribute in jedem Simulationsschritt enthalten.

Eine Möglichkeit ist es, die Protokollerzeugung in die dynamische Semantik der Basissprache aufzunehmen, indem zum Beispiel festgelegt wird, daß sämtliche Attribute aller Entitäten und Relationen bei der Initialisierung und nach jedem Simulationsschritt festgehalten werden.<sup>1</sup> Dieses Vorgehen erzeugt ein umfassendes Protokoll, das man jedoch auch als Datenwüste bezeichnen könnte: die gewünschte Information geht in der Masse der Daten unter.

Eine andere Möglichkeit besteht in der Kennzeichnung von Attributen bei der Typdeklaration, zum Beispiel durch ein spezielles Schlüsselwort. Die Werte derart gekennzeichnete Attribute werden nach jedem Simulationsschritt protokolliert. Auf diese Weise wird die Spreu vom Weizen getrennt, das Protokoll umfaßt nur die gewünschten Attribute.

Zusätzlich sollte es noch die Möglichkeit geben, vom Modell aus beliebige Kommentare in das Protokoll schreiben zu können. Hierfür ist in der Modellbeschreibungssprache von Bernd Schneider (1996a) die Prozedur `SendTutorMessage` vorgesehen, der man eine beliebige Zeichenkette übergeben kann. Auf diese Weise können zum Beispiel Regeln und Benutzerkommandos ihre Ausführung im Protokoll festhalten. Die Basissprache sollte ebenfalls ein solches Konzept zur Kommentierung des Protokolls anbieten.

## 6.2.2 Erweiterungen der Basismaschine

Dieser Abschnitt enthält mögliche Erweiterungen der Basismaschine, die im Rahmen der Konzeption angedacht, aber bisher nicht in die Basismaschine integriert wurden.

### 6.2.2.1 Abgleich des Wörterbuchs mit dem Modell

Die statischen Prüfungen einer Modellbeschreibung durch die Basismaschine vernachlässigen bisher einen wichtigen Aspekt: Es findet kein Abgleich der Benutzerkommandos und der Nachrichten im Modell mit dem Dolmetscherwörterbuch statt, das in der Modellbeschreibung angegeben werden muß. Deshalb kann es während der Modellausführung zu Verständigungsschwierigkeiten zwischen der Basismaschine und dem Dolmetscher kommen. Es erscheint jedoch zweckmäßig, die Sicherstellung der Kompatibilität von Modell und Wörterbuch den höheren Ebenen der Modellerstellung in SESAM-2 zu überlassen.

### 6.2.2.2 Spielstände

Der Basismaschine fehlen einige wichtige Eigenschaften, um ein komfortablen Umgang des Spielers mit den Modellen zu ermöglichen. Es fehlt zum einen die Möglichkeit, vergangene Spiele nachzuspielen, und zum anderen kann ein angefangenes Spiel auch nicht gespeichert werden, um es ein anderes Mal fortzusetzen.

---

1. Die Entitäten werden dabei durch ihren Typ und ihren externen Namen benannt, die Relationen durch ihren Typ und die externen Namen und Typen ihrer Rollenbelegungen. Die Benennung ist notwendig, um die Attributwerte einzelnen Komponenten zuordnen zu können.

Die beiden genannten Punkte lassen sich jedoch nicht in der Basismaschine allein realisieren, da sie den Dolmetscher als Benutzerschnittstelle ebenfalls betreffen. Bisher ist es vom Dolmetscher aus nicht möglich, ein Nachspielen vergangener Spiele oder das Laden und Speichern von Spielständen zu veranlassen.

Das mindeste scheint es zu sein, Spielstände abspeichern und wieder laden zu können. Dies ist in jedem gängigen Abenteuerspiel (*Adventure*) möglich. Das Dolmetscher-Protokoll<sup>1</sup> muß zu diesem Zweck durch neue Befehle erweitert werden, da die Aufforderung zum Laden oder Speichern durch den Benutzer sinnvollerweise ebenfalls vom Dolmetscher entgegengenommen werden sollte.

Um ein Nachspielen vergangener Spiele zu ermöglichen, könnte der Dolmetscher ein Protokoll sämtlicher Benutzereingaben mitführen und dieses auf Verlangen Befehl für Befehl abarbeiten.

### 6.2.3 Neuimplementierung

Es ist bereits geplant, die Basismaschine in einer weiteren Diplomarbeit neu zu implementieren, wobei auch ein möglichst effizienter Regelausführungsalgorithmus realisiert werden soll.

Ich empfehle nach den gemachten Erfahrungen, bei der Wahl der Architektur der Basismaschine dem Interpreter-Ansatz den Vorzug zu geben. Das entwickelte System dürfte um einiges übersichtlicher werden und auch bleiben, wenn es zum Beispiel um die Realisierung neuer Sprachkonstrukte erweitert werden muß.

Dennoch dürfte es möglich sein, einige Bestandteile der Implementierung im Rahmen dieser Arbeit für die Implementierung der neuen Basismaschine wiederzuverwenden. Besonders eignen sich dafür die Module des Code-Generators, die für die Programmanalyse zuständig sind, und Teile des Laufzeitsystems wie der Animator und die Schnittstelle zum Dolmetscher.

---

1. Das Dolmetscher-Protokoll legt fest, welche Nachrichten sich Dolmetscher und Basismaschine gegenseitig schicken können und welche Bedeutung diese Nachrichten haben. Das bisherige Protokoll ist in Spiegel (1995, Anhang B) beschrieben.

# Anhang

## A Glossar

In diesem Glossar sind Begriffe zusammengefaßt, die

- nicht gebräuchlich sind,
- in dieser Arbeit in einem speziellen Sinne verwendet werden oder
- erst in der Arbeit definiert wurden.

Begriffe, die vor allem im Zusammenhang mit der Basissprache BASE-2 eingeführt und verwendet werden, sind durch die Anmerkung „(BASE-2)“ gekennzeichnet.

**Aggregationsniveau.** Beschreibt den Detaillierungsgrad eines Modells. Je mehr Bestandteile des Original in einem Modell zusammengefaßt (aggregiert) werden, desto höher ist das Aggregationsniveau des Modells.

**Aktion.** (BASE-2) Anweisung im Aktionsteil. Aktionen sind Zuweisungen an Attribute, Entfernen und Löschen von Entitäten und Relationen sowie das Versenden von Nachrichten.

**Aktionsteil.** (BASE-2) Der Teil einer Regel oder eines Benutzerkommandos, der die Aktionen enthält. Der Aktionsteil wird ausgeführt, wenn der Bedingungsteil der Regel oder des Benutzerkommandos erfüllt ist.

**Aktivierung/Deaktivierung.** Spezielle Semantik der Regelausführung. Der Aktionsteil einer Regel zerfällt in drei Teile: einen *Aktivierungsteil*, einen *Aktivteil* und einen *Deaktivierungsteil*. Findet eine Regel einen passenden Teilzustand vor, aktiviert sie sich. Dabei führt sie den Aktivierungsteil aus. Solange der Bedingungsteil erfüllt ist, bleibt die Regel aktiv. Nun führt sie in jedem weiteren Simulationsschritt den Aktivteil aus. Wird der Modellzustand durch die Regel selbst oder durch andere Regeln so verändert, daß der Bedingungsteil nicht mehr erfüllt ist, so deaktiviert sich die Regel. Dabei führt sie den Deaktivierungsteil aus.

**Aktivität.** Faßt Regeln zusammen, die einer übergeordneten Tätigkeit zugeordnet werden können, zum Beispiel der Spezifikationsphase eines Projekts. Der Aktivitätenansatz stammt aus der neuen Modellbeschreibungssprache von Bernd Schneider (1996a, Kapitel 14).

**Animator.** Der Teil der Basismaschine, der für die Modellausführung zuständig ist.

**Attribut.** (BASE-2) Attribute dienen der Beschreibung von Eigenschaften einer Entität oder Relation. Sie haben einen Namen und einen Attributtyp.

**Attributbedingungsteil.** (BASE-2) Teil des Bedingungsteils einer Regel oder eines Benutzerkommandos. Enthält Bedingungen über die Attribute der formalen Entitäten und Relationen, die erfüllt sein müssen, damit die Regel ausgeführt werden kann.

**Attributtyp.** (BASE-2) Typ für Attribute. Attributtypen können Basistypen oder Listentypen sein.

- Ausführungsalgorithmus.** Algorithmus, der die oberste Ebene der dynamischen Semantik eines BASE-2-Modells definiert. Legt fest, wann und wie Benutzerkommandos und Regeln ausgeführt werden.
- Auswertungswerkzeug.** Werkzeug, mit denen der Tutor Auswertungen gemachter Spiele durchführen kann. Grundlage der Auswertung ist ein Protokoll des Spiels.
- BASE-2.** Die im Rahmen der Arbeit entworfene Basissprache. BASE-2 ist eine Abkürzung für „Basissprache für SESAM-2“.
- Basismaschine.** Werkzeug zur Ausführung von Modellbeschreibungen in Basissprache.
- Basissprache.** Modellbeschreibungssprache der Basismaschine. Besitzt im Vergleich zur Hochsprache wenige Konzepte. Zielsprache der Übersetzung einer Hochsprache.
- Basistyp.** (BASE-2) Vordefinierter Attributtyp. In BASE-2 sind dies die Typen Boolean, Integer, Real, String, Date Basistypen.
- Bedingungsteil.** (BASE-2) Teil der Regel, der bestimmt, ob sie ausgeführt werden darf. Der Bedingungsteil besteht aus dem Strukturbedingungsteil und dem Attributbedingungsteil. Damit die Regel ausgeführt werden darf, muß der Bedingungsteil erfüllt sein. Dies bedeutet, daß eine Bindung des Strukturbedingungsteil gefunden wird und der Attributbedingungsteil für diese Bindung erfüllt ist.
- Benutzerdefinierte Funktion.** (BASE-2) Eine Funktion über Attributtypen. Benutzerdefinierte Funktionen bestehen aus einem Namen, formalen Parametern, einem Rückgabotyp und dem Funktionsrumpf, in dem der Rückgabewert berechnet wird.
- Benutzerkommando.** (BASE-2) Parametrisierte Regel, deren Ausführung nur vom Benutzer veranlaßt werden kann. Dabei legt der Benutzer die Parameterbelegung fest. Der Aufbau eines Benutzerkommandos ist, abgesehen von den zusätzlichen formalen Parametern und der fehlenden Priorität, analog zu dem einer Regel.
- Benutzerkommandoinstanz.** (BASE-2) Instanz eines Benutzerkommandos. Die formalen Parameter sind belegt und der Bedingungsteil ist erfüllt.
- Bindung.** (BASE-2) Isomorphe Abbildung der formalen Komponenten des Strukturteils in die aktuelle Situation. Jeder formalen Komponente ist genau eine Komponente der aktuellen Situation zugeordnet, die einen typverträglichen Typ aufweist.
- Code-Generator-Ansatz.** Eine mögliche Architektur der Basismaschine. Aus der Modellbeschreibung in Basissprache wird Ada-95-Code erzeugt, der sich in ein ausführbares Modell übersetzen läßt.
- Definition.** (BASE-2) Oberbegriff für (die Definition von) Attributtypen, Entitätstypen, Relationstypen, Regeln, Benutzerkommandos und benutzerdefinierter Funktionen.
- Definitionsteil.** (BASE-2) Teil einer Modellbeschreibung. Umfaßt die Definitionen für das Schemamodell (Attributtypen, Entitätstypen und Relationstypen) und das Regelmodell (Regeln, Benutzerkommandos und benutzerdefinierter Funktionen).

**Diskretes Simulationsmodell.** Spezielles Simulationsmodell. Das Modell besitzt einen Modellzustand, der sich aus verschiedenen Zustandsgrößen zusammensetzt. Die Simulation wird in diskreten Schritten durchgeführt. In jedem Simulationsschritt wird aus dem momentanen Modellzustand der nächste Zustand berechnet.

**Dokument.** Oberbegriff für schriftlich und elektronisch vorliegende Unterlagen eines Software-Projekts. Das können zum Beispiel die Spezifikation oder der Programm-Code, aber auch Ausschreibungen für Software-Entwicklungswerkzeuge sein.

**Dolmetscher.** Werkzeug, das zwischen dem Benutzer und der Basismaschine vermittelt. Nimmt Eingaben des Benutzers entgegen und wandelt sie in Kommandos an die Basismaschine um. Umgekehrt schickt die Basismaschine Nachrichten an den Dolmetscher, der sie in Ausgaben an den Benutzer übersetzt. Zur Vermittlung bedient sich der Dolmetscher des Wörterbuchs.

Aus Benutzersicht ist der Dolmetscher eine natürlichsprachliche Benutzungsschnittstelle zum SESAM-System. Näheres siehe Spiegel (1995).

**Dynamisches Modell.** Ein vollständiges SESAM-Modell, bestehend aus einem Schemamodell und einem dazu passenden Regel- und Situationsmodell. Das Situationsmodell beschreibt den Anfangszustand des Modells.

**Einfachvererbung.** Form der Vererbung, bei dem eine Klasse nur von *einer* anderen Klasse erben kann.

**Entität.** (BASE-2) Instanz eines Entitätstyps.

**Entitätstyp.** (BASE-2) Typen für die Objekte der abstrakten Welt, die durch das Modell definiert ist und in der die Dynamik des Modells abläuft. Bestehen aus einem Namen, einer optionalen Typverträglichkeitsklausel und der Deklaration von Attributen.

**Entity-Relationship-Modell.** Allgemeines computerunabhängiges Datenmodell. Ein Ausschnitt der realen Welt wird durch abgrenzbare, individuelle Objekte modelliert, die untereinander Beziehungen haben. Die Objekte heißen Entities, die Beziehungen Relationships. Entities und Relationships werden durch Attribute charakterisiert. (nach Duden Informatik, 1993, S. 231 ff.; siehe auch dort)

**Event.** Ereignis. In SESAM-1 und in SESAM-Lite sind Eventtypen Bestandteil des Schemamodells (vgl. hierzu Schneider, 1994, Abschnitt 4.3.3). Es werden exogene Events und endogene Events unterschieden.

Exogene Events sind Ereignisse, die außerhalb des Modells erzeugt werden und dann auf das Modell einwirken. Diese dienen dazu, die Auswirkungen von Benutzeraktionen zu modellieren.

Endogene Events sind Ereignisse, die im Modell selbst erzeugt werden. Sie dienen vor allem dazu, Ereignisse, die irgendwann in der Zukunft eintreten sollen, zu modellieren.

**Externer Name.** (BASE-2) Name, unter dem eine Entität nach außen, d. h. dem Dolmetscher und dem Benutzer, bekannt ist.

**Feuern.** Spezielle Semantik der Regelausführung. Eine Regel feuert, wenn ihr Bedingungsteil erfüllt ist. Feuern bedeutet, daß der Aktionsteil der Regel ausgeführt wird. Danach wartet die Regel darauf, wieder feuern zu können. Regeln in BASE-2 feuern.

**Formale Entität.** (BASE-2) Bestandteil des Strukturbedingungsteils einer Regel. Besteht aus einem Namen und einem Entitätstyp.

**Formaler Parameter.** (BASE-2) Bestandteil eines Benutzerkommandos. Dient dazu, Parameter eines Kommandos vom Benutzer aufzunehmen.

**Formale Relation.** (BASE-2) Bestandteil des Strukturbedingungsteils einer Regel. Besteht aus einem Namen, einem Relationstyp und einer Rollenbelegung durch formale Entitäten.

**Geteiltes mentales Modell.** Eine Art Schnittmenge der individuellen mentalen Modelle (bezogen auf einen Gegenstand) der Mitglieder einer Gruppe, die an einem gemeinsamen Problem arbeiten.

**Graphgrammatik.** Grammatik, die im Unterschied zu den Chomsky-Grammatiken nicht über Zeichenketten, sondern über Graphen operiert. Näheres zu Graphgrammatiken kann Götter (1988) und Schürr, Westfechtel (1992) entnommen werden.

**Gültige Bindung.** Eine Bindung eines Strukturbedingungsteils einer Regel, zu der alle Bedingungen des Attributbedingungsteil erfüllt sind.

**Hochsprache.** Modellbeschreibungssprache von SESAM-2. Besitzt im Vergleich zur Basissprache viele, reichhaltige Konzepte. Kann durch einen Hochsprachenübersetzer in Basissprache übersetzt werden.

**Hochsprachenübersetzer.** Werkzeug, das eine Modellbeschreibung in Hochsprache in eine Modellbeschreibung in Basissprache übersetzt.

**Hypergraph.** Graph, bei dem eine Kante nicht wie üblich genau zwei, sondern beliebig viele Knoten verbinden kann.

**Individuelles mentales Modell.** Vereinfachte, erfahrungsbasierte Repräsentation der Wirklichkeit, die jedem Verstehen und Entscheiden zugrunde liegt. Ein solches Modell kann durchaus Widersprüche enthalten.

**Instanz.** Konkrete Wertausprägung eines Typs oder einer Klasse. Die Instanz einer Klasse wird auch als Objekt bezeichnet.

**Interpreter-Ansatz.** Eine mögliche Architektur der Basismaschine. Die Modellbeschreibung in Basissprache wird von einem Interpreter eingelesen und anschließend von ihm ausgeführt.

**Klasse.** hier: Bezeichnung eines Typs in der objektorientierten Begriffswelt.

**Kommando.** Ergebnis der Übersetzung einer Benutzereingabe durch den Dolmetscher, das an die Basismaschine weitergeleitet wird und dort die Ausführung eines Benutzerkommando auslösen kann.

**Komponente.** (BASE-2) Oberbegriff für Entitäten und Relationen. Das Situationsmodell besteht im wesentlichen aus Komponenten.

**Lernlabor.** Konzept zum Einsatz eines Lernmodells. In einem Lernlabor wird die Auseinandersetzung mit dem Lernmodell vorbereitet, begleitet und nachbereitet. Die Spieler tauschen in Zwischen- und Abschlußbesprechungen ihre gewonnenen Erkenntnisse untereinander aus. Der Tutor stellt das dem Lernmodell zugrundeliegende qualitative Modell mit seinen Wirkungsbeziehungen vor.

**Lernmodell.** Ein Simulationsmodell mit Benutzeroberfläche, das auf einem Rechner simuliert wird. Es dient der spielerischen Erfahrung eines Ausschnitts der Realität, der so komplex ist, daß er nicht sofort verstandesmäßig erfaßt werden kann.

Ein bekanntes Beispiel für solche Lernmodelle sind die sogenannten „Management Flight Simulators“, bei denen der Spieler die Rolle eines Unternehmensführers übernimmt und strategische und operative Entscheidungen für das Unternehmen fällen muß.

**Listentyp.** (BASE-2) Benutzerdefinierter Attributtyp. Benanntes Ergebnis der Anwendung des Typkonstruktor Liste auf einen Basistyp.

**Management Flight Simulator.** Lernmodell für die Führung eines Unternehmens. Ein Beispiel eines solchen Management Flight Simulators ist „People Express“ von John D. Sterman. Dort kann man den Geschäftsleiter der Luftfahrtgesellschaft People Express spielen.

**Mehrfachvererbung.** Form der Vererbung, bei dem eine Klasse von einer oder mehreren anderen Klassen erben kann.

**Modell.** Abbild eines Ausschnitts der Realität, des Originals. Dient zur Veranschaulichung oder Erklärung bestimmter Verhaltensweisen des Originals.

**Modellbeschreibungssprache.** Sprache zur Beschreibung eines Modells. In SESAM-2 werden zwei Ebenen unterschieden: Hoch- und Basissprache.

**Modellierer.** Modellersteller. hier: Benutzer der Modellierungswerkzeuge.

**Modellierung.** Vorgang der Modellerstellung. hier: Erstellung eines Simulationsmodells.

**Modellierungswerkzeuge.** Werkzeuge, die zur Erstellung einer Modellbeschreibung in Hochsprache dienen.

**Modellzeit.** Zeitmarkierung des Modellzustands eines Simulationsmodells. In BASE-2 ein Date-Wert.

**Nachricht.** Mitteilung des SESAM-Modells an den Benutzer, die durch den Dolmetscher in eine verständliche Form gebracht und ausgegeben wird.

**Namensäquivalenz.** Eine Form der Typäquivalenz von Instanzen. Zwei Instanzen sind genau dann äquivalent, wenn ihre Typen denselben Namen haben.

**Nicht-Struktur.** (BASE-2) Struktur, die nicht vorhanden sein darf, damit der Strukturbedingungsteil einer Regel erfüllt ist.

**Objekt.** In der objektorientierten Begriffswelt Instanz einer Klasse.

**Original.** Ausschnitt der realen Welt, der zum Gegenstand einer Modellierung gemacht wird.

**Parser.** Der Teil der Basismaschine, der die Modellbeschreibung einliest und auf statisch prüfbare Fehler untersucht. Er übergibt eine Modellbeschreibung an den Animator.

**Priorität.** (BASE-2) Bestandteil einer Regel. Je höher die Priorität einer Regel ist, desto bevorzugter wird sie bei der Regelausführung berücksichtigt.

**Produktqualität.** Aspekt der Qualität einer Software. Die Produktqualität ist hoch, wenn die Software alle Kundenanforderungen in angemessener Weise realisiert.

**Projekt.** hier: Software-Projekt. „Ein Projekt ist die Menge aller Tätigkeiten, Interaktionen und Resultate, die mit dem Versuch zusammenhängen, ein bestimmtes Ziel mit begrenzten Mitteln und innerhalb begrenzter Zeit zu erreichen. Bei dem hier zu betrachtenden Software-Projekt ist das Ziel die Bereitstellung eines Software-Systems oder -Produkts [...].“ (Frühauf, Ludewig, Sandmayr, 1988, S. 10)

**Protokoll.** Von der Basismaschine geführte Aufzeichnungen über die Werte von Attributen während der Modellsimulation. Ist Eingabe für die Auswertungswerkzeuge und dient der Auswertung eines Spiels.

**Prozeßqualität.** Aspekt der Qualität einer Software. Die Prozeßqualität ist hoch, wenn das Software-Projekt in der geplanten Zeit mit den geplanten Verbrauch an Ressourcen abgeschlossen werden kann.

**Qualitatives Modell.** Modell, das durch qualitative Aussagen über seine Bestandteile und ihrer Beziehungen untereinander charakterisiert ist. Es können Aussagen über Zusammenhänge gemacht werden, diese Aussagen können jedoch nicht quantifiziert werden.

**Quantitatives Modell.** Modell, das durch formale, quantitative Aussagen über seine Bestandteile und ihre Beziehungen untereinander charakterisiert ist. Quantitative Modelle können durch Formulierung in einer geeigneten Sprache von einem Rechner simuliert werden.

**Redefinition (einer Rolle).** (BASE-2) Deklaration einer Rolle von einem typverträglichen Typ zu dem Typ, den die Rolle in einem typverträglichen Relationstyp hatte.

**Regel.** (BASE-2) Eine Regel besteht aus einem Regelkopf, einem Bedingungs- und einem Aktionsteil. Der Aktionsteil einer Regel wird ausgeführt, wenn der Bedingungssteil erfüllt ist. Regeln erzeugen die Modelldynamik eines SESAM-Modells.

**Regelausführungsalgorithmus (auch: Regelausführungsmechanismus).** (BASE-2) Algorithmus, der festlegt, wann welche Regel ausgeführt wird. In jedem Simulationsschritt wird der Regelausführungsalgorithmus genau einmal angestoßen.

**Regelinstanz.** (BASE-2) Instanz einer Regel, deren Bindung gültig ist.

**Regelkopf.** (BASE-2) Der Regelkopf enthält den Namen der Regel, ihre Priorität und ihren Zeitverbrauch.

**Regel-Matching.** (BASE-2) Ermittlung aller möglichen Instanzen aller Regeln in der aktuellen Situation.

**Regelmodell.** Teil der dynamischen Modells. Enthält alle Regeln. In BASE-2 zusätzlich die Benutzerkommandos und die benutzerdefinierten Funktionen.

**Relation.** (BASE-2) Instanz eines Relationstyps.

**Relationstyp.** (BASE-2) Typ für die Beziehungen zwischen den Objekte der abstrakten Welt des Modells, also für Beziehungen zwischen Entitäten. Relationstypen bestehen aus einem Namen, einer optionalen Typverträglichkeitsklausel und der Deklaration von Rollen und Attributen.

**Rolle.** (BASE-2) Bestandteil einer Relation. Eine Rolle hat einen Namen und einen Typ, der ein Entitätstyp sein muß. Rollen nehmen die an der Beziehung beteiligten Entitäten auf.

**Schemamodell.** Beschreibt ähnlich einem Entity-Relationship-Modell die abstrakte Welt, in der das Modell abläuft. Besteht aus Entitätstypen und Relationstypen. In BASE-2 kommen noch die Attributtypen zum Schemamodell hinzu.

**SESAM.** Abkürzung für „Software Engineering Simulation by Animated Models“. Projekt der Abteilung Software Engineering des Instituts für Informatik der Universität Stuttgart.

**SESAM-1.** Erste Implementierung des SESAM-Systems in Smalltalk-80. Dient als Pilotsystem für weitere Implementierungen. Zur Zeit die in Forschung und Lehre verwendete Version.

**SESAM-2.** Geplantes, bisher nur ansatzweise realisiertes SESAM-System. Soll SESAM-1 auf lange Sicht ablösen. Ablösung der Implementierungssprache Smalltalk-80 durch Ada 95.

**SESAM-Lite.** Prototyp für SESAM-2 in Smalltalk-80.

**SESAM-Modell.** Ein Simulationsmodell, das durch das SESAM-System animiert werden kann. Entspricht einem dynamischen Modell.

**SESAM-Projekt.** Projekt der Abteilung Software Engineering, in dessen Rahmen das SESAM-System entsteht. Umfaßt auch die Forschungstätigkeiten zur Modellerstellung und -validierung.

**SESAM-System.** Konkrete Implementierung der Konzepte von SESAM. Umfaßt Aspekte der Modellerstellung und Modellanimation sowie der Spielauswertung.

**Simple Lines of Code (SLOC).** Maß für den Umfang von Quell-Code. Zählt alle Zeilen des Quell-Codes ohne Unterscheidung in Kommentarzeilen, Leerzeilen und Code-Zeilen.

**Simulation.** hier: Die Ausführung (Animation) eines Simulationsmodells.

**Simulationsbedarf.** (BASE-2) Zählgröße für Zeitverbrauch von Regeln und Benutzerkommandos. Anhand des Simulationsbedarfs wird festgestellt, ob ein Simulationsschritt durchgeführt werden muß.

**Simulationsmodell.** Modell, das durch einen Rechner ausgeführt (simuliert) werden kann.

**Simulationsschritt.** Bei diskreten Modellen der Vorgang der Berechnung eines Folgezustands aus dem momentanen Modellzustand anhand der Berechnungsvorschriften des Modells. Bei einem Simulationsschritt wird die Modellzeit um die Simulationsschrittweite fortgeschaltet.

**Simulationsschrittweite.** Bei diskreten Modellen die Zeitspanne, die zur Modellzeit hinzugezählt wird, wenn ein Simulationsschritt durchgeführt wird.

**Simulator.** Werkzeug in SESAM-1, das für die Modellausführung zuständig ist.

**Situationsmodell.** Repräsentiert den eigentlichen Modellzustand, auf den das Regelmodell angewandt wird. Es enthält Instanzen der im Schemamodell eingeführten Entitäts- und Relationstypen, ist also eine Ausprägung des Schemamodells. Zusätzlich umfaßt das Situationsmodell die Modellzeit.

**Spieler.** Benutzer des SESAM-Systems, der den Projektleiter des simulierten Projekts mimit. Kommuniziert über den Dolmetscher mit einem SESAM-Modell.

**Spielstand.** Modellzustand (in Form eines Situationsmodells) eines SESAM-Modells während eines Spiels.

**Startdatum.** (BASE-2) Anfangswert der Modellzeit.

**Startsituation.** Situationsmodell, das den Anfangszustand eines dynamischen Modells beschreibt.

**Startsituationsteil.** (BASE-2) Umfaßt die Startsituation in Form von Komponentenerzeugungsanweisungen, das Startdatum, die Simulationsschrittweite und den Namen des Wörterbuchs des Dolmetschers.

**Strukturäquivalenz.** Eine Form der Typäquivalenz von Instanzen. Zwei Instanzen sind genau dann äquivalent, wenn ihre Typen dieselbe Struktur haben. Dies ist dann der Fall, wenn sich die Typen durch die Anwendung derselben Typkonstruktoren auf dieselben Basistypen ergeben.

**Strukturbedingungsteil.** (BASE-2) Teil des Bedingungsteils einer Regel. Definiert die Struktur, die in der aktuellen Situation vorgefunden werden muß, damit die Regel feuern kann. Die Struktur wird durch formale Entitäten und formale Relationen beschrieben.

**System.** hier: „Zusammenfassung mehrerer Komponenten zu einer als ganzes aufzufassenden Einheit.“ (Duden Informatik, 1993, S. 717)

**System Dynamics.** Von J. W. Forrester begründeter Ansatz zur Erstellung von Simulationsmodellen. Ein System-Dynamics-Modell besteht aus Zustandsgrößen (sogenannte Levels), die den Modellzustand bilden, und Flußgrößen (sogenannte Raten), die die Veränderung des Modellzustands über die Zeit beschreiben. Vgl. Forrester (1971 und 1974).

**Systemgrenzen.** hier: Grenzen des betrachteten Ausschnitts der realen Welt. Modelliert wird nur innerhalb der Systemgrenzen.

**Tutor.** Betreuer (Spielleiter) bei Spielen mit dem SESAM-System. Benutzt die Auswertungswerkzeuge zur Analyse der Spiele.

**Typ.** Hier verwendet im Sinne von Datentyp. Jede Instanz, auch Wert genannt, besitzt einen Typ.

**Typkonstruktor.** Konzept zur Erzeugung neuer Typen auf der Grundlage bisher vorliegender Typen. Beispiele für Typkonstruktoren sind Felder (Array) oder Verbunde (Record). Aufzählungstypen können auch als Ergebnis der Anwendung eines Typkonstruktors aufgefaßt werden.

**Typverträglichkeit.** Eine Beziehung zwischen Typen. Wenn ein Typ *A* zu einem Typ *B* typverträglich ist, kann ein Objekt vom Typ *A* anstelle eines Objekts vom Typ *B* verwendet werden, da *A* alle Eigenschaften von *B* zur Verfügung stellt. Jeder Typ ist zu sich selbst typverträglich.

**Validierung.** Vorgang der Überprüfung eines Simulationsmodells im Hinblick auf die Entsprechung mit dem Original. Zum Beispiel werden Situationen, für die das Verhalten des Originals bekannt ist, am Modell nachvollzogen und das Originalverhalten mit dem Modellverhalten verglichen.

**Vererbung.** Begriff aus der objektorientierten Begriffswelt. Eine Klasse kann ihre Eigenschaften (Attribute und Operationen) an eine andere Klasse weitergeben (vererben). Diese erhält dann eine *Kopie* dieser Eigenschaften.

Die vererbende Klasse heißt Oberklasse der erbenden Klasse, die erbende Unterklasse der vererbenden Klasse.

Es gibt Einfachvererbung und Mehrfachvererbung.

**Video-Gaming-Syndrom.** Risiko bei der Anwendung von Lernmodellen. Der Aspekt des Erzielens guter Ergebnisse wird zum alleinigen Spielzweck. Der Spieler spielt mit dem Modell herum, ohne die Ergebnisse seines Tuns einer kritischen Reflexion zu unterziehen. Stattdessen wird durch Herumprobieren eine Vorgehensweise identifiziert, mit der sich in diesem Spiel optimale Ergebnisse erzielen lassen.

**Wörterbuch.** Wird vom Dolmetscher verwendet, um zwischen einem SESAM-Modell und dem Spieler vermitteln zu können. Spezifiziert, welche Kommandos das Modell versteht und welche Nachrichten es versenden kann. Zu jedem Kommando sind die möglichen Eingaben des Benutzers, die dieses Kommando auslösen sollen, genannt. Für jede Nachricht des Modells wird der Text angegeben, der beim Erhalt einer Nachricht dieses Typs ausgegeben werden soll.

**Zeitverbrauch.** (BASE-2) Gibt an, wieviel Modellzeit (in Minuten) die Ausführung einer Instanz einer Regel oder eines Benutzerkommandos den Projektleiter (den Spieler) kostet.

## B Beispielprogramm

Dieses Beispielprogramm faßt die Beispiele aus Kapitel 4 zusammen und stellt sie in einen Zusammenhang. Modelliert wird hier ein Software-Projekt, das einen Projektleiter, Peter Leiter, und einen angestellten Entwickler, John Bankmiller, hat. Der Projektleiter hat nur die Möglichkeit, Bewerbungsgespräche zu führen (Benutzerkommando Interview), Entwickler einzustellen (Benutzerkommando Hire) und Entwickler an der Spezifikation arbeiten zu lassen (Benutzerkommando LetSpecify). Die Regel EnhanceSpecification aus Abschnitt 4.3.1 tritt hier wieder auf, zusammen mit einer Regel DecreaseBudget, die für einen angestellten Entwickler dessen Tageskosten vom Projektbudget abzieht.

Kommentare werden in BASE-2, wie in Ada 95, durch zwei Bindestriche eingeleitet und gehen bis zum Zeilenende.

```

-----
-- BASE-2 Beispielprogramm --
-----

model Example is

-----
-- Definitionsteil --
-----

-----
-- Definition der Entitätstypen --
-----

entity type Project is
attributes
  name: String;
  customer: String;
  budget: Real;
end type;

entity type Person is
attributes
  name: String;
  age: Integer;
end type;

entity type Manager conforming Person is
attributes
  name: String;
  age: Integer;
end type;

entity type Developer conforming Person is
attributes
  name: String;
  age: Integer;
  experience: Real;
  motivation: Real;
  cost_per_day: Real;
end type;

```

```
entity type Document is
attributes
  name: String;
  version: Real;
end type;

entity type Specification conforming Document is
attributes
  name: String;
  version: Real;
  function_points: Real;
  max_function_points: Real;
end type;

-----
-- Definition der Relationstypen --
-----

relation type WorksOn is
roles
  who: Person;
  what: Document;
attributes
  intensity: Real;
end type;

relation type Specifies conforming WorksOn is
roles
  who: Developer;
  what: Specification;
attributes
  intensity: Real;
end type;

relation type IsAssociatedWith is
roles
  who: Person;
  what: Project;
end type;

relation type IsMemberOf conforming IsAssociatedWith is
roles
  who: Developer;
  what: Project;
end type;

relation type Manages conforming IsAssociatedWith is
roles
  who: Manager;
  what: Project;
end type;

relation type TalksTo is
roles
  who: Person;
  whom: Person;
end type;
```

```

relation type Interviews conforming TalksTo is
roles
  who: Manager;
  whom: Developer;
end type;

-----
-- Definition der Funktionen --
-----

function min(a: Real, b: Real) return Real is
  -- returns the minimum of a and b
begin
  if a > b then
    return b;
  else
    return a;
  end if;
end function;

-----
-- Definition der Regeln --
-----

rule EnhanceSpecification[1000] taking 0 is
structure
  s: Specification;
  d: Developer;
  w: Specifies(d,s);
constraints
  s.function_points < s.max_function_points;
declare
  new_function_points :=
    (s.max_function_points - s.function_points)*0.1
    * d.experience * d.motivation * w.intensity;
begin
  s.function_points :=
    min(s.function_points + new_function_points,
        s.max_function_points);
end rule;

rule DecreaseBudget[0] taking 0 is
structure
  p: Project;
  d: Developer;
  m: IsMemberOf(d,p);
begin
  p.budget := p.budget - d.cost_per_day/24;
  -- Faktor 1/24 zur Umrechnung von Tagen auf Stunden
end rule;

-----
-- Definition der Benutzerkommandos --
-----

command Interview(d: Developer) taking 90 is
structure
  p: Manager;
  j: Project;
  m: Manages(p,j);
constraints

```

```

    not exists(Interviews(p,d));
begin
    create relation Interviews with
        who := p;
        to := d;
    end create;
    send_Message( HaveInterview, d );
end command;

command Hire(whom: Developer) taking 60 is structure
    p: Manager;
    j: Project;
    t: Interviews(p,whom);
    m: Manages(p,j);
constraints
    not exists(IsMemberOf(whom,j));
begin
    create relation IsMemberOf with
        who := whom;
        what := j;
    end create;
    send_message( Hired, whom );
    delete(t);
end command;

command LetSpecify(whom: Developer) taking 125 is
structure
    s: Specification;
    p: Project;
    m: IsMemberOf(whom,p);
constraints
    not exists(Specifies(whom,s));
begin
    create relation Specifies with
        who := whom;
        what := s;
        intensity := 1;
    end create;
    send_message( Specifies, whom );
end command;

-----
-- Startsituationsteil --
-----

begin at 1995/11/20/08:00 timestep 60 using "demo"

    create entity DasProjekt: Project aka "ScheSch" with
        name := "ScheSch";
        customer := "Suebia";
        budget := 250000;
    end create;

    create entity Projektleiter: Manager aka "Peter Leiter" with
        name := "Peter Leiter";
        age := 39;
    end create;

```

```
create relation Manages with
  who := Projektleiter;
  what := DasProjekt;
end create;

create entity Andrea: Developer aka "Andrea Sevensleep" with
  name := "Andrea Sevensleep";
  age := 26;
  experience := 0.7;
  motivation := 0.95;
  cost_per_day := 400;
end create;

create entity John: Developer aka "John Bankmiller" with
  name := "John Bankmiller";
  age := 35;
  experience := 0.94;
  motivation := 0.5;
  cost_per_day := 600;
end create;

create relation IsMemberOf with
  who := John;
  what := DasProjekt;
end create;

create entity Spec: Specification aka "Die Spezifikation" with
  name := "Die Spezifikation";
  version := 1.0;
  function_points := 0;
  max_function_points := 234;
end create;

end model;
```

## C BASE-2-Grammatik

Dies ist die vollständige Grammatik von BASE-2 in Erweiterter Backus-Naur-Form (EBNF). Die Produktionen wurden alphabetisch geordnet, um sie leichter auffinden zu können.

Das Startsymbol der Grammatik ist das Symbol `model`.

```

action_part = [local_vars] "begin" statements "end" .
aka_clause = "aka" expression .
assignment = extended_ident "!=" expression .
attribute_decl = identifier ":" identifier .
attribute_list = "attributes" attribute_decl ";" {attribute_decl ";" } .
attributetype = basetype | listtype .
attributetype_definition = "type" identifier "is" attributetype .
base_literal = boolean_literal | integer_literal
                real_literal | string_literal | date_literal .
basetype = "boolean" | "integer" | "real" | "string" | "date" .
boolean_literal = "true" | "false" .
conforming_clause = "conforming" identifier {"," identifier } .
constraints_clause = "constraints" expression ";" {expression ";" } .
create_clause = entity_creation | relation_creation .
creations = {create_clause ";" } .
command_definition = "command" identifier [parameter_list]
                    time_consumption "is" [structure_clause]
                    [constraints_clause] action_part "command" .
date_literal = digit digit digit digit "/" digit digit "/" digit digit
                "/" digit digit ":" digit digit .
definition = attributetype_definition | entitytype_definition
                | relationtype_definition | rule_definition
                | command_definition | function_definition .
definitions = definition ";" {definition ";" } .
digit = "0" | "1" | ... | "9" .
entity_creation = "create" "entity" identifier ":" identifier
                aka_clause [with_clause] "end" "create" .
entitytype_definition = "entity" "type" identifier [conforming_clause]
                "is" [attribute_list] "end" "type" .
expression = "(" operator_expression ")" | function_call
                | operator_expression | simple_expression .
extended_ident = identifier [ "." identifier ] .
formal_component = identifier ":" identifier
                [ "(" identifier {"," identifier } ")" ] .
function_call = identifier [ "(" expression {"," expression } ")" ] .
function_definition = "function" identifier [parameter_list]
                "return" identifier "is" action_part "function" .
identifier = letter { [ "_" ] ( letter | digit ) } .

```

```

if_statement = "if" expression "then" statements
              ["else" statements] "end" "if" .
init = identifier "!=" expression .
integer_literal = ["-"] digit{digit} .
interpreter_file = "using" string_literal .
letter = "a" | "b" | ... | "z" .
list_literal = "(" [base_literal {"," base_literal}] ")" .
listtype = "list" "of" basetype .
local_vars = "declare" var_decl ";" {var_decl ";"} .
model = "model" identifier "is" definitions start_situation "model" ";" .
operator_expression = "-" expression | "not" expression
                    | expression ("+" | "-" | "*" | "/" | "&") expression
                    | expression ("<" | ">" | ">=" | "<=" | "=" | "/=") expression
                    | expression ("and" | "or") expression .
parameter = identifier ":" identifier .
parameter_list = "(" parameter {"," parameter} ")" .
priority = integer_literal .
procedure_call = identifier "(" expression {"," expression} ")" .
real_literal = ["-"] ( digit{digit}["."{digit}] | "."digit{digit} )
              ["E"["-"]digit{digit}] .
relation_creation = "create" "relation" identifier
                  with_clause "end" "create" .
relationtype_definition = "relation" "type" identifier
                        [conforming_clause] "is" role_list [attribute_list]
                        "end" "type" .
return_statement = "return" expression .
role_decl = identifier ":" identifier .
role_list = "roles" role_decl ";" role_decl ";" {role_decl ";"} .
rule_definition = "rule" identifier "[" priority "]"
                time_consumption "is" [structure_clause]
                [constraints_clause] action_part "rule" .
simple_expression = base_literal | list_literal | extended_ident .
start_date_declaration = "at" date_literal .
start_situation = "begin" start_date_declaration time_step
                 interpreter_file creations "end"
statement = assignment | create_clause | if_statement
          | while_statement | procedure_call | return_statement .
statements = statement ";" {statement ";"} .
string_literal = "" {character} "" .
structure_clause = "structure" formal_component ";"
                 {formal_component ";"} .
time_consumption = "taking" integer_literal .
time_step = "timestep" integer_literal .

```

```
var_decl = identifier ":" identifier "!=" expression .  
while_statement = "while" expression "loop" statements "end" "loop" .  
with_clause = "with" init ";" {init ";"}
```

## Literaturverzeichnis

- Abteilung Software Engineering (1995a): *Overview of SESAM*. Bestandteil der elektronisch verfügbaren Dokumentation des SESAM-1-Systems, Institut für Informatik, Universität Stuttgart. Erhältlich unter `ftp://ftp.informatik.uni-stuttgart.de/pub/sesam/sesam1.1.tar.gz`.
- Abteilung Software Engineering (1995b): *Operating Instructions for SESAM Players*. Bestandteil der elektronisch verfügbaren Dokumentation des SESAM-1-Systems, Institut für Informatik, Universität Stuttgart.
- Abteilung Software Engineering (1995c): *Instructions for SESAM Model Builders*. Bestandteil der elektronisch verfügbaren Dokumentation des SESAM-1-Systems, Institut für Informatik, Universität Stuttgart.
- Bossel, H. (1994): *Modeling and Simulation*. Vieweg, Wiesbaden.
- Brooks, F. P. Jr. (1982): *The mythical man-month*. In: Brooks, F. P. Jr.: *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, Massachusetts, S. 13-26.
- Deininger, M. (1995): *SESAM-Lite: Systembeschreibung*. Interner Bericht, Abteilung Software Engineering, Institut für Informatik, Universität Stuttgart.
- Duden Informatik*. 2. Auflage, Dudenverlag, Mannheim, 1993.
- Forrester, J. W. (1971): *Principles of Systems*. Wright-Allen Press, Cambridge, Massachusetts.
- Forrester, J. W. (1974): *Industrial Dynamics*. Cambridge, Massachusetts.
- Frühauf, K., J. Ludewig, H. Sandmayr (1988): *Software-Projektmanagement und -Qualitätssicherung*. Teubner, Stuttgart.
- Ghezzi, C., M. Jazayeri (1989): *Konzepte der Programmiersprachen: Begriffliche Grundlagen, Analyse und Bewertung*. Oldenbourg, München.
- Goldberg, A., D. Robson (1990): *SMALLTALK-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachusetts.
- Göttler, H. (1988): *Graphgrammatiken in der Softwaretechnik: Theorie und Anwendungen*. Informatik-Fachberichte Nr. 178, Springer, Berlin.
- Hoare, C. A. R. (1973): *Hints on Programming Language Design*. Programmatische Rede, gehalten bei der ACM SIGACT/SIGPLAN Conference on Principles of Programming Languages, Boston, Oktober 1973.
- Intermetrics (1995): *Ada 95 Reference Manual: The Language, The Standard Libraries*. Intermetrics, Cambridge, Massachusetts.
- Kemeny, J. G., T. E. Kurtz (1985): *Back to Basic*. Addison-Wesley, Reading, Massachusetts.

- Kiesler, S., L. Sproull (1982): *Managerial response to a changing environment: Perspectives on problem sensing from social cognition*. In: Administrative Science Quarterly, Band 27, S. 548-570.
- Kim, D. H. (1993): *The link between individual and organizational learning*. In: Sloan Management Review, 35, 1, S. 37-50.
- Krickhahn, R., B. Radig (1987): *Die Wissensrepräsentationssprache OPS5*. Vieweg, Wiesbaden.
- Langfield-Smith, K. (1992): *Exploring the need for a shared cognitive map*. In: Journal of Management Studies, 29, 3, S. 349-367.
- Ludewig, J. (1994): *SESAM: Grundidee und Überblick*. In: J. Ludewig (Hrsg.): SESAM: Software-Engineering Simulation durch animierte Modelle. Bericht 5/1994, Abteilung Software Engineering, Institut für Informatik, Universität Stuttgart.
- Meyer, B. (1992): *Eiffel: The Language*. Prentice Hall, New York.
- Reißing, R. (1996): *Sprachbeschreibung für BASE-2*. Interner Bericht, Abteilung Software Engineering, Institut für Informatik, Universität Stuttgart.
- Schneider, B. (1996a): *Modellbeschreibungssprache für SESAM-2*. Interner Bericht, Abteilung Software Engineering, Institut für Informatik, Universität Stuttgart.
- Schneider, B. (1996b): *Konzeption und Realisierung einer Modellbeschreibungssprache für SESAM-2*. Diplomarbeit (in Entstehung), Institut für Informatik, Universität Stuttgart.
- Schneider, K. (1994): *Ausführbare Modelle der Software-Entwicklung: Struktur und Realisierung eines Simulationssystems*. vdf Hochschulverlag, Zürich.
- Schürr, A., B. Westfechtel (1992): *Graphgrammatiken und Graphersetzungssysteme*. Aachener Informatik-Berichte Nr. 92-15, RWTH Aachen.
- Senge, P. M. (1990): *The Fifth Discipline: The Art and Practice of the Learning Organization*. Doubleday Currency, New York.
- Senge, P. M., J. D. Sterman (1992): *Systems thinking and organizational learning: Acting locally and thinking globally in the organization of the future*. In: European Journal of Operational Research, Band 59, S. 137-150.
- Spiegel, A. (1995): *Konstruktion eines Dolmetschers*. Diplomarbeit 1304, Institut für Informatik, Universität Stuttgart.
- Sterman, J. D. (1994): *Learning in and about Complex Systems*. In: System Dynamics Review, 10, 2-3, S. 291-330.
- Strata, R. (1989): *Organizational learning: The key to management innovation*. In: Sloan Management Review, 30, 3, S. 63-74.
- Stroustrup, B. (1992): *Die C++-Programmiersprache*. 2. Auflage, Addison Wesley, Bonn.
- Wirth, N. (1988): *Programming in Modula-2*. 4. Auflage, Springer, New York.

### **Erklärung**

Ich erkläre hiermit, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Insbesondere versichere ich, daß ich alle wörtlichen und sinn-  
gemäßen Übernahmen aus anderen Werken nach bestem  
Wissen und Gewissen als solche kenntlich gemacht habe.

Stuttgart, den 22.04.1996

\_\_\_\_\_