Forschungszentrum Karlsruhe
Technik und Umwelt
Wissenschaftliche Berichte
FZKA 5974

Ein Konzept zur flexiblen Unternehmensmodellierung und Software-Entwicklung

L. Bogdanowicz
Institut für Angewandte Informatik

Dezember 1997

Forschungszentrum Karlsruhe
Technik und Umwelt
Wissenschaftliche Berichte
FZKA 5974

Ein Konzept zur flexiblen Unternehmensmodellierung und Software-Entwicklung

Leszek Bogdanowicz

Institut für Angewandte Informatik

Von der Fakultät für Wirtschaftswissenschaften der Universität Fridericiana zu Karlsruhe genehmigte Dissertation

Forschungszentrum Karlsruhe GmbH, Karlsruhe
1997

Als Manuskript gedruckt

Forschungszentrum Karlsruhe GmbH
Postfach 3640, 76021 Karlsruhe

Mitglied der Hermann von Helmholtz-Gemeinschaft
Deutscher Forschungszentren (HGF)

ISSN 0947-8620

Ein Konzept zur flexiblen Unternehmensmodellierung und Software-Entwicklung

Zusammenfassung

Die Integration eines Unternehmens wird durch den Einsatz von Unternehmensmodellen und einer Integrationsplattform unterstützt. Mit Hilfe von Unternehmensmodellen können die Funktionen, Daten, Organisation und Betriebsmittel des Unternehmens beschrieben werden, um damit eine reibungslose Kooperation zwischen einzelnen CIM-Komponenten (z.B. PPS, CAD/CAM) zu gewährleisten. Die Integrationsplattform definiert die Schnittstellen, Datenformate, Kommunikationsprotokolle und Dienste zur Unterstützung des Informationsaustausches zwischen diesen Komponenten.

In Entwicklung befindliche Konzepte zur Unternehmensintegration werden oft schon während des Entwicklungsprozesses validiert (wie z.B. CIMOSA in dem ESPRIT-Projekt VOICE). Da die endgültige und vollständige Definition des Konzeptes nicht vorliegt, muß die Validierung durch Prototypen erfolgen, mit denen nur Teile des Konzeptes schrittweise validiert werden.

Prototypische Unternehmensmodelle dienen dabei zur Validierung der Modellierungsmethode, die in dem Konzept definiert wird. Prototypische Softwarekomponenten dienen zur Validierung der Spezifikation der Integrationsplattform, die aus diesen Komponenten besteht. Bestimmte Änderungen (bzw. Erweiterungen) des Konzeptes können zu folgenden Situationen führen:

- die Modellierungsmethode (als Bestandteil des Konzeptes) kann sich ändern;
- für die Validierung bestimmter Teile der Spezifikation der Softwarekomponenten wurde ein Prototyp mit einer Softwareentwicklungsmethode realisiert. Für die Validierung weiterer Teile der Spezifikation wird die nächste Version des Prototyps realisiert. Es kann vorkommen, daß dies nicht mehr mit der bis jetzt eingesetzten Softwareentwicklungsmethode zu realisieren ist, da diese Methode nicht ausreichend "mächtig" ist.

Dies bedeutet, daß die Entwicklung der Prototypen (Unternehmensmodelle oder Softwarekomponenten) nach diesen Konzeptänderungen mit einer neuen Version der bis jetzt eingesetzten Methode (Modellierungs- oder Softwareentwicklungsmethode) fortgeführt werden muß. Die Erfahrungen, die bei der Validierung des CIMOSA-Konzeptes in dem ESPRIT Projekt VOICE erworben wurden, zeigen, daß dies mit verfügbaren Werkzeugen nicht möglich bzw. nur sehr schwierig zu realisieren ist. Es existieren keine Ansätze (also auch keine Werkzeuge), die die Änderungen einer Methode während der mit dieser Methode durchgeführten Entwicklung unterstützen.

In dieser Arbeit wurde ein Ansatz ausgearbeitet, der die Entwicklung von Prototypen zur Validierung von noch nicht vollständig definierten Unternehmensintegrations-Konzepten ermöglicht, indem die Änderungen der zur Entwicklung eingesetzten Methode während des Entwicklungsprozesses unterstützt werden. Eine Methode wird dabei als eine planmäßige Vorgehensweise zusammen mit der eingesetzten Beschreibungstechnik verstanden.

Concept of flexible enterprise modelling and software development

Abstract

Enterprise integration is supported by use of enterprise models and an integration platform. With a help of enterprise models functions, data, organisation and enterprise resources can be described and at the same time good cooperation between various CIM components (e.g. PPS, CAD/CAM) ensured. On the other hand, integration platform defines interfaces, data formats, communication protocols and services in order to support information exchange between these components.

Currently developed conceptions for enterprise integration are often validated already during the development process (like in case of CIMOSA in ESPRIT project VOICE). Because the final definition of the concept is not available, validation must be conducted with a help of prototypes which allow only a gradual validation of parts of the conception.

Prototype enterprise models are used also to validate a modelling method defined in this concept. Prototype software components are used to validate the specifications of the integration platform, which consists of these elements. Certain changes (or extensions) of the conception may lead to the following situations:

- the modelling method may be changed (as a part of the conception)
- to validate given part of the specification of the software components a prototype was elaborated with a help of a software development method. To validate other parts of the specification another version of the prototype is developed. It may happen that it will not be possible to achieve with a help of the method (of software development) used so far because this method is not „strong" enough.

It means that the development of prototypes (enterprises models or software components) must be continued after the changes of the concept with a help of a new version of the so far applied method (modelling method or software development method). Experience gained during validation of the CIMOSA concept in ESPRIT project VOICE indicates that this continuation is not possible with a help of available tools or it is very difficult to accomplish. There are no approaches (or tools) which would support changes of the method during development realised with a help of this method.

In this work an approach has been elaborated which allows development of prototypes for validation of not fully defined conceptions of the enterprise integration by supporting changes of the method during the process of development in which this method is being applied. Such method is defined here as the organised approach together with the used description technique.

Inhaltsverzeichnis

Abkürzungen

γ	Generierungsfunktion der dynamischen Baustein-Templates
$\mathbf{a_I}$	Arc-Instanz der Methode m
$\mathbf{A_I}$	Menge der Arc-Instanzen der Methode m
$\mathbf{A_{O,s}}$	Menge der Kanten des System-Modells s_m (der Arc-Occurrences in dem System-Modell s_m)
$\mathbf{a_{O,s}}$	eine Kante des System-Modells s_m (eine Arc-Occurrence in dem System-Modell s_m)
AWF	Ausschuß für Wirtschaftliche Fertigung e.V.
BDE	Betriebsdatenerfassung
CAD	Computer Aided Design
CAM	Computer Aided Manufacturing
CAP	Computer Aided Planning
CAQ	Computer Aided Quality Assurance
CASE	Computer Aided Software Engineering
CI	Control Input
$\mathbf{c_I}$	Component-Instanz der Methode m
CIM	Computer Integrated Manufacturing
CIMOSA	CIM Open System Architecture
CNMA	Communications Network for Manufacturing Application
CO	Control Output
$\mathbf{c_{O,s}}$	eine Komponente des System-Modells s_m (eine Component-Occurrence in dem System-Modell s_m)
CP-Netze	Coloured-Petri-Netze
DP	Domain Process
EA	Enterprise Activity
ER	Entity Relationship
ESPRIT	European Strategic Program for Research and Development in Information Technology
FhG-ISI	Frauenhofer-Gesellschaft, Institut für Systemtechnik und Innovationsforschung
FI	Function Input
FO	Functional Operation
FO	Function Output
IE	Information Element

IIS	Integrating Infrastructure
I	Menge der Baustein-Instanzen der Methode m
i	Baustein-Instanz der Methode m
ISDN	Intergrated Services Digital Network
LAN	Local Area Network
M	Mächtigkeit der Methode
m	Methode
MAP	Manufacturing Automation Protocol
$\mathbf{n_I}$	Node-Instanz der Methode m
$\mathbf{N_I}$	Menge der Node-Instanzen der Methode m
$\mathbf{n_{O,s}}$	ein Knoten des System-Modells s_m (eine Node-Occurrence in dem System-Modell s_m)
$\mathbf{N_{O,s}}$	Menge der Knoten des System-Modells s_m (der Node-Occurrences in dem System-Modell s_m)
NR/T-Netze	Nested-Relation/Transition-Netze
$\mathbf{O_s}$	Menge der Baustein-Occurrences in dem System-Modell s_m
OO	Object Oriented
OOA	Object Oriented Analysis
OOD	Object Oriented Design
OSA	Open System Architecture
OV	Object View
P/T-Netze	Place/Transition-Netze
PPS	Produktionsplanung und -steuerung
Pr/T-Netze	Prädikat/Transitionsnetzen
RI	Resource Input
RO	Resource Output
RT-Methode	Real-Time-Analysis-Methode
SA	Strukturierte Analyse
SADT	Structured Analysis and Design Technique
SET	Standard d'Exchange et de Transfer
$\mathbf{s_m}$	Ein mit Hilfe der Methode m entwickeltes System-Modell
STEP	Stadard for the Exchange of Product Model Data
$\mathbf{T_d}$	Menge der dynamischen Baustein-Templates, die bei der Definition der Methode m generiert werden
$\mathbf{t_{d,A}}$	dynamisches Arc-Template der Methode m

$T_{d,A}$	Menge der dynamischen Baustein-Templates (Arc-Templates), die für die Arc-Instanzen der Methode m generiert werden
$t_{d,C}$	dynamisches Component-Template der Methode m
$t_{d,N}$	dynamisches Node-Template der Methode m
$T_{d,N}$	Menge der dynamischen Baustein-Templates (Node-Templates), die für die Node-Instanzen der Methode m generiert werden
TOP	Technical Office Protocol
$t_{s,A}$	statisches Arc-Template
$t_{s,C}$	statisches Component-Template
$t_{s,N}$	statisches Node-Template
V	Vollständigkeit des Systems
VDA	Verband der deutschen Automobilindustrie
VDAFS	VDA-Flächenschnittstelle
VOICE	Validating OSA in Industrial CIM Environments

1. Einführung

Die Informationstechnologie gewinnt in heutigen Unternehmen immer größere Bedeutung. Der permanente Zwang zur Minimierung der Kosten und Durchlaufzeiten sowie zur Maximierung der Qualität führt zu zunehmendem Einsatz von computerunterstützten Anwendungen. Diese Anwendungen werden in verschiedenen Unternehmensbereichen eingesetzt, die von der Entwicklung und Konstruktion (CAD) über die Arbeitsplanung (CAP), Produktionsplanung und -steuerung (PPS), Teilefertigung, Montage und Transport (CAM) bis zur Qualitätssicherung (CAQ) reichen. Sie werden immer öfters in einem unternehmensweiten bereichsübergreifenden System integriert [AWF85]. In solchen Systemen, die mit dem Begriff CIM (Computer Integrated Manufacturing) bezeichnet werden, spielt die Integration der Funktionen und Daten eine wesentliche Rolle.

Diese Integration wird einerseits durch Unternehmensmodelle unterstützt, die sowohl die Funktionen und Daten als auch die Organisation und verfügbare Betriebsmittel des Unternehmens beschreiben können und die für eine reibungslose Kooperation zwischen einzelnen computerunterstützten Anwendungen (die als CIM-Komponenten bezeichnet werden) sorgen. Auf der anderen Seite wird eine Integrationsplattform eingesetzt, die den problemlosen Austausch der Information zwischen diesen Komponenten durch Definition von entsprechenden Schnittstellen, Datenformaten und Kommunikationsprotokollen gewährleistet (Abb. 1).

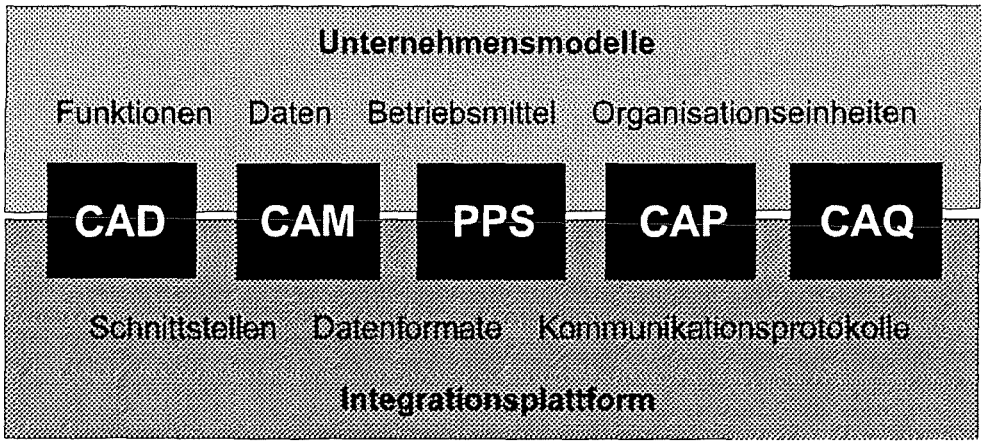


Abb. 1. Integration der CIM-Komponenten

In beiden in der Abbildung gezeigten Bereichen existieren bereits verschiedene Ansätze. In den Modellierungskonzepten wird ein Unternehmen in verschiedenen Sichten analysiert und beschrieben. Die zwei wichtigsten Aspekte: Funktionen und Daten werden beispielsweise in dem Information-Engineering-Konzept betrachtet [Mar89]. Andere Konzepte befassen sich zusätzlich

mit organisatorischen Aspekten (INCOME) [SNS89] und Steuerungsaspekten (ARIS) [Sch92]. Auch die Beschreibung von Betriebsmitteln kann Bestandteil eines Modellierungskonzeptes sein (CIMOSA) [AMICE93].

Die wichtigsten Werkzeuge, die eine Integrationsplattform bilden, sind Netzwerke, Datenschnittstellen und Datenbanken. Die Netzwerke schaffen durch eine physikalische Verbindung zwischen CIM-Komponenten eine Voraussetzung für den Informationsaustausch innerhalb eines integrierten Systems. Die Datenschnittstellen sorgen dafür, daß diese Information für verschiedene Komponenten lesbar ist, indem sie die Form der ausgetauschten Information festlegen. Die Datenbanken ermöglichen die Integration und Verwaltung von Daten, die zwischen den CIM-Komponenten ausgetauscht wird.

Die Integrationswerkzeuge können verwendet werden, um mehrere Schnittstellen zwischen mehreren CIM-Komponenten zu schaffen. Da die Zahl der notwendigen Schnittstellen sehr schnell mit der Zahl der verbundenen Komponenten wächst (bei n Komponenten sind das ½ n (n-1) Schnittstellen), werden neue Lösungen, wie z.B. in dem CIMOSA-Konzept [AMICE93], vorgeschlagen, die den Einsatz einer gemeinsamen Infrastruktur mit bestimmten, für alle CIM-Komponenten verfügbaren Diensten (implementiert in Form von Softwarekomponenten) vorsehen. Die Integration einer neuen Komponente bedingt in diesem Konzept die Definition von nur einer Schnittstelle zu der Infrastruktur, was die Zahl der notwendigen Schnittstellen im gesamten CIM-System deutlich verringert (bei n Komponenten sind das nur n Schnittstellen).

In Entwicklung befindliche Integrationskonzepte (wie z.B. CIMOSA) werden oft schon während des Entwicklungsprozesses validiert. Da die endgültige und vollständige Definition des Konzeptes nicht vorliegt, muß die Validierung durch die Entwicklung von Prototypen erfolgen, mit denen die existierenden Teile des Konzeptes validiert werden können. Die Erfahrungen, die bei der Validierung des CIMOSA-Konzeptes erworben worden sind, zeigen, daß die verfügbaren Methoden und Werkzeuge nicht alle Anforderungen erfüllen, die während der Validierung von nicht vollständig definierten Systemen anfallen.

Im Rahmen dieser Arbeit wird ein Konzept entwickelt, das den Einsatz von formalen Beschreibungsmethoden für die Entwicklung von Prototypen von Unternehmensmodellen (Modellentwicklung bzw. Modellierung) und bei der Entwicklung von Prototypen der Softwarekomponenten einer Integrationsplattform (Softwareentwicklung) ermöglicht. Die formalen Beschreibungsmethoden brauchen dabei nicht vollständig bzw. endgültig definiert zu sein, d.h. die Definition einer Methode kann während der Prototyp-Entwicklung geändert werden.

Es wird ein Werkzeug vorgestellt, das dieses Konzept unterstützt. Dies wird anhand von Beispiel-Entwicklungen demonstriert. Die ausführliche Definition des in der Arbeit behandelten Problems findet im Kapitel 3 statt. Im nächsten Kapitel erfolgt zunächst, neben der Erklärung der in der Arbeit verwendeten Begriffe, eine Präsentation der mit dem hier präsentierten Konzept verbundenen Problematik.

2. Stand der Technik

In den meisten Ansätzen haben die Entwicklung von Unternehmensmodellen und die Entwicklung von Softwarekomponenten der Integrationsplattform viele Gemeinsamkeiten. Der Entwicklungsprozeß wird in beiden Fällen in einzelne Phasen eingeteilt, von denen vor allem die drei Phasen Spezifikation, Entwurf und Implementation von größter Bedeutung sind. Der Verlauf des Entwicklungprozesses wird mit Hilfe von sogenannten Lebenszyklen beschrieben, die in den folgenden Kapiteln kurz erläutert werden.

In den einzelnen Entwicklungsphasen können verschiedene Basismethoden eingesetzt werden, wobei die selbe Methode in verschiedenen Phasen verwendet werden kann. Die Basismethoden werden heutzutage oft zu einer kombinierten Methode zusammengefaßt, die sowohl bei der Entwicklung von Unternehmensmodellen als auch bei der Entwicklung von Softwarekomponenten eingesetzt werden kann. Die Basismethoden und die kombinierten Methoden werden im weiteren näher beschrieben. Unter dem Begriff **Methode** wird "eine planmäßige, angewandte, begründete Vorgehensweise zur Erreichung von festgelegten Zielen" [Sch91] zusammen mit der dabei eingesetzten Beschreibungstechnik bzw. -sprache verstanden. In vielen Fällen wird diese Definition so reduziert, daß eine Methode mit einer Beschreibungstechnik bzw. -sprache gleichgesetzt wird.

Wegen der Parallele werden beide oben genannte Entwicklungsprozesse mit dem Begriff Systementwicklung bezeichnet. Unter dem Begriff **System** wird "eine abgegrenzte Anordnung von aufeinander einwirkenden Gebilden" verstanden [Sch91]. Im Fall der Unternehmensmodellierung sind die Gebilde entsprechende Objekte, die Funktionen, Daten, Betriebsmittel und organisatorische Einheiten eines Unternehmens oder auch weitere Untermodelle repräsentieren. Bei der Entwicklung von Softwarekomponenten können die Gebilde als Datenstrukturen, Funktionen, Prozesse und Module einer Anwendung interpretiert werden.

Die vorliegende Arbeit konzentriert sich auf die Entwicklung von Prototypen der oben genannten Systeme. Ein **Prototyp** eines Systems wird als ein Modell verstanden, das alle wesentliche Eigenschaften bzw. Leistungen des Systems (Unternehmensmodell oder Softwarekomponente) aufweist. Diese Eigenschaften bzw. Leistungen werden sukzessiv in einem Prototyp-Entwicklungsprozeß erreicht [Sch91]. Ein Prototyp wird dabei zur Validierung der Spezifikation

des Systems oder zur Validierung[1] der Methode, mit der dieses System entwickelt wird, eingesetzt.

Die im Rahmen dieser Arbeit entwickelte Methode unterstützt die Entwicklung von Prototypen der oben genannten Systeme, also sowohl der Unternehmensmodelle als auch der Softwarekomponenten, die für die Integration von verschiedenen CIM-Komponenten vorgesehen werden.

Am Ende dieses Kapitels werden die einzelnen CIM-Komponenten und die Definition eines CIM-Systems vorgestellt. Es werden dabei Probleme angesprochen, die bei der Entwicklung und Validierung von nicht vollständig definierten Systemen auftreten.

[1] Durch die Validierung wird die Überprüfung der Spezifikation eines Systems bzw. der Definition einer Methode auf Eignung für den vorgesehenen Verwendungszweck verstanden

2.1 Lebenszyklen eines Systems

Bei der Entwicklung von Systemen werden verschiedene Aufgaben realisiert: Identifizierung des Problems, Definition der Lösung, Implementation, Validierung und Wartung des Systems. Diese Aufgaben werden in mehreren Phasen der Entwicklung realisiert. Ein **Lebenszyklus** (bzw. ein Lebenszyklus-Modell) des Systems beschreibt dabei, welche Phasen und in welcher Reihenfolge das System in dem Entwicklungsprozeß durchläuft.

Es werden verschiedene Lebenszyklen unterschieden [Agr86]. Die meistdefinierten Phasen (bzw. Aktivitäten, die in dieser Phasen durchgeführt werden) dieser Lebenszyklen sind: Analyse und Definition der Anforderungen (Spezifikation), Entwurf und Implementation. Die einzelne Lebenszyklen unterscheiden sich vor allem durch die unterschiedliche verfeinerte Definition und Anordnung von diesen drei Phasen [Fle94]. Die folgende Definitionen der Aktivitäten und Beschreibungen der verschiedenen Lebenszyklen konzentrieren sich deswegen auf diesen Phasen.

In [Sch91] wird die erste Aktivität folgendermaßen definiert:

> **Spezifikation** ist eine detaillierte Beschreibung der Teile eines Ganzen und ihrer Eigenschaften bezüglich Größe, Qualität, Performance usw. sowie ihre Beziehungen untereinander ... Im wesentlichen geht es darum festzulegen
>
> - was ein System tun sollte, d.h. seine Funktionen;
> - worauf das System wirken sollte, d.h. seine Daten;
> - unter welchen Bedingungen das System arbeiten sollte, d.h. seine physische Umgebung und
> - welchen Einschränkungen das System unterliegen sollte, d.h. seine Grenzen.

Die Spezifikation, die oft als Definition der Anforderungen genannt wird, dient als Ausgangspunkt zum **Entwurf**. Im Entwurfsprozeß werden die Architektur des Systems, seine Komponenten und Module, sowie die in dem System verwendeten Schnittstellen und Daten definiert. Diese Definition muß dabei die in der Spezifikation beschriebenen Anforderungen erfüllen [ThD90]. Anders als bei der Spezifikation wird beim Entwurf nicht definiert, *was* das System tun soll, sondern *wie*.

Durch die **Implementation** wird der detaillierte Entwurf in den ausführbaren Code konvertiert. Zusätzlich umfaßt die Implementierung auch die Installation und das Testen des Systems [ThD90].

2.1.1 Konventioneller Lebenszyklus

In dem klassischen Lebenszyklus-Modell [Roy70], das als Wasserfall-Model bezeichnet wird, werden die Aktivitäten des Entwicklungsprozesses in einer Sequenz angeordnet. In der Spezifikationsphase werden die Anforderungen des Benutzers an das Zielsystem analysiert und definiert. Die Resultate dieser Phase dienen als Input für den Entwurf, bei dem die innere Struktur des Systems definiert wird. Das geschieht normalerweise durch die systematische Aufteilung auf einzelne, hierarchisch aufgebaute Module. Dazu werden zusätzlich die Schnittstellen und die Funktionalität jedes Moduls beschrieben. Zusätzlich können beim Entwurf auch die notwendigen Ressourcen, Anforderungen und die Leistungsfähigkeit, formale Definition des Verhaltens der Module betrachtet werden. In der Implementierung wird die entworfene Struktur des Systems in einer konkreten Programmiersprache implementiert, und es werden alle notwendigen Tests durchgeführt.

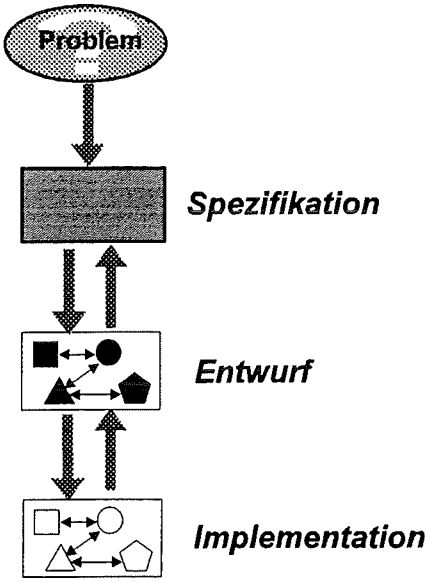


Abb. 2. Konventioneller Lebenszyklus.

Ursprünglich war keine Rückkopplung zwischen den einzelnen Phasen vorgesehen. Die in der Wartungsphase aufgefallenen notwendigen Änderungen und Erweiterungen des Systems waren durch die Rückkopplung der Implementation- mit der Spezifikation-Phase möglich. Dies ist oft unrealistisch, da in der Praxis viele kleinere Rückschritte zu den vorausgehenden Phasen unvermeidbar sind. Viele Erweiterungen des klassischen Modells [Boe76], [Agr86a] lassen deswegen die Rückkopplung zwischen einzelnen Phasen zu (Abb. 2) .

2.1.2 Inkrementeller Lebenszyklus

Der inkrementelle Lebenszyklus (incremental life cycle) [Fle94] unterscheidet sich von dem konventionellen Modell nur in der Implementationsphase. Ähnlich wie im "Wasserfall"-Modell wird eine komplette Spezifikation und ein vollständiger Entwurf durchgeführt. Die danach folgende Implementation ist aber nicht vollständig und dient dazu, möglichst schnell einen Prototyp zu entwickeln, und dadurch eine frühere Validierung zu ermöglichen (Abb. 3).

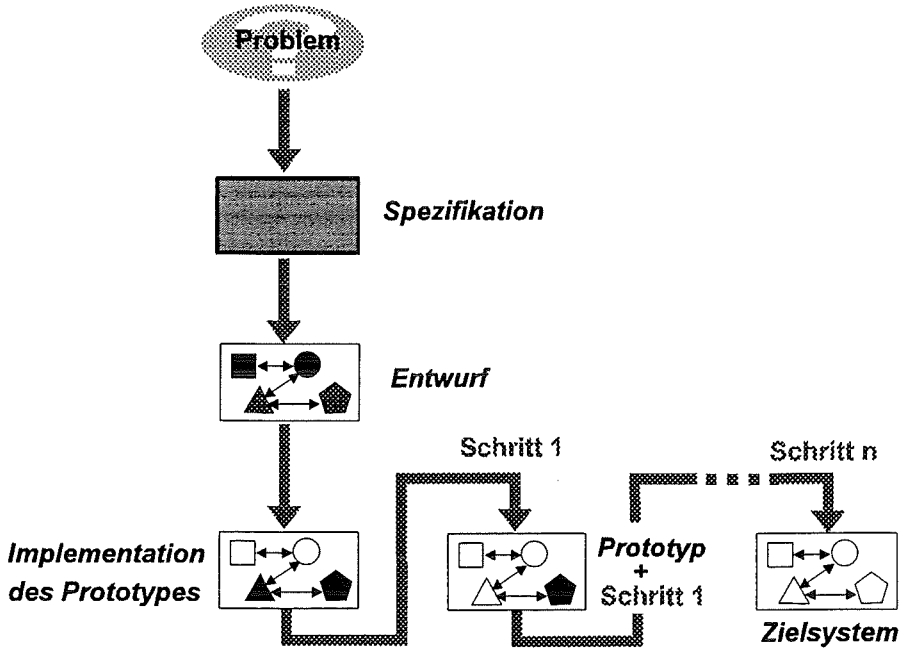


Abb. 3. Inkrementeller Lebenszyklus

Dieses Modell liegt zwischen dem konventionellen und dem evolutionären Modell. Ähnlich wie beim evolutionären Modell wächst das System im Laufe der Zeit, jedoch weniger dynamisch - die Änderungen und Erweiterungen können nicht so umfangreich sein, weil ein vollständiger, nicht änderbarer Entwurf als Basis für die Implementierung der Prototypen vorliegt.

2.1.3 Evolutionärer Lebenszyklus

Die wichtigste Aufgabe im evolutionären Lebenszyklus-Modell (evolutionary life cycle) ist die Erstellung eines Kernentwurfes, der ein Grundgerüst des Zielsystemes definiert [Bro87]. Diese Rahmendefinition ist genügend einfach, um eine schnelle Entwicklung der ersten Version des Systems durchzuführen und dadurch dem zukünftigen Benutzer schon in den ersten Entwicklungsphasen eine Vorstellung über das System, seine Grundstruktur und -funktionalität zu geben.

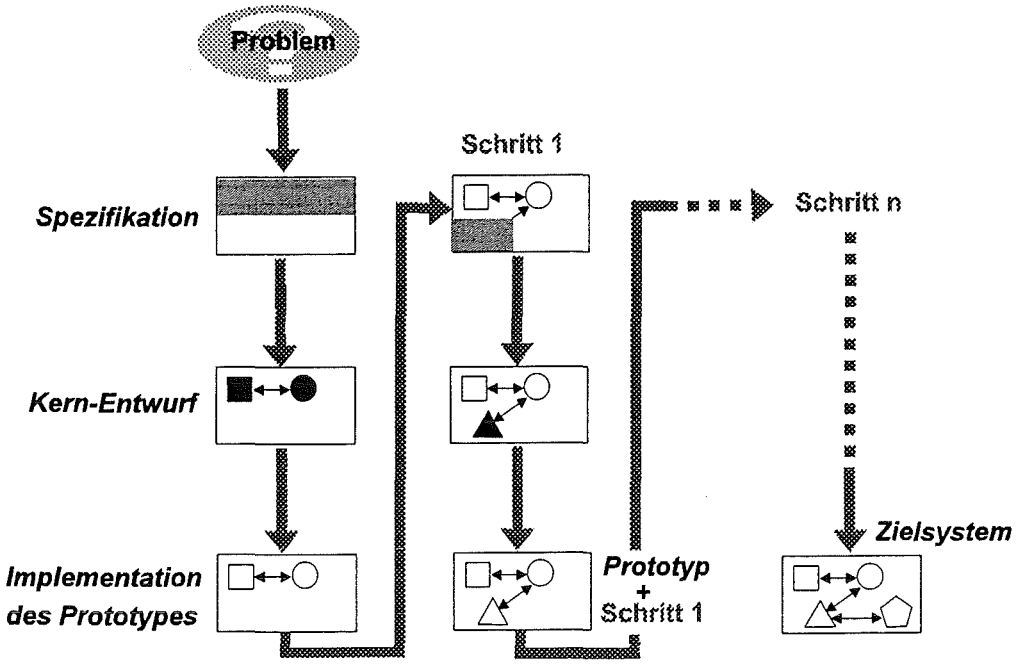


Abb. 4. Evolutionärer Lebenszyklus

Der Kernentwurf entsteht auf der Basis einer nicht vollständigen Spezifikation. Er wird bei der Implementation des ersten Prototyps verwendet (Abb. 4). Die mit dem Prototyp durchgeführte Validierung ermöglicht eine Erweiterung der vorhandenen Spezifikation. Die erweiterte Spezifikation führt dann in einen neuen Entwurf, und anschließend wird eine neue prototypische Version des Systems implementiert. Diese Schritte werden solange wiederholt, bis die gewünschte Funktionalität des Systems erreicht ist. Damit wird die endgültige Funktionalität nicht in einem Schritt erreicht - das Systems wächst (evoluiert) im Laufe der Zeit [Fle94].

2.1.4 Operationaler Lebenszyklus

In der Spezifikationsphase des operationales Lebenszyklus (operational life cycle) [Zav84] wird eine sogenannte operationalle Spezifikation erstellt, die implementationsunabhängig, jedoch aber gleichzeitig ausführbar ist. Die operationale Spezifikation beschreibt nur das Verhalten des Systems. Die Ausführung der Spezifikation erlaubt es, sowohl dem Entwickler als auch dem Benutzer schon in der Phase der Analyse, die gestellten Anforderungen an das System zu validieren [Agr86b].

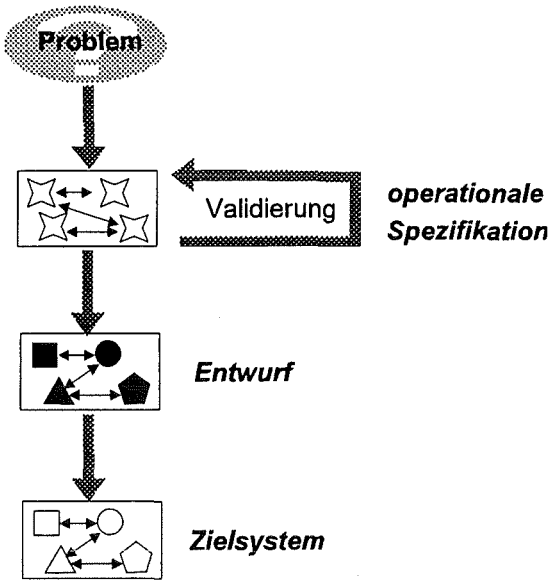


Abb. 5. Operationaler Lebenszyklus

Die Aufgabe der Entwurfsphase ist die Transformation der operationalen Spezifikation in die Entwurfsdefinition. Es wird dabei die problemorientierte Spezifikation in die implementationsorientierte Beschreibung des Systems umgewandelt. In der nachfolgenden Implementation wird diese Beschreibung in einer konkreten Programmiersprache implementiert.

2.1.5 Transformativer Lebenszyklus

Ein transformativer Lebenszyklus (transformational life cycle) setzt die Anwendung von Werkzeugen voraus, die zur automatischen Umwandlung der Spezifikation in die Implementation dienen. Die Spezifikation wird in diesem Modell betrachtet als [Agr86b]:

- ein Objekt, das im Hinblick auf die Benutzeranforderungen validiert wird
- ein Startpunkt für die Transformation in die Implementation
- ein Objekt, an dem die Wartung des Systems ausgeführt wird.

Die so definierte Spezifikation wird dann in mehreren Schritten in die Implementation in einer konkreten Programmiersprache und für eine konkrete Plattform umgesetzt (Abb. 6) - die Entwurfsphase kann daher in diesem Lebenszyklus nicht eindeutig abgegrenzt werden.

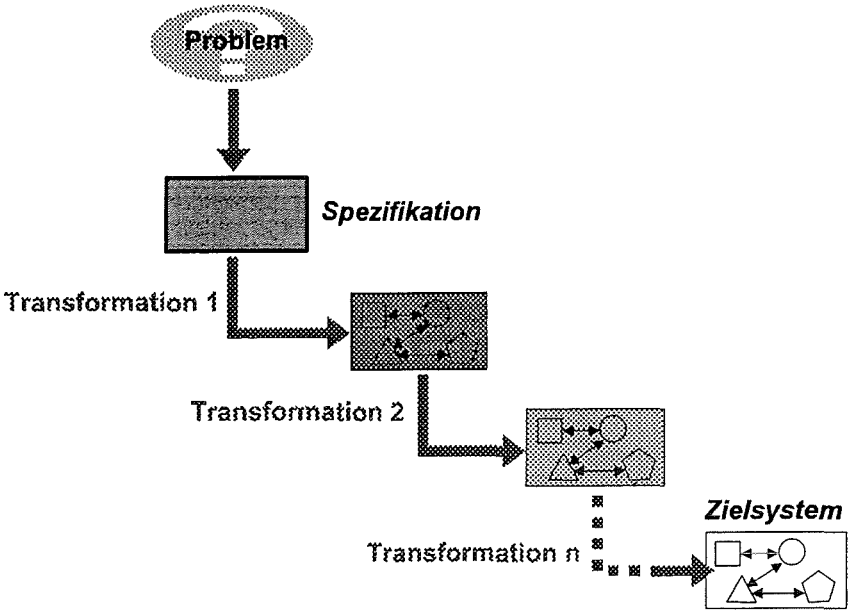


Abb. 6. Tansformationaler Lebenszyklus

2.1.6 Wissensbasierter Lebenszyklus

Das wissensbasierte Modell basiert auf einer Vorgehensweise, in der die Umsetzung der Spezifikation in die Implementation von einem wissensbasierten System unterstützt und dokumentiert wird (Abb. 7). Dieses System hat viele Aufgaben zu erfüllen, wie z.B. Wählen einer passenden Methode für die Problemlösung, Hilfestellung bei Entwurf und Implementation, Hilfestellung bei Debugging und Testen, Monitoring der Ausführung usw. [BCG83].

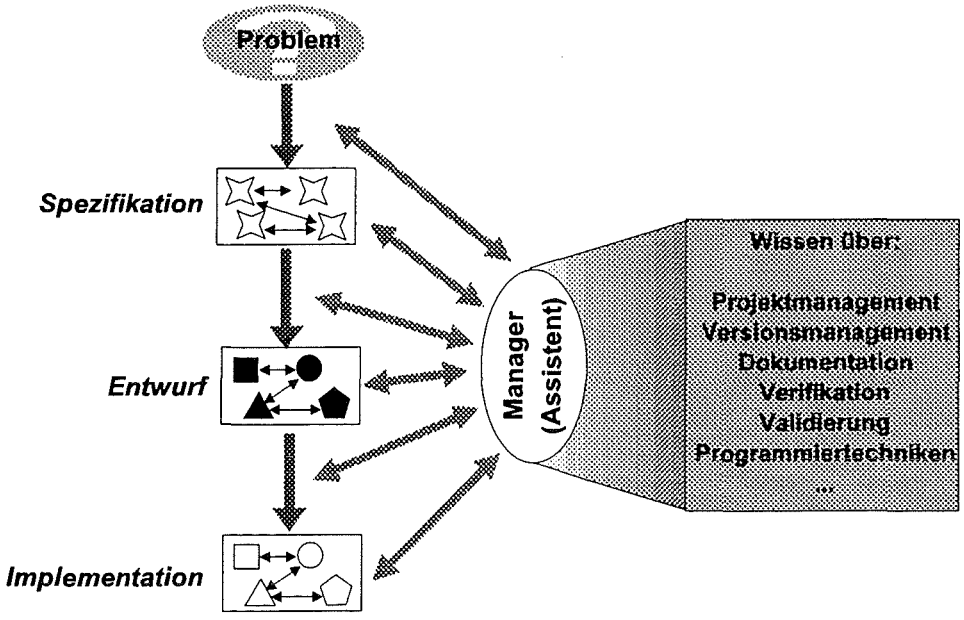


Abb. 7. Wissensbasierter Lebenszyklus

Dieser Lebenszyklus wird in verschiedenen Variationen eingesetzt, wie z.B. in [Asl91]. Die gemeinsame Eigenschaft dieser Ansätze ist die Anwendung eines (oder mehrerer) *Assistenten*, der alle Phasen der Entwicklung mit Hilfe des vorhandenen Wissens vielseitig unterstützt.

2.1.7 Zusammenfassung

Die hier vorgestellten Modelle sind nicht gleichermaßen geeignet für alle Typen von Systemen. Der konventionelle Lebenszyklus ist vor allem für von der Umgebung isolierte Systeme geeignet, bei denen die Interaktionen mit dem Benutzer bei der Entwicklung nicht betrachtet werden müssen. Für nicht isolierte Systeme sind andere Lebenszyklus (evolutionär, operational, transformational und wissensbasiert) besser geeignet. Der operationale Lebenszyklus hat dabei gegenüber dem evolutionären Lebenszyklus den Vorteil, daß die meisten Aktivitäten von Werkzeugen unterstützt werden können [Fle94]. Für die nicht vollständig definierten Systeme ist jedoch das evolutionäre Modell am besten geeignet, weil hier die Entwicklung der ersten Prototypen auf der Basis einer nicht vollständiger Spezifikation erfolgt. Die mit diesen Prototypen durchgeführte Validierung ermöglicht eine Erweiterung der vorhandenen Spezifikation des Systems.

Alle diese Modelle haben eine gemeinsame Eigenschaft: sie setzen den Einsatz einer vordefinierten Entwicklungsmethode voraus, um sich mit der für jede Entwicklungsphase spezifischen Problematik auseinanderzusetzen. Dies hat zur Folge, daß das System sich im Laufe der Entwicklung immer in einem eindimensionalem Raum bewegt. Die einzige Achse in dem Raum wird durch die einzelnen Entwicklungsphasen (Spezifikation, Entwurf, Implementation) bestimmt (Abb. 8). Die Bewegungen auf der Achse entsprechen den Änderungen des Entwicklungszustandes des Systems.

Bei der Entwicklung von Prototypen von Systemen, für die die entsprechende Anforderungen noch nicht vollständig formuliert sind, kann im Laufe des Entwicklungsprozesses festgestellt werden, daß die eingesetzte Entwicklungsmethode nicht mehr angemessen ist - die auf den neuen Anforderungen basierte Spezifikation und auch der entsprechende Entwurf kann möglicherweise nicht mehr mit der bis dahin verwendeten Methode realisiert werden. Es muß dann eine andere Methode eingesetzt bzw. eine vorhandene Methode erweitert werden. Die bis dahin entwickelte Systembeschreibung (Spezifikation bzw. Entwurf) muß dabei entsprechend konvertiert werden (Abb. 8). Diese Konvertierung ist aber nicht immer möglich - es fehlen oft die Schnittstellen zwischen den Methoden bzw. zwischen den Werkzeugen, die diese Methoden unterstützen.

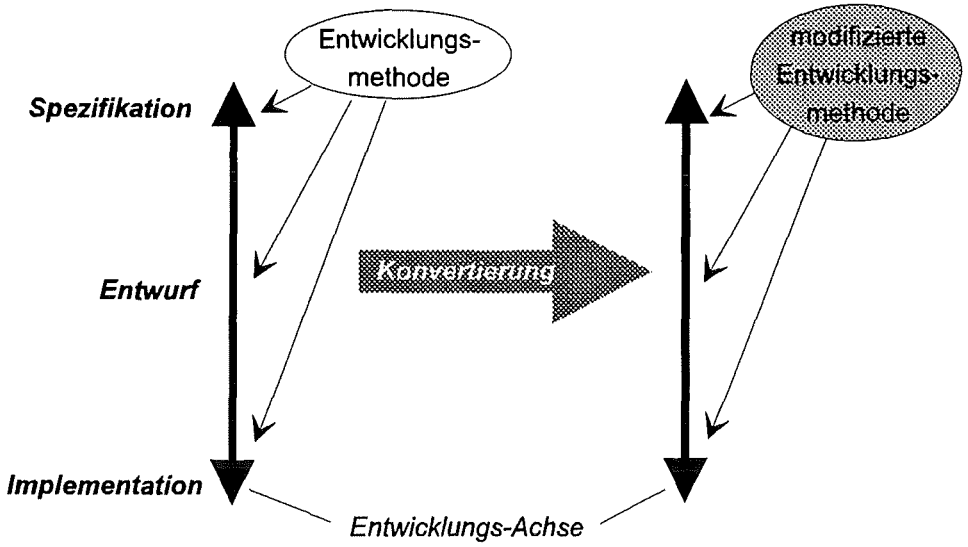


Abb. 8. Die Entwicklungsachse der heutigen Entwicklungsprozesse und die Konvertierung

Das Konzept der Meta-Modellierung [VaG91] löst dieses Problem der Schnittstellen. Die Methode wird mit Hilfe einer übergeordneten Methode (Meta-Methode) entwickelt. Entsprechende Tools, die als Meta-CASE Tools bezeichnet werden [Ald91], ermöglichen es, in einer komfortablen Weise eine gewünschte Methode zu entwickeln oder eine bereits vorhandene Methode zu erweitern. Da die verschiedenen Methoden immer von der selben Meta-Methode abgeleitet werden, ist es auch möglich, die bereits erstellten Systembeschreibungen zwischen den einzelnen Methoden auszutauschen. Das geschieht durch eine entsprechende Konvertierung der Modelle. Da die Konvertierungsprozesse aufwendig und zeitraubend sind, verlangsamt sich der Entwicklungsprozeß deutlich. Dies ist im Falle einer prototypischen Entwicklung, bei der die schnelle Implementation eines Prototyps sehr wichtig ist, ein nicht zu unterschätzender Nachteil.

2.2 Beschreibungsmethoden

In den drei Phasen der Entwicklung (Spezifikation, Entwurf, Implementation) werden verschiedene Beschreibungsmethoden eingesetzt. In den nächsten Kapiteln werden Methoden vorgestellt, die vor allem in der Spezifikations- und der Entwurfsphase verwendet werden. Da die selben Methoden in beiden Phasen benutzt werden können, ist eine klare Trennung zwischen Spezifikations- und Entwurfsmethoden nicht möglich [Sch91]. Hier werden die Methoden lediglich in zwei Gruppen eingeordnet: die Basismethoden und die kombinierten Methoden, die den kombinierten Einsatz mehrerer Basismethoden voraussetzen.

2.2.1 Basismethoden

Viele der in den letzten Jahren eingeführten Spezifkations- und Entwurfsmethoden lassen sich auf bestimmte Basismethoden zurückführen. Jede Basismethode ist nicht weiter auf andere Methoden reduzierbar, kann also als eine elementare Methode betrachtet werden.

Die Basismethoden werden vor allem nach ihrer Orientierung geordnet, d.h. aus welcher Sicht beim Einsatz der Methoden das gegebene System hauptsächlich betrachtet wird. So kann man folgende Sichten (zusammen mit Beispielmethoden) identifizieren:

- **funktionale Sicht** - Funktionsbaum, Datenflußdiagramm
- **datenorientierte Sicht** - Jackson-Diagramm, Entity-Relationship-Diagramm
- **objektorientierte Sicht** - OO-Methode
- **zustandsorientierte Sicht** - Zustandsautomat, Petri-Netz

2.2.1.1 Funktionsorientierte Methoden

Bei der Betrachtung eines Systems aus der funktionalen Sicht können **Funktionsbäume** eingesetzt werden. Mit ihrer Hilfe läßt sich eine statische oder dynamische Hierarchie von Funktionen oder Modulen, die eine bestimmte Funktionalität realisieren, darstellen. Es werden viele Typen von Hierarchien unterschieden werden, wie z.B. Aufrufhierarchie, Ist-Teil-von-Hierarchie, Zugriffsrecht-Hierarchie, Sichtbarkeits- und Dientsleistungs-Hierarchie [Bal93]. Alle diese Hierarchien haben einen ähnlichen Aufbau in Form eines Baumes, in dem die meist generischen Funktionen (Module) in der obersten Ebene lokalisiert sind. Die untergeordneten Elemente stellen die Verfeinerung der oberen Elemente dar.

Funktionsbäume können praktisch in jedem Typ von Lebenszyklus eingesetzt werden, weil sie nicht formal und präzise definiert sind, wodurch eine entsprechende Anpassung der Methode

möglich ist. Die fehlenden Verfeinerungskriterien sind aber auch nachteilig, da die Methode bei verschiedenen Entwicklern meist zu unterschiedlichen Ergebnissen führt.

Um den Datenfluß durch ein informationsverarbeitendes System anschaulich zu machen, können **Datenflußdiagramme** sehr hilfreich sein. Sie werden mit Hilfe von verschiedenen graphischen Symbolen konstruiert, welche Daten, Datenverarbeitungsvorgänge, Zugriffsmöglichkeiten dieser Vorgänge auf Daten und Schnittstellen zur Umwelt darstellen können. Es existieren mehrere Notationen zur Darstellung von Datenflußdiagrammen [ScN90]. In der Notation von DeMarco [DeM79] werden z.B. als Symbole gerichtete Kanten, Kreise, Linien und Kasten verwendet, die zusammen mit ihrer Bedeutung in der Abb. 9 dargestellt werden.

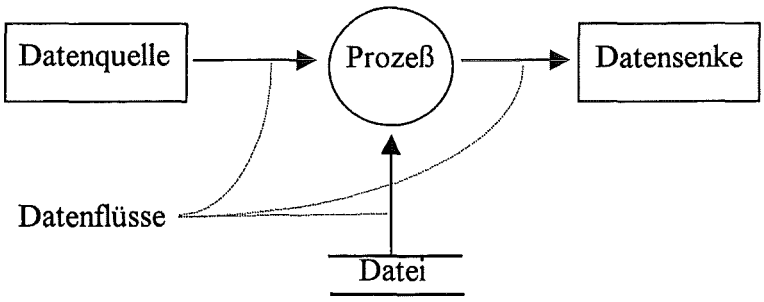


Abb. 9. Symbole der Datenflußdiagramme nach DeMarco

Um die Übersichtlichkeit von komplexen Diagrammen zu erreichen, können diese Diagramme hierarchisiert werden [Svo90]. Die Bildung der entsprechenden Hierarchieebenen erfolgt durch schrittweise Verfeinerung der Prozesse (Top-Down-Vorgehensweise). Die Verfeinerung endet auf der Ebene, in der sich nur elementare Prozesse befinden, deren weitere Verfeinerung keinen Sinn hat.

Datenflußdiagramme werden sehr oft eingesetzt, da sie sehr einfach zu lernen (vor allem wegen einer geringen Anzahl von Symbolen) und für viele Lebenszyklustypen geeignet sind. Diese Methode wird von vielen Tools unterstützt, meist als Bestandteil einer umfangreicheren kombinierten Methode (z.B.: SADT).

2.2.1.2 Datenorientierte Methoden

Die in [Jac75] eingeführten **Jackson-Diagramme** basieren auf einer graphischen Sprache zur Beschreibung von Daten und Kontrollstrukturen. Die Sprache besteht aus nur drei Grundkonstrukten: Sequenz, Alternative und Iteration (Abb. 10).

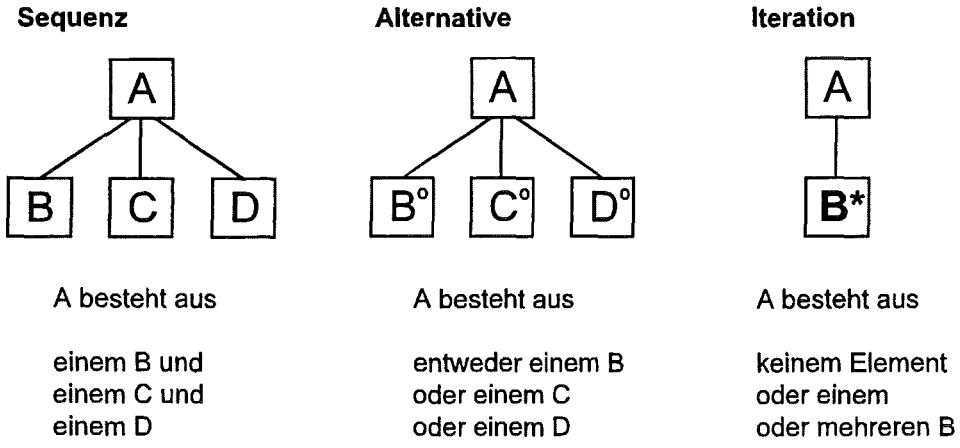


Abb. 10. Symbole der Jackson-Diagramme

In einer Sequenz treten einzelne Elemente jeweils nur einmal und in der angegebenen Reihenfolge auf. Beim Auftreten einer Alternative wird nur ein Element ausgewählt. Die Iteration beschreibt eine Sequenz von sich wiederholenden Elementen, wobei die Zahl der Elemente auch Null sein kann. Diese Konstrukte können auch für die Beschreibung von Kontrollstrukturen verwendet werden. In dem Fall werden sie anders interpretiert - sie besitzen eine andere Semantik.

Die Jackson-Diagramme basieren auf dem konventionellen Lebenszyklus. Evolutionäre Änderungen der Spezifikation werden nicht unterstützt.

Die hierarchische Modellierung von Datenstrukturen kann mit Hilfe von **Warnier/Orr-Diagrammen** realisiert werden [War81]. Diese semi-graphische Darstellung ist sehr einfach zu lesen (siehe Beispiel in der Abb. 11), erlaubt es aber trotzdem, komplizierte Datenstrukturen zu definieren. Diese Diagramme können, ähnlich wie die Jackson-Diagramme, auch für die Beschreibung der Programmabläufe verwendet werden [Hig78].

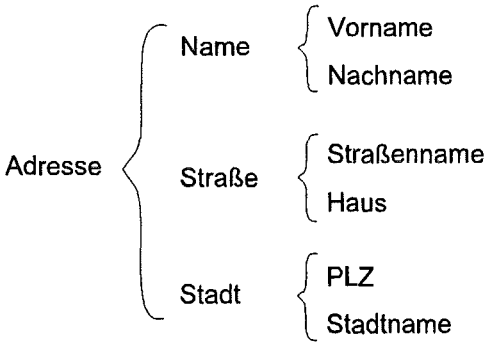


Abb. 11. Beispiel eines einfachen Warnier/Orr-Diagramms

Die Methode schreibt dem Entwickler eine festgelegte Reihenfolge der einzelnen Schritte vor [ScN90]:

1. Entwurf der Ausgabenstruktur
2. Entwurf der Eingabedatenstruktur
3. Ereignisanalyse
4. Physischer Datenentwurf
5. Programmentwurf

Wegen dieser festen Vorgehensweise können die Warnier/Orr-Diagramme praktisch nur im konventionellen Lebenszyklus eingesetzt werden [Fle94].

Die **Entity-Relationship-Methode** (ER-Methode) konzentriert sich vor allem auf die Beschreibung von Beziehungen zwischen Datenelementen. Es existieren unterschiedliche Ausprägungen der ER-Methode, die auf die ursprüngliche Definition von Chen [Che90] zurückgehen. In allen diesen Ausprägungen der ER-Methode werden einzelne Elemente des Systems als Entitäten bezeichnet. Für diese Entitäten werden entsprechende Attribute und Beziehungen zu anderen Entitäten definiert. Für jede Beziehung kann auch die Komplexität (auch als Kardinalität bezeichnet) angegeben werden.

Für die Darstellung von ER-Diagrammen werden unterschiedliche Notationen verwendet [ScS83]. In der Abb. 12 wird der Vergleich zwischen der sogenannten Min-Max-Notation und der sogenannten Krähnfuß-Notation präsentiert.

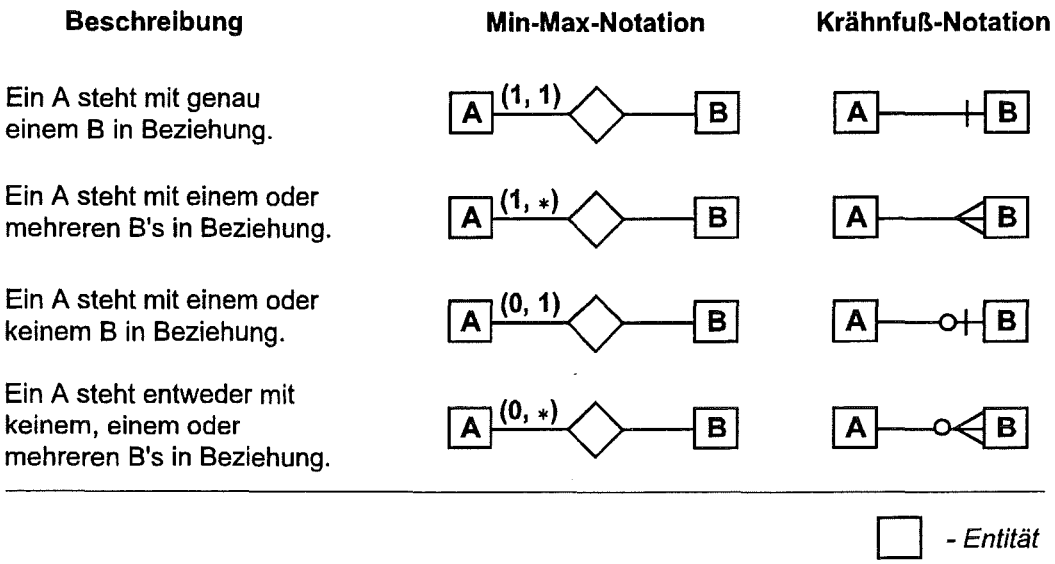


Abb. 12. Unterschiedliche Darstellung von Beziehungen in ER-Diagrammen [ScN90]

Ursprünglich wurde die ER-Methode vor allem für den Entwurf von Datenbanken verwendet [ScS83]. Sie hat aber in der letzten Zeit eine breite Akzeptanz als Werkzeug für die Kommunikation zwischen dem Systementwickler und dem Benutzer während der Spezifikations- und Entwurfsphase erreicht. Sie wird auch für die Modellierung von Unternehmensdaten verwendet [JOS94], [AMICE93].

Die ER-Methode, anders als bei den Jackson- und Warnier/Orr-Diagrammen, schreibt keine Vorgehensweise bei der Systementwicklung vor. Sie kann deswegen in verschiedenen Lebenszyklen eingesetzt werden.

Für die Beschreibung von Daten kann auch die **EXPRESS-Sprache** verwendet werden. Sie ist Bestandteil des ISO-10303-Standards STEP [ISO92a]. Die Sprache, ähnlich wie ER, stellt mehrere Konstrukte für die Datenbeschreibung zur Verfügung, mit denen die Entitäten, ihre Attribute und Beziehungen in einer textuellen Form definiert werden können. Die Sprache zeigt auch Gemeinsamkeiten mit objektorientierten Sprachen. So wird bei der Definition von Entitäten die Vererbung unterstützt. Es existiert auch eine graphische Variante EXPRESS-G, die aber nur über eine Untermenge der EXPRESS-Konstrukte verfügt (siehe Kapitel 5.3).

Zwar war die EXPRESS-Sprache ursprünglich für den Produktdatenaustausch gedacht, es finden sich aber in der letzten Zeit auch Anwendungen aus dem Bereich der Unternehmensmodellierung [AMICE93], in welcher EXPRESS zur Definition der Modellierungskonstrukte verwendet wird.

Als Hilfsmittel bei der Datenmodellierung kann ein **Data Dictionary** eingesetzt werden. Ein Data Dictionary ist ein Verzeichnis, das Informationen über die Struktur der Daten, ihre Eigenschaften

sowie ihre Verwendung enthält. Data Dictionary eignet sich zur Überprüfung der Redundanzfreiheit oder Widerspruchsfreiheit der Daten [Bal93].

Die Datenstrukturen und Datenelemente eines Data Dictionary werden nach DeMarco [DeM79] in einer modifizierten Backus-Naur-Form (vgl. [Bac60]) dargestellt. Die wichtigsten Symbole dieser Notation werden in der Tab. 1 präsentiert.

Symbol	Bedeutung	Beispiel
=	Äquivalenz	A = B + C
+	Sequenz	X = X1 + X2 + X3
[ ]	Auswahl (entweder ... oder)	A = [B | C]
{ }	Wiederholung	A = { B }
M { } N	Wiederholung von M bis N	A = 1 { B } 10
( )	Option, äquivalent zu: **0 { } 1**	A = B + (C)

Tab. 1. Symbole der Notation zur Beschreibung der Daten in einem Data Dictionary

2.2.1.3 Objektorientierte Methoden

Mit den objektorientierten Methoden wird die reale Welt in eine Sammlung von abstrakten Objekten abgebildet [Boo86]. Die statischen Eigenschaften von Objekten werden in Form von Attributen definiert und das dynamische Verhalten mit Hilfe von Operationen (Methoden) beschrieben. Durch die **Einkapselung** von Objekten ist die Manipulation der Attribute nur mit Hilfe der zur Verfügung gestellten Operationen möglich. Zur Vollendung einer Operation kann ein Objekt Operationen anderer Objekte verwenden. Die Werte der Attribute eines Objektes können auf andere Objekte verweisen.

Gleichartige Objekte werden in **Klassen** zusammengefaßt - die Objekte einer Klasse verfügen über die gleichen Attribute und die gleichen Operationen. Durch das Konzept der **Vererbung** ist es möglich, von einer Klasse (die als Superklasse bezeichnet wird) eine andere Klasse (die als Subklasse bezeichnet wird) mit ähnlichen Eigenschaften abzuleiten. Eine Subklasse erbt alle Attribute und Operationen ihrer Superklasse, wobei für die Subklasse gewöhnlich zusätzliche Attribute und/oder Operationen spezifiziert werden. Damit wird eine neue Klasse definiert, die als eine Erweiterung der ursprünglichen Klasse betrachtet werden kann. Das Konzept in dem eine

Subklasse von mehreren Superklassen abgeleitet werden kann, wird mit dem Begriff Mehrfachvererbung bezeichnet.

Wenn eine Subklasse nicht nur als Erweiterung ihrer Superklasse dienen soll, können die ererbten Operationen durch **Überschreiben** neu definiert werden. Die überschriebenen Operationen können dabei die ererbten Attribute auf andere Weise verwenden bzw. manipulieren. In dem Fall erhalten die ererbten Attribute eine neue Interpretation. Die beim Überschreiben geänderte Eigenschaften einer Klasse können durch die weitere Vererbung an die Subklassen dieser Klasse weitergegeben werden - es sei denn, sie werden wieder in dieser Subklassen überschrieben.

Über die Vorteile der objektorientierten Methoden vor allem gegenüber funktionsorientierten Methoden wird intensiv diskutiert - die eindeutige Aussage läßt sich nicht formulieren [Loy89]. Der Anwendungsbereich der OO-Methoden wird trotzdem immer größer - es existieren inzwischen zahlreiche Anwendungen für die Unternehmensmodellierung wie z.B. in [GKR92].

2.2.1.4 Zustandsorientierte Methoden

Mit zustandsorientierten Methoden lassen sich die Zustände eines Systems und die möglichen Übergänge zu anderen Zuständen (als Reaktion des Systems auf Ereignisse aus der Umwelt) beschreiben. Für eine solche Beschreibung können **endliche Automaten** verwendet werden. Ein endlicher Automat ist eine hypothetische Maschine, die sich in einem bestimmten Zeitpunkt nur in einem von mehreren möglichen Zuständen befinden kann. Als Reaktion auf eine Eingabe generiert die Maschine eine bestimmte Ausgabe und geht zu einem neuen Zustand über [Dav88].

Es werden zwei Typen von endlichen Automaten unterschieden: kombinatorische und sequentielle Automaten. Bei der kombinatorischen Automaten ergibt sich der neue Zustand unabhängig vom aktuellen Zustand nur aufgrund der entsprechenden Eingabe. Bei der sequentiellen Automaten hängt dagegen der neue Zustand von dem aktuellen Zustand, und damit auch von früheren Eingaben, ab [ScN90].

Zur Beschreibung von kombinatorischen Automaten können Entscheidungstabellen verwendet werden, in denen dargestellt wird, welche Eingaben zu welchen Zuständen führen. Zur Darstellung von sequentiellen Automaten können Zustandsdiagramme eingesetzt werden. In diesen Diagrammen werden Zustände als Kreise und Zustandsübergänge als Pfeile dargestellt. Eingaben und Ausgaben werden als Beschriftung der Pfeile repräsentiert. Die Abb. 13 zeigt einen sequentiellen Automat, der die Funktionsweise einer Digitaluhr beschreibt.

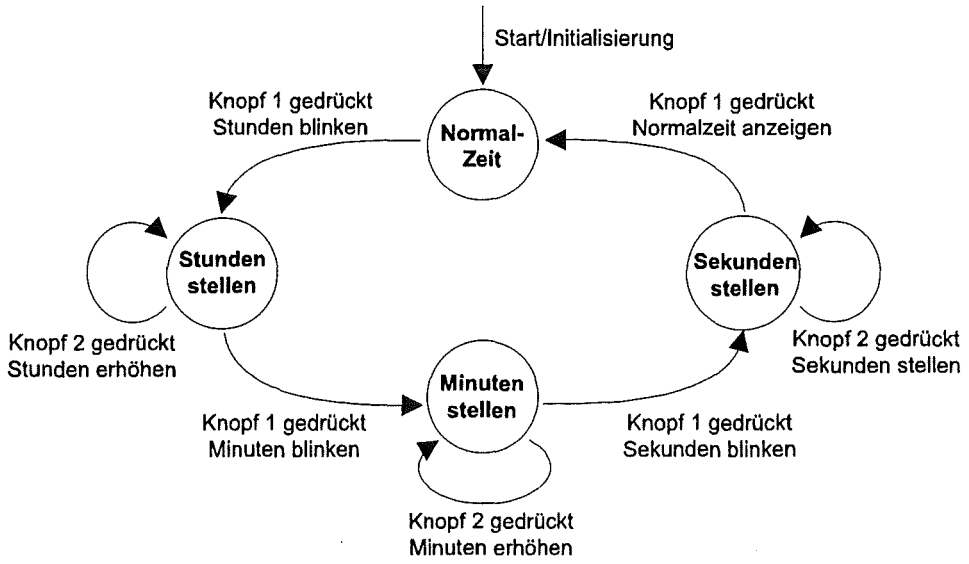


Abb. 13. Ein sequentieller, endlicher Automat: die Digitaluhr [Bal93]

Bei komplexen Systemen wird die Darstellung in Form von Zustandsdiagrammen wegen der großen Zahl möglicher Zustände sehr unübersichtlich. Als eine alternative Darstellung werden in dem Fall Zustandstabelle verwendet. In der einzelnen Spalten dieser Tabelle werden entsprechend die aktuelle Zustände, Eingaben, Ausgaben und Folgezustände aufgelistet.

Für die Modellierung von komplexen Systemen mit nebenläufigen Vorgängen sind die **Petri-Netze** [Rei86] sehr gut geeignet. In den sogenannten **Place/Transition-Netzen** (P/T-Netze), die eine der einfachsten Variante der Petri-Netze ist, werden zur Beschreibung eines Systems drei Grundstrukturen verwendet: Stellen, Transitionen und Kanten. Stellen, die graphisch als Kreise dargestellt werden, repräsentierten passive Komponenten des Systems und dienen als Speicher für Objekte. Diese Objekte werden als Marken (oder Tokens) bezeichnet. Alle Marken eines P/T-Netzes definieren den aktuellen Zustand des Systems. Aktive Komponenten des Systems werden mit Transitionen repräsentiert, die graphisch als Rechtecke oder Balken dargestellt werden. Stellen und Transitionen werden mit Kanten (graphisch als gerichtete Pfeile dargestellt) miteinander verbunden (Abb. 14).

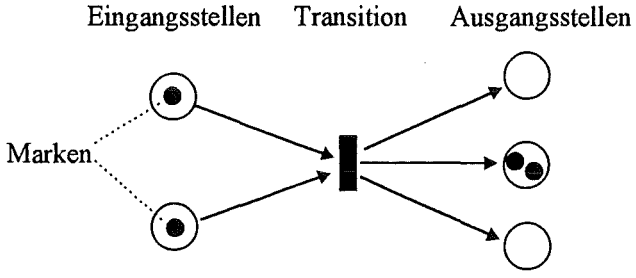


Abb. 14. Die Grundstrukturen der P/T-Netze

Jeder Stelle kann eine Kapazität zugeordnet werden, die die maximale Zahl der Marken in der entsprechenden Stelle definiert. Jeder Kante kann ein Gewicht zugeteilt werden, das definiert, wieviele Marken bei der Zustandsänderung des Systems von der Stelle abgenommen bzw. in der Stelle abgelegt werden. Das geschieht durch Schalten (bzw. Feuern) einer Transition, die mit dieser Stelle durch diese Kante verbunden ist. Beim Schalten können Transitionen Marken aus ihren Eingangsstellen konsumieren und Marken in ihrer Ausgangstellen ablegen. Damit wird das System in einen neuen Zustand überführt. Eine Transitionen wird nur dann geschaltet, wenn folgende Bedingungen erfüllt werden:

- In jeder Eingangsstelle befindet sich mindestens eine Marke, wobei die Zahl der Marken in der Stelle größer oder gleich dem Gewicht der Kante von dieser Stelle zu der Transition ist
- Für jede Ausgangsstelle ist die Summe der Marken in der Stelle und des Gewichtes der Kante von der Transition zu dieser Stelle kleiner oder gleich der Kapazität dieser Stelle

Heutzutage werden vor allem Petri-Netze höherer Stufen verwendet. In **Pr/T-Netzen** (Prädikat/-Transitionsnetzen) repräsentieren die Marken beliebige, identifizierbare Objekte und werden damit als Informationsträger gesehen [Gen86]. Anstelle des Gewichtes verfügen hier die Kanten über Beschriftungen (in Form von Variablen und Konstanten), die definieren, welche und wieviele Objekte beim Schalten einer Transition von einer Stelle entfernt bzw. in einer Stelle abgelegt werden. Die Bedingungen für Schalten können durch entsprechende Transitionbeschriftung beliebig ergänzt werden. Die Beschriftungen werden in einer Programmiersprache definiert - welche Sprache dabei verwendet wird, hängt von dem Tool, die für die Erstellung der Pr/T-Netze eingesetzt wird.

Die **CP-Netze** (Coloured Petri Nets) [Jen86], die eine Erweiterung von Pr/T-Netzen sind, erlauben zusätzlich die Definitionen von formalen Funktionen, die den Kanten zugeordnet werden können. Mit Hilfe der Funktionen können die durch die Marken repräsentierten Objekte manipuliert und damit die Bedingungen für das Schalten der Transitionen beeinflußt werden.

In **NR/T-Netzen** (Nested Relation/Transition Nets) [OSS94] wird jeder Stelle eine komplexe, verschachtelte Datenstruktur zugewiesen. Die Datenstruktur kann mit verschiedenen Konstrukten, wie Klassifikation, Aggregation, Spezialisierung und Gruppierung, definiert werden. Es wird dafür die Entity-Relationship-Methode verwendet.

Die zustandsorientierten Methoden, u.a. verschiedene Dialekte der Petri-Netze, können vor allem als Beschreibungsmethoden betrachtet werden. Sie legen keine Vorgehensweise bei der Systemmodellierung und -entwicklung fest. Sie können deswegen in allen Typen von

Lebenszyklen eingesetzt werden. Petri-Netze sind dabei auch für die Entwicklung von Prototypen gut geeignet (vgl. [Sch89], [Did92]). Sie werden sowohl für die Modellierung von Unternehmensaktivitäten [JOS94] als auch für die Ausführung von diesen Unternehmensaktivitäten [DNB93] eingesetzt.

2.2.2 Kombinierte Methoden

Heutzutage werden Methoden eingesetzt, die ein System nicht nur aus einer Sicht betrachten. Es werden komplexe Methoden verwendet, die als Kombination von mehreren Basismethoden definiert werden. Durch die Berücksichtigung von mehreren Aspekten (Funktionen, Daten, Kontrolle) wird die Beschreibung des Systems vollständiger. Der Einsatz von mehreren Methoden führt zu mehreren unterschiedlicher Teilbeschreibungen des Systems. Durch Analyse und Vergleich dieser Beschreibungen können Unvollständigkeiten und Widersprüche entdeckt werden.

2.2.2.1 Methoden für strukturierte Analyse und Entwurf

Die **Strukturierte Analyse** (SA) kombiniert folgende Basismethoden: Datenflußdiagramme, Entscheidungstabellen und Entscheidungsbäume [DeM79]. Die Definition des Systems fängt mit der Erstellung eines Datenflußdiagramms an, das den Datenfluß innerhalb des Systems und zwischen dem System und externen Prozessen auf der obersten Ebene beschreibt. Durch die Einführung von mehreren Hierarchieebenen wird das Diagramm schrittweise verfeinert. Wenn die Dekomposition aller Prozesse ihre einfachste funktionale Ebene erreicht hat, werden die übrigen Basismethoden zur endgültigen Definition des Systems eingesetzt. Zusätzlich wird für die detaillierte Beschreibung der Abläufe Pseudocode eingesetzt. Bei Einsatz von Pseudocode werden die Kontrollstrukturen ähnlich wie in einer Programmiersprache dargestellt, die Anweisungen werden dagegen in einer natürlichen Sprache formuliert. Die detaillierte Beschreibung der Daten wird ein einem Data Dictionary erfaßt.

Für Systeme mit zeitabhängigen Verhalten ist die **RT-Methode** (Real-Time-Analysis-Methode) gut geeignet [WaM85]. Sie erweitert die SA-Methode um Kontrollflußdiagramme. Diese Diagramme beschreiben den Fluß von Signalen, welche die Abläufe in dem System steuern. Mit der Methode läßt sich das Triggern von Prozessen in Abhängigkeit von externen Ereignissen und von komplexen Bedingungen modellieren. Zusätzlich können auch die externe Zeitanforderungen berücksichtigt werden, die z.B. die Zeit zwischen einem Eingabeereignis und einer Ausgabe des

Systems beschreiben können. Für Systeme mit komplexen Daten kann zudem die Entity-Relationship-Methode verwendet werden.

Die **IDEF**-Methode (ICAM-Definition) wurde im Rahmen des ICAM-Projektes der amerikanischen Luftwaffe entwickelt [ICAM81]. Sie kombiniert mehrere Modellierungsmethoden, wobei die am meisten bekannten Methoden sind: $IDEF_0$, $IDEF_1$ und $IDEF_2$. **$IDEF_0$** wird für die Erstellung von funktionalen Modellen verwendet und entspricht der SADT-Methode (Software Analysis and Design Technique), die für die Modellierung von komplexen Systemen entwickelt wurde [DMR79]. Bei dieser Methode wird ein System mit Hilfe von hierarchischen Diagrammen beschrieben. Auf jeder Abstraktionsebene werden immer die gleichen graphische Konstrukte verwendet: Kästchen, die zur Dekomposition von Aktivitäten dienen, und Kanten, die den Informationsfluß zwischen den Aktivitäten darstellen. **$IDEF_1$** ist für die Datenmodellierung vorgesehen und basiert auf der Entity-Relationship-Methode. Dynamische Aspekte können mit der **$IDEF_2$**-Methode betrachtet werden. Das hier zugrunde liegende Konzept der Petri-Netze macht diese Methode zu einem Werkzeug, das für die Simulation im Bereich der Unternehmensmodellierung eingesetzt werden kann.

2.2.2.2 Objektorientierte Methoden

Bei den kombinierten objektorientierten Methoden können Methoden zur Analyse, die als **OOA-Methoden** (Object-Oriented-Analysis-Methoden) bezeichnet werden, und Methoden zum Entwurf, die als **OOD-Methoden** (Object-Oriented-Design-Methoden) genannt werden. Es ist eine Vielzahl von OOA- und OOD-Methoden entworfen worden. Als Beispiel werden hier die OOD- und die OOA-Methode nach Coad/Yourdon kurz dargestellt.

Die **OOA-Methode** nach Coad/Yourdon [CoY90] verwendet für eine OO-Methode typische Konstrukte bzw. Konzepte, wie z.B. Objekte, Klassen, Vererbung. Da sie jedoch in vielen Fällen nicht ausreichend sind, um eine komplexe, reale Welt zu beschreiben, werden zusätzlich andere Basismethoden integriert. Es handelt sich hier vor allem um die Entity-Relationship-Methode, die für die Beschreibung der Beziehungen zwischen Objekten bzw. Klassen verwendet wird. Zudem werden auch Zustandsdiagramme und Pseudocode für die Beschreibung des Verhaltens der Objekte eingesetzt.

Die Modellierung des Systems erfolgt in mehreren Schritten:

1. Identifizieren der Klassen und Objekte.
2. Definition der Beziehungen zwischen Klassen und/oder Objekten. Es werden dabei zwei Strukturierungsformen verwendet: Generalisierungs- bzw. Spezialisierungsstrukturen (Gen-Spec-Structure) und Ganzheits-Teile-Beziehung (Whole-Part-Structure).
3. Identifizieren der Subjektstruktur. In diesem Schritt werden die zusammengehörenden Klassen zu Subjekten zusammengefaßt. Subjekte dienen zu Strukturierung der Modelle, was bei einer großen Anzahl von Klassen die Übersichtlichkeit und Handlichkeit dieser Modelle erhöht. Klassen, die keine Beziehungen mit anderen Klassen haben, stellen mit sich selbst eigene Subjekte dar.
4. Definition der Attribute der Klassen. Für jedes Attribut werden der Name, die Beschreibung und Gültigkeitsbedingungen (z.B. Grenzwerte, Zugriffsbedingungen, Maßeinheit usw.) spezifiziert. Anschließend werden die Exemplarverbindungen (instance connection) zwischen einzelnen Objekten identifiziert. Die Verbindungen entsprechen den Relationen der ER-Methode.
5. Erweiterung der Objekte um Dienste (services). Dienste sind Aktivitäten, die von Objekten ausgeführt werden sollen. Die Dienste können mit Hilfe von Nachrichtenverbindungen verschiedene Botschaften an andere Objekte schicken. Die Spezifikation der Dienste erfolgt mit Hilfe von Zustandsdiagrammen.

Klassen mit ihren Attributen und Diensten und die Beziehungen zwischen den Klassen werden graphisch dargestellt. Für die Darstellung der Klassen werden abgerundete Rechtecke verwendet, in denen die Namen der Attribute und der Dienste aufgelistet werden. Für die Darstellung der Beziehungen zwischen Klassen werden Linien mit zusätzlichen Symbolen verwendet (Abb. 15).

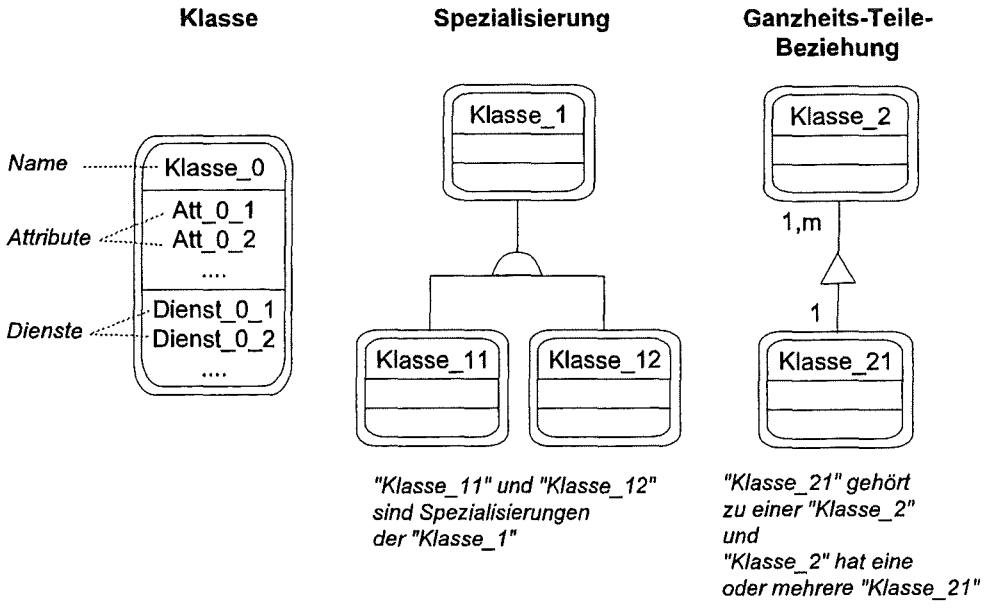


Abb. 15. OOA-Methode nach Coad/Yourdon: Symbole zur Darstellung der Klassen und ihren Beziehungen

Während die oben präsentierte OOA-Methode sich mit allgemeinen Aspekten der Modellierung befaßt, werden in der **OOD-Methode** nach Coad/Yourdon [CoY91] vor allem implementierungsbezogene Aspekte betrachtet, etwa:

1. Definition der Problembereichskomponente (problem domain component),
2. Entwurf der Benutzerschnittstelle (human interaction component),
3. Definition der Task-Management- und Daten-Management-Komponente (task management component, data management component)

Als Ausgangspunkt für den Entwurf mit der OOD-Methode dienen Ergebnisse der mit der oben präsentierten OOA-Methode. Sie werden bei beim Entwurf ergänzt bzw. modifiziert, etwa durch:

- Zusammenfassung von problemspezifischen Klassen,
- Abfassung eines Protokolls (als Menge der gemeinsamen Dienste) durch Hinzufügung einer verallgemeinernden Klasse,
- Beschreibung der Benutzer
- Entwurf der Interaktionen mit Benutzer
- Entwurf einer Benutzerschnittstelle

Die Notation in der OOD-Methode unterscheidet sich nicht von der in der Abb. 15 präsentierten Notation.

2.2.2.3 Methoden zur ganzheitlichen Unternehmensmodellierung

Die bisher erörterten kombinierten Methoden wurden für die Softwareentwicklung konzipiert und sie werden vorwiegend in diesem Bereich eingesetzt. Die hier präsentierten Methoden eignen sich dagegen zum (bzw. wurden speziell entwickelt für) Erstellen von ganzheitlichen Unternehmensmodellen. Sie erlauben es, ein Unternehmen aus verschiedenen Sichten zu betrachten. Die Modellierung kann dabei folgende Aspekte umfassen: Funktion (Abläufe), Information, Betriebsmittel, Organisation und Wirtschaftlichkeit. In den meisten Methoden werden nur bestimmte Aspekte der Modellierung bevorzugt, vor allem die Funktions- und Informationsmodellierung.

Die **Information-Engineering**-Methode war zwar ursprünglich für die Softwareentwicklung gedacht, eignet sich aber auch zur Unternehmensmodellierung [Mar89]. Die Methode befaßt sich nur mit der Modellierung von Funktionen und Daten. In der Aktivitätssicht werden Geschäftsprozesse zusammen mit ihren Input- und Outputdaten beschrieben. Die vollständige Definition von diesen Daten erfolgt in der Datensicht, wo solche Aspekte wie Produkte, Organisation und Betriebsmittel betrachtet werden. Zusätzlich zu diesen beiden

Modellierungssichten werden auch mehrere Modellierungsebenen (Information Strategy Planning, Business Area Analysis, System Design und Construction) spezifiziert, die eine systematische Entwicklung der Anwendung ermöglichen.

Die **INCOME**-Methode [SNS89] befaßt sich außer mit der Definition der Funktionen und Informationen auch mit organisatorischen Aspekten. Zur Spezifikation des Ablaufsmodells wird eine Variante der höheren Petri-Netze, die NR/T-Netze (siehe Kapitel 2.2.1.4) verwendet. Sie ermöglichen, außer der Darstellung und Simulation von parallelen Abläufen, auch die Abbildung von Restriktionen für unzulässige Zustände oder Aktivitätsfolgen. Zeitliche Bedingungen und Ausnahmesituationen können ebenfalls behandelt werden. Die Ablaufmodelle bilden ein Gerüst, das mit den Organisations- und Informationsmodellen vervollständigt wird (Abb. 16).

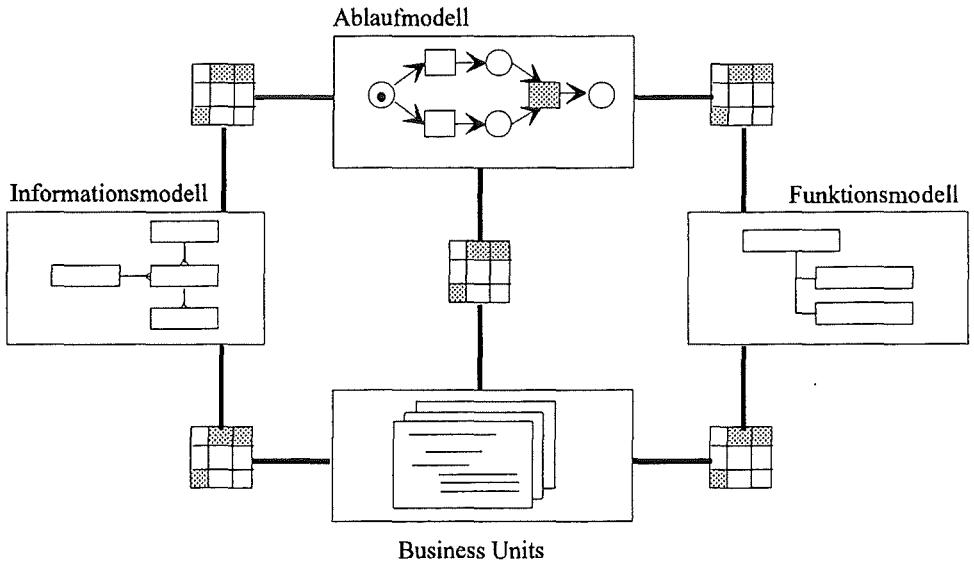


Abb. 16. Unternehmensmodellierung in der INCOME-Methode.

Die organisatorischen Aspekte werden in den sogenannten Business-Units und in dem Funktionsmodell berücksichtigt. Die Information über den Organisationsaufbau, die in den Modellen beinhaltet ist, wird mit den Aktivitäten des Ablaufsmodells (und miteinander) mit Hilfe von Matrizen verknüpft. Es werden damit entsprechende Verantwortungsbereiche festgelegt. Die Matrizen können auch Referenzen zum Informationsmodell definieren. Das Informationsmodell beschreibt entsprechende Daten, indem diese als Objekte dargestellt werden, deren Attribute und gegenseitige Beziehungen mit Hilfe der ER-Methode beschrieben werden.

In dem Modellierungskonzept der Architektur integrierter Informationssysteme **ARIS** wird ein Informationssystem in vier Phasen erstellt, die von der fachlichen Ausgangslösung, über das

Fachkonzept (requirements definition) und das DV-Konzept (design specification) zu der technischen Implementierung (implementation description) führen [Sch92].

In der ersten Phase wird ein System mit Hilfe von Vorgangsketten beschrieben. Ein Vorgang, der Grundbaustein der Vorgangsketten, ist "ein zeitverbrauchendes Geschehen, das durch ein Ereignis gestartet wird und durch ein Ereignis beendet wird". Einzelne Vorgänge können sowohl zur Werkstofftransformation als auch zur Informationsverarbeitung dienen. Andere Elemente der Vorgangsketten stellen die Ereignisse, Zustände, Betriebsmittel, Werkstoffe und Organisationseinheiten dar.

In der zweiten Phase (Fachkonzept) werden einzelne Sichten des Anwendungssystems modelliert. Es werden 4 Sichten unterschieden: Funktions-, Daten-, Organisations- und Steuerungssicht (Abb. 17). Einzelne Elemente der Vorgangsketten werden damit zu den entsprechenden Sichten zugeordnet.

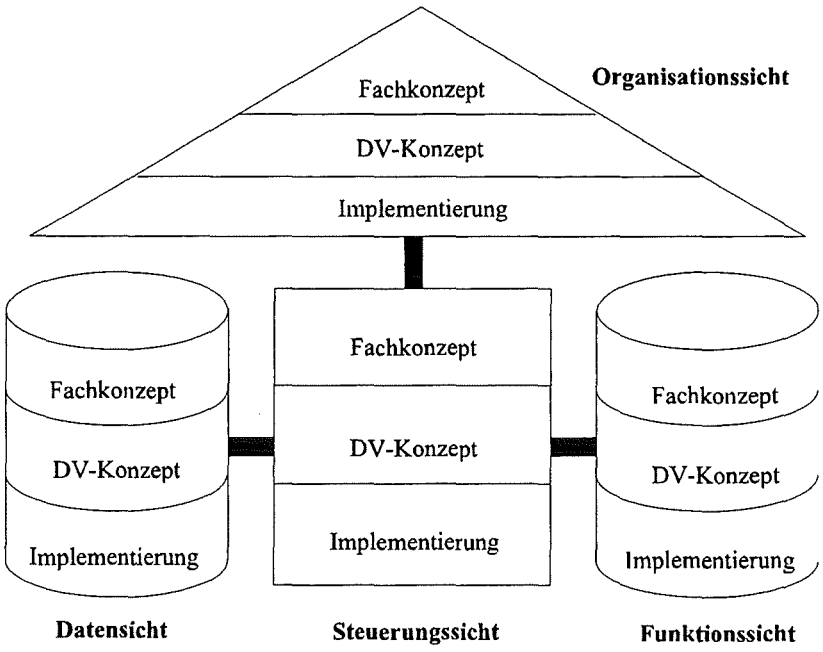


Abb. 17. Modellierungssichten im ARIS-Konzept.

In der technischen Implementierung erfolgt die Umsetzung des Fachkonzeptes in die physischen Datenstrukturen, Hardware- und Softwarekomponenten.

Eine weitere Methode zur ganzheitlichen Unternehmensmodellierung bietet auch das **CIMOSA**-Konzept (CIM Open System Architecture), das in mehreren ESPRIT[2]-Verbundprojekten entwickelt wurde [AMICE93]. Das Modellierungskonzept wird in Form einer dreidimensionalen Referenzarchitektur definiert (Abb. 18), bei der jede Dimension einer Komponente des Modellierungsprozesses entspricht. Die drei Komponenten sind: Instantiierungs-, Generierungs- und Ableitungsprozeß.

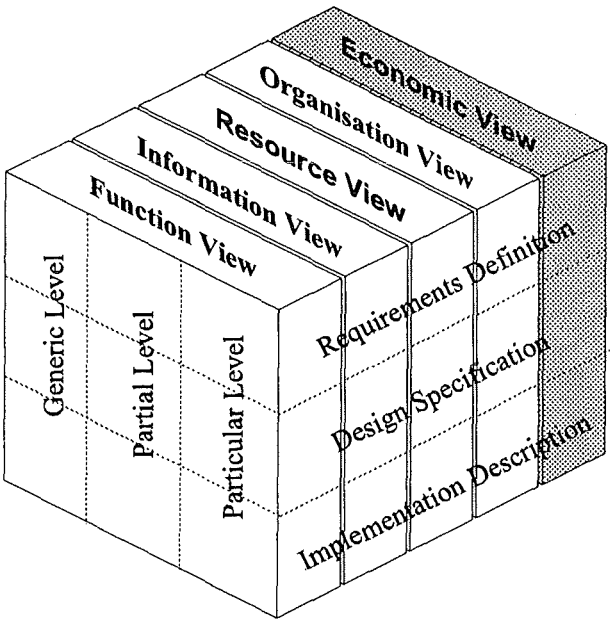


Abb. 18. Das CIMOSA-Modellierungskonzept

Im Instantiierungsprozeß werden Modelle unterschiedlicher Detaillierungsgrade erstellt. Die erste Ebene (Generic Level) kann als ein Referenzkatalog von vordefinierten Modellierungsbausteinen betrachtet werden. Die zweite Ebene (Partial Level) ist ein Katalog von partiellen Modellen, die für spezifische Anwendungsbereiche (etwa Automobilmontage, Monitoringanwendung, Robotermanipulation usw.) vordefiniert wurden. Die Modelle der dritten Ebene (Particular Level) werden mit Hilfe von Bausteinen der ersten Ebene erstellt. Um den Modellierungsprozeß zu beschleunigen, können vordefinierte partielle Modelle verwendet werden (ähnlich wie Macros).

[2] ESPRIT ist Abkürzung von European Strategic Program for Research and Development in Information Technology

Im Generierungsprozeß wird ein Unternehmen aus verschiedenen Sichten betrachtet:

1. die Funktions-Sicht (Function View) beschreibt die Funktionalität der Anwendung durch Definition der Geschäftsprozesse
2. die Informations-Sicht (Information View) dient zur Modellierung der Daten, die von den Geschäftsprozessen benötigt bzw. erzeugt werden
3. die Ressourcen-Sicht (Resource View) befaßt sich mit der Struktur und den Eigenschaften der in den Geschäftsprozessen eingesetzten Betriebsmittel (Menschen, Maschinen und Anwendungen)
4. die Organisations-Sicht (Organisation View) definiert die organisatorische Struktur des Unternehmens
5. die Ökonomische Sicht (Economic View) befaßt sich mit der Wirtschaftlichkeitsanalyse der entwickelten Modelle

Die ökonomische Sicht ist noch nicht Bestandteil des CIMOSA-Konzeptes, wurde aber als Erweiterung vorgeschlagen, die den fehlenden Aspekt der wirtschaftlichen Analyse der Modelle in das Konzept integriert [Neu94].

Die funktionalen Aspekte werden in dem Konzept mit Hilfe einer modifizierten SADT-Methode beschrieben. Die Information (Daten, Ressourcen und Organisation) wird mit der ER-Methode modelliert. Für eine alternative Darstellung der Funktionen und Informationen wird die Datenbeschreibungssprache EXPRESS eingesetzt.

In dem Ableitungsprozeß (Derivation Process) wird ein kompletter Lebenszyklus der Modelle beschrieben. Er beginnt mit der Anforderungsebene, in der die Ziele und Einschränkungen sowie Modellierungsbereiche definiert werden. Auf der Entwurfebene werden die Modelle detailliert, wobei verschieden Alternativen ausgearbeitet werden und durch Simulation die optimale Lösung für die vorliegende Anforderungen gefunden wird. Auf der Implementationsebene werden diese detaillierte Modelle in ausführbare Systems umgesetzt, wobei die Restriktionen konkreter Geräte, Anwendungen und Datenbanken berücksichtigt werden.

Im Gegensatz zu vielen anderen Modellierungskonzepten sieht CIMOSA auch die Ausführung der Modelle vor, indem die Funktions- und Informationsmodelle interpretiert und die Steuerung der realen Abläufe im Unternehmen verwendet werden. Diese Aufgaben realisiert die sogenannte Integrierende Infrastruktur, deren Dienste auch die Integration der Informations-Technologien des Unternehmens ermöglicht. Die Integrierende Infrastruktur wird im Kapitel 2.3.3 näher vorgestellt.

2.2.3 Zusammenfassung

Die weiter oben erwähnten Basismethoden konzentrieren sich immer auf einen bestimmten Aspekt des zu entwickelnden Systems. Sie beschreiben das System nur aus einer Sicht, was bei komplexen Systemen, wie z.B. im CIM-Bereich, nicht ausreichend ist. Viele Informationen über die Struktur und das Verhalten eines Systems werden bei der Entwicklung nicht berücksichtigt, wodurch das Entstehen von Fehlern begünstigt wird. Aus diesem Grund werden heutzutage im CIM-Bereich (Modellierung des Unternehmens und Entwicklung der CIM-Infrastruktur) vor allem die ganzheitlichen Konzepte eingesetzt, die mehrere Basismethoden in einer komplexen Methode kombinieren.

Die hier vorgestellten ganzheitlichen Ansätze werden von entsprechenden Entwicklungs- bzw. Modellierungstools unterstützt. Zu diesen Tools gehört beispielsweise INCOME/STAR [OSS94], das die INCOME-Methode unterstützt. Mit dem Tool können verteilte Informationssysteme und Unternehmensabläufe mit Hilfe von NR/T-Netzen (Nested Relation/Transition Netze) und Entity-Relationship-Modellen modelliert und validiert werden. Zur Erstellung von SADT-Modellen kann das Design-Tool-Set verwendet werden [CPN89][IDEF89]. Das Tool-Set bietet zusätzlich die Möglichkeit, die erstellte SADT-Modelle automatisch in CP-Netze (Coloured Petri Nets) zu übersetzten. Die generierte CP-Netze können anschließend simuliert werden. Damit können die SADT-Modelle validiert werden.

Für das Erstellen von CIMOSA-Modellen wird das Tool CIMOSA/RD verwendet. Es ist ein Werkzeug, das speziell für den CIMOSA-Ansatz mit Hilfe von GraphTalk entwickelt wurde [GT91]. GraphTalk ist ein Meta-Tool zur Erzeugung von Modellierungstools. Das Tool unterstützt eine objektorientierte, graphische Sprache, die zur Definition von Meta-Modellen verwendet wird. Durch die Compilation eines Meta-Modells entsteht ein Modell, das in das Tool geladen werden kann. Damit erhält man ein neues Tool, dessen Funktionalität durch das Meta-Modell beschrieben ist. Ein großer Vorteil der hier eingesetzten Meta-Modellierung ist die Möglichkeit, die Änderungen der Funktionalität des zu entwickelnden Tools durch Änderungen des Meta-Modells durchzuführen. Diese Eigenschaft ist für ein CIMOSA-Modellierungstool sehr wichtig, da das CIMOSA-Konzept noch nicht vollständig definiert ist. Die Definition der Modellierungskonstrukte, die von dem Tool unterstützt werden sollen, kann sich daher immer noch ändern - es können sogar neue Konstrukte hinzugefügt werden. Das Konzept der Meta-Modellierung ermöglicht die Anpassung der Funktionalität des Tools an neue Anforderungen, die jedoch leider mit relativ großem Aufwand zu realisieren ist (etwa durch die Tatsache, daß die

Meta-Modelle sehr komplex und damit sehr unhandlich sind, sowie durch die Notwendigkeit, bei jeder Änderung der Meta-Modelle die bis dahin entwickelten Modelle zu konvertieren).

Mit GraphTalk lassen sich graphische Editor-Tools schnell entwickeln, die jedoch keine Simulation unterstützen können. Aus diesem Grund werden die mit dem CIMOSA/RD-Tool erstellte CIMOSA-Modelle zur Validierung zum McCIM-Testbett übertragen [Bro93]. McCIM basiert auf PACE, einem Tool, das zur Erstellung und Simulation von höheren Petri-Netzen dient [PACE93]. PACE wurde im Rahmen des ESPRIT-Projektes VOICE (Validating OSA in Industrial CIM Environments) erweitert, so daß die Darstellung und Simulation von CIMOSA-Modellen mit Hilfe von Petri-Netzen möglich ist [Bog93]. Nach der Validierung können eventuelle Änderungen und Erweiterungen der Modelle mit dem CIMOSA/RD-Tool vollgezogen werden. Die geänderten Modelle werden dann wiederum zum McCIM-Testbett übertragen und validiert. Die Übertragung der Modelle zwischen den beiden Tools ist allerdings aufwendig und bereitet viele Probleme, die mit den unterschiedlichen Datenformaten und Darstellungen der Modellierungskonstrukte verbunden sind.

2.3 CIM

In letzter Zeit wurden viele verschiedene Definitionen und Interpretationen des CIM-Begriffes (Computer Integrated Manufacturing) sowohl von den Herstellern als auch von den Hochschulinstituten entwickelt. Viele davon basieren jedoch auf der Definition nach AWF[3] [AWF85], die eine recht weit verbreitete, herstellerunabhängige Empfehlung zum Thema CIM ist. Nach dieser Definition ist CIM ein integrierter EDV-Einsatz in allen mit der Produktion zusammenhängenden Unternehmensbereichen, wie:

- Entwicklung und Konstruktion (Computer Aided Design: **CAD**)
- Arbeitsplanung (Computer Aided Planning: **CAP**)
- Produktionsplanung und -steuerung (**PPS**)
- Teilefertigung, Montage, Lagerung und Transport (Computer Aided Manufacturing: **CAM**)
- Qualitätssicherung (Computer Aided Quality Assurance: **CAQ**)

CIM wird in zwei Bereiche unterteilt: technischer Bereich CAD/CAM und administrativer Bereich PPS (Abb. 19).

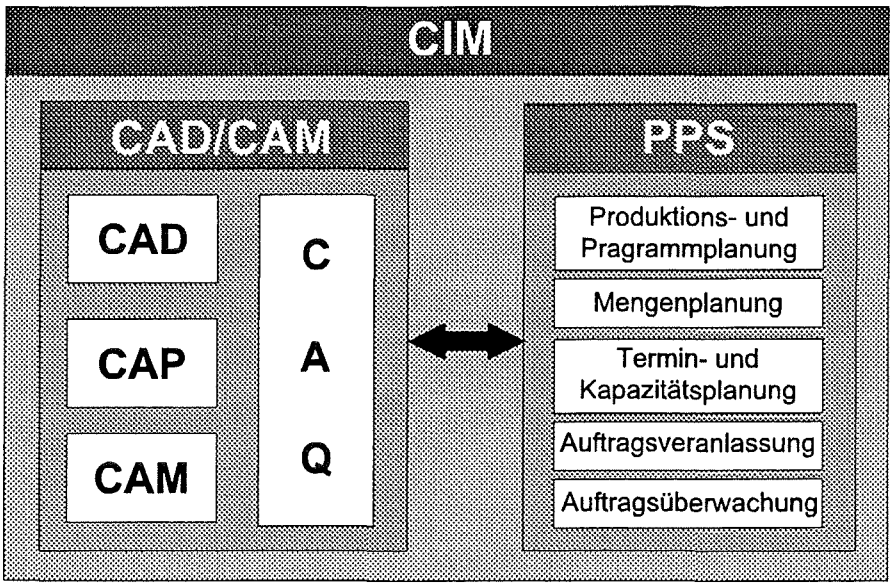


Abb. 19. Die CIM-Definition nach AWF

In den folgenden Kapiteln werden die einzelnen Komponenten eines CIM-Systems näher dargestellt. Ferner wird die Problematik der Integration der Komponenten erläutert. Anschließend wird das Konzept der Integrierenden Infrastruktur des CIMOSA-Ansatzes dargestellt.

---

[3] AWF steht für Ausschuß für Wirtschaftliche Fertigung e.V.

2.3.1 CIM-Komponenten

Computer Aided Design (CAD) umfaßt alle rechnerunterstützten Aktivitäten, die im Rahmen von Entwicklungs- und Konstruktionstätigkeiten ausgeführt werden. Im Mittelpunkt steht hier eine interaktive Arbeit mit graphisch orientierten Werkzeugen zur Erstellung und Bearbeitung von technischen Zeichnungen und Konstruktionsmodellen. Die Ergebnisse der Tätigkeiten (Entwicklung, technische Berechnung, Konstruktion und Zeichnungserstellung) werden in einer Datenbank abgelegt und stehen anderen CIM-Komponenten zur Verfügung.

Computer Aided Planning (CAP) ist ein Sammelbegriff für die Aufgaben der rechnerunterstützten Arbeitsplanung. Es sind Aufgaben, die mit der Auswahl von Verfahren und Betriebsmitteln sowie mit der Erstellung von Daten für die Steuerung der Betriebsmittel verbunden sind. Als Basis für die Tätigkeiten dienen die Ergebnisse der konventionellen oder rechnerunterstützten Konstruktion. Die Bereiche der rechnerunterstützten Arbeitsplanung sind: Montageplanung, Arbeitsplanerstellung, Vorrichtungs- und Sonderwerkzeug-Konstruktion, NC-Programmierung und Prüfplanung.

Mit dem Begriff **Computer Aided Manufacturing (CAM)** wird die Unterstützung der Steuerung und Überwachung der Betriebsmittel bezeichnet. Damit werden alle durch Rechnereinsatz automatisierten Fertigungsprozesse zusammengefaßt. Es sind u.a.: Fertigung von Werkstücken, Transport, Lagerung und Handhabung von Material und Fertigungshilfsmitteln.

Der Begriff **Computer Aided Quality Assurance (CAQ)** bezeichnet alle rechnerunterstützten Funktionen, die bei der Qualitätssicherung ausgeführt werden. Sie umfassen die Qualitätsplanung, Qualitätsprüfung und Qualitätslenkung. Mit Hilfe von CAQ kann die Erstellung von Prüfplänen, Prüfprogrammen und Kontrollwerten vollgezogen werden. Auch eine ständige Überwachung der Prozesse, die eine Korrektur der auftretenden Abweichungen ermöglicht, kann als Aufgabe von CAQ gesehen werden.

Mit Hilfe von **PPS**-Systemen (**Produktionsplanung und -steuerung**) werden vor allem die betriebswirtschaftliche Aufgaben realisiert. Darunter fallen alle administrative Tätigkeiten, die mit der Organisation und Überwachung des Fertigungsprozesses von der Angebotsbearbeitung bis zum Versand verbunden sind. Die Hauptfunktionen eines PPS-Systems sind: Produktionsprogrammplanung, Mengenplanung, Termin- und Kapazitätsplanung, Auftragsveranlassung und Auftragsüberwachung.

2.3.2 Integration der CIM-Komponenten

Es existieren viele Varianten der CIM-Definition [Cro90]. In allen Fällen ist jedoch die Aufgabe des CIM-Systems der Einsatz von rechnerunterstützen Aktivitäten in allen mit dem Fabrikbetrieb verbundenen Bereichen und die Integration der CIM-Komponenten durch Unterstützung des Informationsaustausches zwischen diesen Komponenten. Die Informationsflüsse in einer CIM-Fabrik zeigt die Abb. 20.

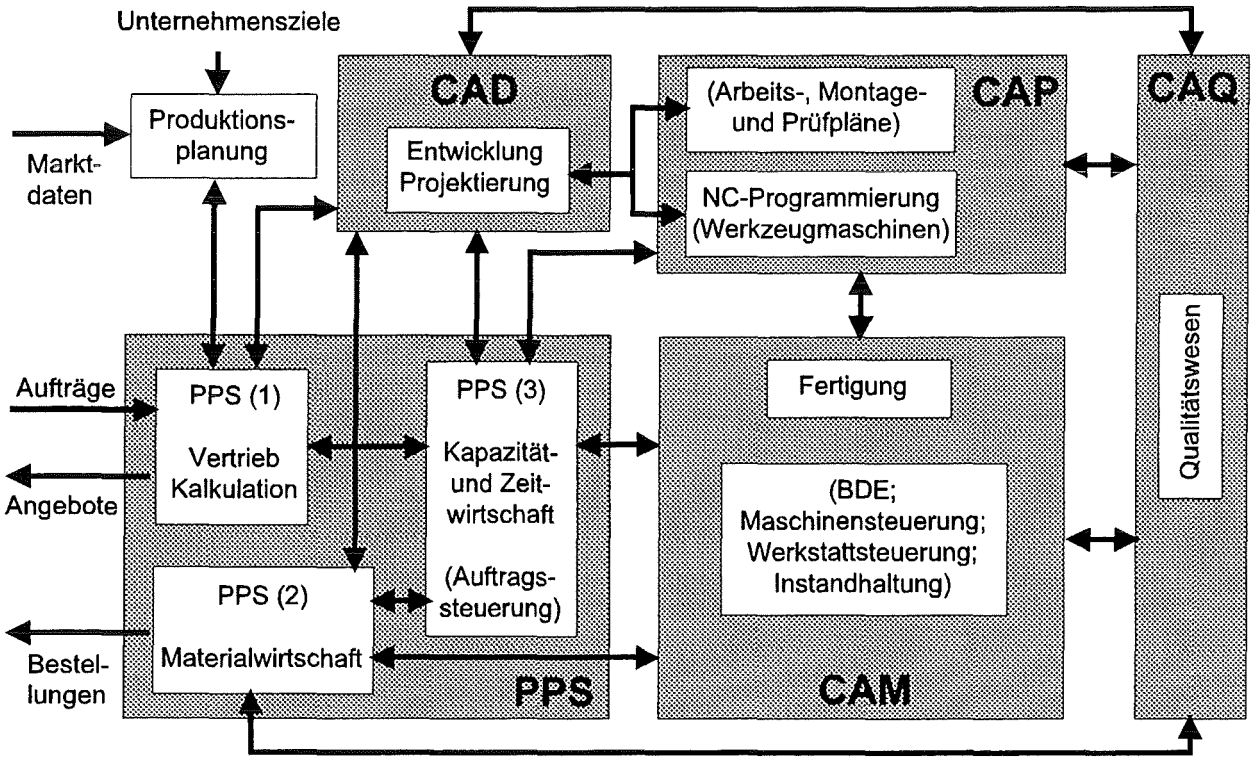


Abb. 20. Informationsfluß zwischen den CIM-Komponenten (nach FhG-ISI[4])

Die ausgetauschte Information kann zu einer von vier Kategorien gehören: Objekt-, Steuerungs-, Ergebnis- oder Anstoßinformationen. Objektinformationen sind Informationen, die in den CIM-Komponenten verarbeitet werden. Steuerungsinformationen können u.a. Absatzziel, Baupläne, Stücklisten und Arbeitspläne beschreiben. Die Istgrößen (Verbrauchsmengen, Informationen des BDE[5]-Systems) werden durch Ergebnisinformationen repräsentiert und dienen als Basisinformationen für die Kontrolle der Fertigungsprozesse. Anstoßinformationen, wie z.B.

[4] FhG-ISI steht für Frauenhofer-Gesellschaft, Institut für Systemtechnik und Innovationsforschung

[5] BDE steht für Betriebsdatenerfassung

Auftragsfreigabe oder Kostenüberschreitung, triggern entsprechende Prozesse, deren Abläufe durch die Steuerungsinformationen gelenkt werden.

Die wichtigsten **Integrationswerkzeuge** für die Unterstützung des Informationsaustausches zwischen einzelnen CIM-Komponenten sind lokale Netzwerke, Datenschnittstellen und Datenbanken [Cro90]. Mit diesen Werkzeugen lassen sich alle notwendige Schnittstellen erstellen, die eine reibungslose Kooperation der CIM-Komponenten gewährleisten.

Durch eine physikalische Verbindung zwischen verschiedenen CIM-Komponenten schaffen die **lokalen Netzwerke (Local Area Network, LAN)** die erste Voraussetzung für den Informationsaustausch innerhalb eines CIM-Systems. Verschiedene Netzwerk-Konfigurationen werden eingesetzt. Die Kopplung zwischen Komponenten kann z.B. durch den Einsatz von dezentralen Kleinrechnern, oder aber auch einen zentralen Großrechner realisiert werden. Die Entwicklung geht jedoch in die Richtung dezentraler Verbundsysteme, in denen verschiedene kleinere Systeme über lokale Netzwerke miteinander verbunden werden. Um die Integration verschiedener Systemkomponenten unterschiedlicher Hersteller zu erreichen, müssen bestimmte Regeln (z.B. Kommunikationsprotokolle) für den Informationsaustausch definiert werden. In diesem Bereich ist der Stand der internationalen Normung weit fortgeschritten. Zu erwähnen sind hier: TOP (Technical Office Protocol) für Büroanwendungen, MAP (Manufacturing Automation Protocol) und CNMA (Communications Network for Manufacturing Application) - beide für Anwendungen im Fertigungsbereich - sowie ISDN (Integrated Services Digital Network) für Kommunikation innerhalb und außerhalb eines Unternehmens.

Die mit den lokalen Netzwerken übertragene Information muß für die verschiedenen CIM-Komponenten lesbar sein. Dafür sorgen entsprechende **Datenschnittstellen**, welche die Form der ausgetauschten Information festlegen. Zahlreiche Gremien und Forschungsprojekte beschäftigen sich mit dieser Problematik. Die bisherige Ergebnisse der Arbeit sind mehrere genormte Schnittstellen, die den Datenaustausch zwischen unterschiedlichen CIM-Komponenten ermöglichen. Die meisten Entwicklungen konzentrieren sich jedoch im CAD-Bereich. Hier sind vor allem die Übertragungsformate zum Austausch von Modellen und Produktdaten zu erwähnen, z.B.: IGES (Initial Graphic Exchange Specification), VDAFS (VDA[6]-Flächenschnittstelle), SET

[6] VDA steht für Verband der deutschen Automobilindustrie

(Standard d'Exchange et de Transfer) sowie STEP (Standard for the Exchange of Product Model Data) mit der Datenbeschreibungssprache EXPRESS.

Für die vollständige Integration eines CIM-Systems muß neben der Übertragung der ausgetauschten Information auch eine entsprechende Verwaltung für die großen Datenmengen vorgesehen werden. Das ist die Aufgabe von **Datenbanken**, welche die Datenintegration und -verwaltung für die ganze CIM-Fabrik schaffen. Alle Daten aus verschiedenen Systemkomponenten werden in einer gemeinsamen Datenbasis gespeichert und den verschiedenen Anwendungen zur Verfügung gestellt. Dadurch lassen sich viele Nachteile der getrennten Abteilungsdatenbanken, wie z.B. Redundanz der Daten und Konsistenzprobleme, vermeiden.

2.3.3 Integrierende Infrastruktur des CIMOSA-Konzeptes

Einen ganzheitlichen und offenen Ansatz für die CIM-Integration bietet **das CIMOSA-Konzept**. In diesem Konzept wird außer dem Modellierungsgerüst (siehe Kapitel 2.2.2.3) auch die sogenannte **Integrierende Infrastruktur (IIS)** definiert, die sowohl für die Ausführung der mit dem Modellierungsgerüst erstellten Unternehmensmodelle als auch für die Integration von heterogenen Informations-Technologien eines Unternehmens zuständig ist. Die Offenheit des Konzeptes soll durch die Definition genormter Schnittstellen erreicht werden [AMICE93].

Um die erwähnte Aufgaben zu erfüllen, werden innerhalb der IIS sogenannte **IIS-Services** definiert, die in fünf Gruppen eingeteilt werden: Businnes Services, Presentation Services, Information Services, Common Services und System Management Services.

Die **Business Services** stellen die Funktionen zur Verfügung, die für die Ausführung eines Unternehmensmodells hinsichtlich der Funktions-Sicht und Ressourcen-Sicht notwendig sind. Die wichtigsten Aufgaben dieser Services sind: Steuerung der Ausführung von Unternehmensmodellen, sowie Verwaltung von Betriebsmitteln und Bereitstellung von Informationen über den Status von Unternehmensaktivitäten und Betriebsmitteln.

Die **Information Services** stellen alle allgemeinen Funktionen zur Verfügung, die den Zugriff zu Daten, die Integration der Daten und die Manipulation von Daten ermöglichen. Diese Services ermöglichen damit die Integration und Benutzung vorhandener Datenbanken. Sie haben den Zweck, eine flexible Verwaltung und Aktualisierung der Informationen innerhalb des Unternehmens zu unterstützen und die Konsistenz und Integrität des Informationssystems zu gewährleisten.

Die **Presentation Service** beinhalten die Funktionen, die zur Kontrolle der Ausführung von Unternehmensaktivitäten durch heterogene Ressourcen (Maschinen, Anwendungen und Personal) erforderlich sind. Zusätzlich stellen die Services aktuelle Statusinformationen über ausgeführten Aktivitäten und die mit ihnen assoziierten Ressourcen zur Verfügung. Sie sind auch für die Verwaltung und Konfiguration der Ressourcen zuständig. Diese Services bilden eine Schnittstelle der Integrierenden Infrastruktur zu der externen Welt.

Die **Common Services** wurden als Unterstützung für die übrigen Diensten im Bereich der Kommunikation definiert. Sie sollen eine transparente, netzwerkunabhängige Kommunikation zwischen verteilten IIS-Services ermöglichen. Die Common Services nehmen an der Ausführung von CIMOSA-Modellen nicht teil und werden durch die Ergebnisse der Modellierung nicht beeinflußt. Die Aufgaben dieser Services sind u.a. Austausch von Informationen innerhalb der IIS und Verwaltung von Nachrichtenwarteschlangen für jeden der IIS-Services.

Die Funktionalität der **System Management Services** ist noch nicht detailliert spezifiziert. Es wurde nur eine Liste der Grundanforderungen der Funktionalität dieser Services zusammengestellt. Insbesondere sollen die System Management Services das Verteilen, Installieren und Konfigurieren von neuen Versionen der IIS-Services ermöglichen.

Auch die Common Services sind noch nicht vollständig spezifiziert. Man muß aber betonen, daß für die Ausführung der Modelle und für die Integration von heterogenen Betriebsmitteln die volle Funktionalität der Dienste (je nach Einsatzbereich) nicht immer erforderlich ist. Das ist ein großer Vorteil, da die Zielkonfiguration der Integrierenden Infrastruktur, zumindest in einer Testumgebung, mit mehreren Näherungsschritten in einem Iterationsprozeß erreicht werden kann.

Die Spezifikation der Integrierenden Infrastruktur konzentriert sich vor allem auf die Gruppierung der Funktionen in die einzelnen Bereiche (durch Aufteilung der IIS in die verschiedenen Dienste) und auf die Zusammenarbeit dieser Dienste. Die Spezifikation erfolgt durch die Festlegung der Aufgaben und durch die Definition der Schnittstellen, welche die Erfüllung der Aufgaben gewährleisten sollen. Für die Schnittstellen werden verschiedene Protokolle definiert, welche die Funktionen eines Dienstes anderen Diensten zur Verfügung stellen. CIMOSA befaßt sich nicht mit der Frage, wie die Ausführung dieser Funktionen implementiert werden soll. Dieses Problem soll bei der Entwicklung der IIS-Dienste durch Softwareentwickler gelöst werden.

2.3.4 Zusammenfassung

Die wichtigste Aufgabe bei der Realisierung eines CIM-Systems ist die Einführung von CIM-Komponenten, die den Einsatz von rechnerunterstützten Aktivitäten ermöglichen, und die Integration der Komponenten, die als Unterstützung des Informationsaustausches und der Kooperation der Komponenten verstanden wird. Die Integration kann durch Realisierung von spezifischen Schnittstellen zwischen verschiedenen CIM-Bausteinen erreicht werden. Dieser Ansatz hat aber den Nachteil, daß das Hinzufügen von neuen Komponenten einen Einsatz von mehreren Schnittstellen zu den übrigen Komponenten bedingt.

Andere Lösung bietet das CIMOSA-Konzept, bei dem der Informationsaustausch und die Kooperation zwischen den CIM-Komponenten durch eine Integrierende Infrastruktur unterstützt wird. Die Integration einer neuen Komponente bedingt den Einsatz von nur einer Schnittstelle zu der Integrierenden Infrastruktur. Die Definition der entsprechenden Schnittstellen ist im CIMOSA enthalten und soll in der nächsten Zukunft genormt werden. Damit können die CIM-Komponenten von den Herstellern mit entsprechenden Schnittstellen ausgestattet und dadurch problemlos an die vorhandene Infrastruktur angekoppelt werden. Diese Eigenschaft ist der Grund dafür, daß die Integrierende Infrastruktur des CIMOSA-Konzeptes als "offen" bezeichnet wird.

Das CIMOSA-Konzept befindet sich noch in der Entwicklungsphase. Parallel zu der Entwicklung werden jedoch Arbeiten im Rahmen des ESPRIT-Projektes VOICE geführt, welche u.a. die Validierung der Integrierenden Infrastruktur durch die Entwicklung von Prototypen der entsprechenden Dienste als Ziel haben. Die Analyse der verfügbaren Tools [DiN91] hat gezeigt, daß für die Entwicklung und Validierung der noch nicht vollständig definierten Integrierenden Infrastruktur vor allem Petri-Netz-Werkzeuge geeignet sind. Mit Hilfe dieser Werkzeuge lassen sich schnell, trotz lückenhafter Spezifikation, entsprechende Prototypen entwickeln und validieren. Die durch die Vervollständigung der Spezifikation bedingte Änderungen und Anpassungen der Funktionalität der Prototypen können damit in einfacher Weise durchgeführt werden.

Für die Entwicklung von Prototypen der Dienste der Integrierenden Infrastruktur wird das Petri-Netz-Tool PACE verwendet [PACE93], das ein Bestandteil des McCIM-Testbettes ist [Bog92]. Die Entwicklung der Prototypen hat aber gezeigt, daß mit diesem Tool nicht alle Anpassungen der Funktionalität der Prototypen an sich ändernde Spezifikationen möglich sind. Bestimmte Erweiterungen der Prototypen bedingen Änderungen der Funktionalität des Tools (z.B. neue Konstrukte), die aber wegen der Geschlossenheit des Tools nicht möglich sind.

3. Problemstellung und Zielsetzung

Bei der Validierung von Konzepten zur Unternehmensintegration spielt die Prototypenentwicklung für Unternehmensmodelle und Softwarekomponenten einer Integrationsplattform eine wesentliche Rolle. Mit den Prototypen läßt sich die Überprüfung der von dem Konzept definierten Modellierungsmethode und Spezifikation der Softwarekomponenten auf Eignung für den vorgesehenen Verwendungszweck bereits in den früheren Phasen der Konzeptsausarbeitung durchführen. Die sich von dieser Überprüfung ergebenden Konzeptänderungen können bei der Entwicklung der nächsten Prototypen berücksichtigt werden. Damit wird die Validierung des Konzeptes beschleunigt und die Übereinstimmung der Eigenschaften der Endversion der Modellierungsmethode bzw. der Funktionalität des End-Prototyps mit der geänderten Konzeptdefinition gewährleistet.

Bei der Unternehmensmodellierung werden Prototypen eines beispielhaften Unternehmensmodells entwickelt. Das Objekt der Validierung ist nicht das entwickelte Modell, sondern die zugrunde gelegte Modellierungsmethode, die ein Bestandteil des übergeordneten Unternehmensintegrations-Konzeptes ist. Bei der Integrationsplattform werden Prototypen von Softwarekomponenten entwickelt. Das Objekt der Validierung ist die Spezifikation der Softwarekomponenten, die auch ein Bestandteil des Unternehmensintegrations-Konzeptes ist. Die eingesetzte Entwicklungsmethode ist hier nur ein Werkzeug, mit dem die Prototypen entwickelt werden, und sie wird nicht validiert.

Die Validierung der Modellierungsmethode eines nicht vollständig definierten Unternehmensintegrations-Konzeptes bereitet Probleme, weil die Modellierungsmethode (als Bestandteil des Konzeptes) sich während des Prototypenentwicklungsprozesses ändern kann. Der Grund dafür kann die "Freigabe" einer neuen Version des Konzeptes sein. Der Validierungsprozeß kann ebenfalls Ursache der Änderung sein - die Ergebnisse der Validierung können z.B. Inkonsistenzen oder Schwachstellen des Konzeptes aufweisen, so daß Änderungen der Modellierungsmethode notwendig werden.

Für die Validierung der nicht endgültig definierten Spezifikation der Integrationsplattform sind nachträgliche Änderungen des Konzeptes nicht so schwerwiegend. Sie bedeuten lediglich Änderungen der Funktionalität des zu entwickelnden Systems. Bei bestimmten Konzeptänderungen kann sich jedoch herausstellen, daß die neu spezifizierte Funktionalität sich nicht mehr mit der bereits angewandten Entwicklungsmethode realisieren läßt. So könnten z.B.

die von der gegebenen Methode zur Verfügung gestellten Konstrukte nicht ausreichend sein, so daß entweder eine neue Methode eingesetzt oder die vorhandene Methode mit neuen Konstrukten erweitert werden muß.

In bestimmten Fällen tritt daher eine Situation auf, in der die weitere Entwicklung der Prototypen (Modelle oder Softwarekomponenten) mit einer neuen Version der bis dahin eingesetzten Methode (Modellierungs- oder Softwareentwicklungsmethode) fortgeführt werden muß. Die Erfahrungen, die bei der Validierung des CIMOSA-Konzeptes in dem ESPRIT Projekt VOICE erworben wurden, zeigen, daß dies mit verfügbaren Werkzeugen nicht möglich bzw. nur sehr schwierig zu realisieren ist. Es existieren keine Ansätze (also auch keine Werkzeuge), die die Änderungen einer Methode während der mit dieser Methode durchgeführten Entwicklung unterstützen.

Die Änderungen eines Unternehmensintegrations-Konzeptes, die keinen Einfluß auf die zur Entwicklung von Prototypen eingesetzte Methode haben, werden hier als **sekundäre Änderungen** bezeichnet. Die sekundären Änderungen betreffen nur die Spezifikation eines beispielhaften Unternehmensmodells bzw. die zu validierende Spezifikation von Softwarekomponenten einer Integrationsplattform. Die schwerwiegenden Änderungen, die Änderungen der Methode nach sich ziehen, werden als **primäre Änderungen** bezeichnet. Die primären Änderungen betreffen die zu validierende Modellierungsmethode bzw. die für die Entwicklung von Prototypen von Softwarekomponenten einer Integrationsinfrastruktur eingesetzte Entwicklungsmethode (Abb. 21).

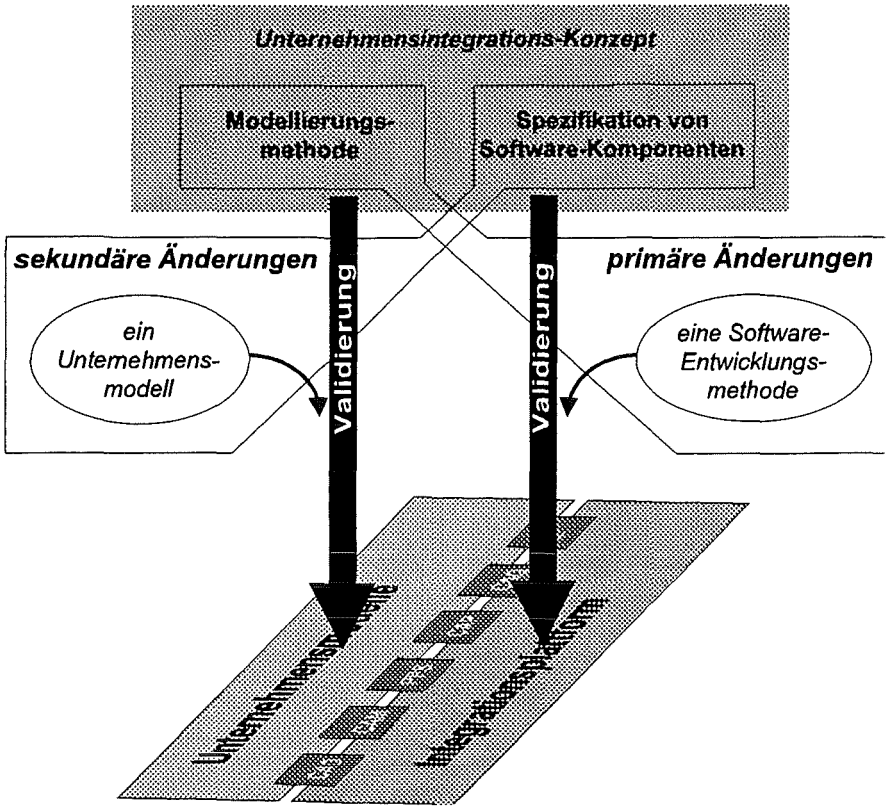


Abb. 21. Prototypenentwicklung und Konzeptänderungen

Für die Beschreibung der Entwicklung nicht vollständig definierter Systeme ist der evolutionäre Lebenszyklus am besten geeignet. Der Grund dafür ist, daß hier Prototypen aufgrund nicht vollständiger Spezifikation entwickelt werden können. Die mit den Prototypen durchgeführten Validierungstests führen zu Änderungen und Erweiterungen der Spezifikation, die dadurch in mehreren Entwicklungsschritten vervollständigt wird (siehe Kapitel 2.1.3). Somit kann der evolutionäre Lebenszyklus die sekundären Änderungen eines Konzeptes unterstützen. Die primären Änderungen werden dagegen nicht unterstützt - die Evolution betrifft nur die Funktionalität von Prototypen und nicht die eingesetzte Entwicklungsmethode.

Das Problem der fehlenden Unterstützung der Methodenänderungen kann teilweise durch den Einsatz des Meta-Modellierungs-Konzepts [VaG91] gelöst werden. Der durch die Methodenänderung eingeführte Bruch im Lebenszyklus eines Systems bleibt jedoch weiterhin bestehen: nach der Änderung bzw. Erweiterung der Entwicklungsmethode muß die bis zu diesem Zeitpunkt erstellte Spezifikation entsprechend konvertiert werden, damit sie wieder verwendet werden kann (siehe Kapitel 2.1.7).

Ziel der Arbeit ist daher die Entwicklung eines Ansatzes, der die Durchführung von sekundären und primären Änderungen eines Unternehmensintegrations-Konzeptes bei der Entwicklung von

Prototypen zur Validierung dieses Konzeptes voll unterstützt, ohne einen Bruch in den Lebenszyklus des Prototyp-Systems einzuführen. Der Begriff "System" hat in dieser Arbeit immer zwei Interpretationen: er kann sowohl Unternehmensmodelle als auch Softwarekomponenten einer Infrastruktur, die zur Integration von CIM-Komponenten dient, repräsentieren (siehe Kapitel 2). Im Rahmen der Arbeit wird ein prototypisches Werkzeug entwickelt, das diesen Ansatz demonstriert.

Als Basis für eine beispielhafte Anwendung des erarbeiteten Ansatzes wird die CIMOSA-Architektur verwendet (siehe Kapitel 2.2.2.3 und 2.3.3). Mit dem im Rahmen der Arbeit entwickelten Tool sollen zwei Prototypen implementiert werden: ein beispielhaftes, mit der CIMOSA-Referenzarchitektur konformes Unternehmensmodell und eine beispielhafte Softwarekomponente, welche die Grundfunktionalität eines Dienstes der Integrierenden Infrastruktur des CIMOSA-Konzeptes implementiert.

4. Das Konzept

Als neuer Faktor der Prototypenentwicklung zur Validierung von Unternehmensintegrationskonzepten wird in der Arbeit ein neuer, **zweidimensionaler Lebenszyklus** eines Systems vorgestellt. Der Lebenszyklus soll neben den evolutionären Funktionalitätsänderungen des Systems auch die Änderungen und Erweiterungen der zur Entwicklung dieses Systems eingesetzten Methode unterstützen.

Die Entwicklung von Prototypen des Systems kann somit nicht nur in einer Dimension erfolgen, welche nur die Funktionalitätsänderungen des Systems umfaßt. Die Entwicklung kann zusätzlich in einer anderen Dimension realisiert werden, welche die Änderungen der eingesetzten Entwicklungsmethode umfaßt. Damit können die noch nicht vollständig definierten Unternehmensintegrations-Konzepte durch Entwicklung von Prototypen in einem in sich abgeschlossenen Lebenszyklus validiert werden. Die primären Änderungen (siehe Kapitel 3), die bei solchen unvollendeten Konzepten anfallen können, führen, anders als in den übrigen Ansätzen zur Prototypenentwicklung, zu keinem Bruch in dem Lebenszyklus des zu entwickelnden Systems, weil sie von diesem Lebenszyklus unterstützt werden.

Die Idee des zweidimensionalen Lebenszyklus, dessen Realisierung und die Definitionen der dabei verwendeten Begriffe werden in den weiteren Abschnitten (4.3 und 4.4) vorgestellt. Als erstes werden jedoch in den nächsten zwei Abschnitten die Anforderungen erläutert, die von dem in dieser Arbeit präsentierten Konzept und von einem dieses Konzept unterstützenden Tool erfüllt werden sollen.

4.1 Anforderungen an das Konzept

Für die Entwicklung von Prototypen eines Systems, dessen Spezifikation nicht vollständig ist und sich im Laufe des Entwicklungsprozesses ändern kann, ist der **evolutionäre** Lebenszyklus am besten geeignet. In diesem Lebenszyklus wird nämlich eine solche unvollständige Spezifikation als Ausgangspunkt der Entwicklung des ersten Prototyps angenommen. Sie wird bei der Entwicklung der nächsten Prototypen schrittweise vervollständigt. Diese Änderungen der Spezifikation entsprechen den in dem Kapitel 3 definierten sekundären Änderungen. Der evolutionäre Lebenszyklus soll daher als Basis für den neuen zweidimensionalen Lebenszyklus dienen. Es müssen dabei entsprechende Erweiterungen der Definition des Lebenszyklus vorgenommen werden, um auch die primären Änderungen in diesem Lebenszyklus zu unterstützen.

Um einen Bruch im Lebenszyklus zu vermeiden, müssen die Änderungen der Spezifikation des zu entwickelnden Systems (sekundäre Änderungen) und die Änderungen der Entwicklungsmethode (primäre Änderungen) möglichst **unabhängig voneinander durchzuführen** sein. Die Änderungen der Entwicklungsmethode müssen demgemäß möglichst **geringe Auswirkung** auf den Verlauf des Entwicklungsprozesses haben.

Mit Sicherheit lassen sich nicht alle Änderungen der Entwicklungsmethode ganz unabhängig vom Entwicklungsprozeß durchführen. Wenn z.B. eine Methode komplett durch eine andere Methode ersetzt wird, ist die Konvertierung der bis dahin entwickelten Modelle des Systems, falls überhaupt möglich, nicht zu vermeiden. In dem Konzept werden solche extremen Fälle nicht betrachtet. Als Änderungen der Entwicklungsmethode werden daher nur die **Erweiterungen** bzw. **Modifikationen** der Methode verstanden, die die Anpassungen der Eigenschaften dieser Methode an sich ändernde Anforderungen ermöglichen. Dieses Konzept soll solche Änderungen unterstützten, wobei der Spielraum der Änderungen möglichst groß sein soll.

Sowohl für die Erstellung von Unternehmensmodellen als auch für die Entwicklung von Softwarekomponenten werden in der Praxis zahlreiche Methoden verwendet. (siehe Kapitel 2.2). Wie der Überblick der Methoden in den vorausgehenden Kapiteln zeigt, basieren die meisten Methoden auf einer graphischen Beschreibungssprache, welche erlaubt, das Spezifikations- bzw. das Entwurfsmodell eines Systems in Form eines **Graphen** darzustellen. Das Konzept soll daher in erster Linie den Einsatz von solchen graphenorientierten Entwicklungsmethoden ermöglichen.

4.2 Anforderungen an das Werkzeug

Bei vielen Änderungen sowohl der Funktionalität eines Prototyps als auch einer Entwicklungsmethode ist eine entsprechende Dokumentation der Änderungen und des neuesten Standes der Entwicklung von großer Bedeutung. Die Dokumentation (Beschreibung der Prototypen-Funktionalität und der Entwicklungsmethode) soll in einer **standardisierten Form** erstellbar sein. Dieser offene Ansatz ermöglicht eine einfache Übertragung der Funktionalitätsbeschreibung von Prototypen und der Methodenbeschreibung zwischen Werkzeugen verschiedener Hersteller.

Die Validierung einer Methode zur Unternehmensmodellierung kann durch Erstellen von verschiedenen beispielhaften Modellen erfolgen. Damit wird die Methode auf die Eignung zur Beschreibung der Unternehmensfunktionen und -daten geprüft. Die Validierung der Spezifikation der Software-Komponenten einer Integrationsplattform muß die Tests der Funktionalität der

entwickelten Prototypen beinhalten. Damit kann die Eignung der Spezifikation zur Realisierung von angenommenen Zielen überprüft werden. Diese Tests können mit Hilfe der **Simulation** durchgeführt werden. Das Werkzeug muß also die Möglichkeit bieten, die Funktionalität der entwickelten Prototypen durch Simulation zu testen.

Das Konzept läßt den Einsatz von vielen verschiedenen Entwicklungsmehoden zu. Um den Umgang mit dem Werkzeug bei der Verwendung von verschiedenen Methoden zu erleichtern, soll in dem Werkzeug eine von der verwendeten Methode unabhängige **Benutzeroberfläche** zur Verfügung gestellt werden.

4.3 Zweidimensionaler Lebenszyklus

4.3.1 Grundsätzliche Überlegungen

Wie bereits in dem Kapitel 2.1.7 erwähnt wurde, weisen die heutzutage auftretenden Lebenszyklen eine gemeinsame Eigenschaft auf: sie verwenden eine vordefinierte Methode, um sich mit der für jede Entwicklungsphase spezifischen Problematik auseinanderzusetzen. Die Entwicklung in einem Lebenszyklus verläuft immer entlang einer Achse, die durch einzelne Entwicklungsphasen (Spezifikation, Entwurf, Implementation) bestimmt wird. Die Bewegungen auf der Achse bedeuten Änderungen des Entwicklungszustandes des Systems.

Bei den Lebenszyklen, in denen die endgültige Funktionalität des Systems durch die Implementation von mehreren Prototypen erreicht wird, kann jeder Position auf der Entwicklungsachse ein Wert zugewiesen werden, der den Erfüllungsgrad der gestellten Anforderungen beschreibt. Es wird dabei der Begriff "Vollständigkeit des Systems" verwendet:

> **Die Vollständigkeit (V)** des zu entwickelnden Systems beschreibt den Erfüllungsgrad der an das System gestellten Anforderungen, den die nacheinander folgenden Prototypen des Systems aufweisen. Wenn alle Anforderungen erfüllt sind, erreicht das System die Zielvollständigkeit.

Die Vollständigkeit eines Systems läßt sich nicht quantifizieren - sie wird von dem Entwickler aufgrund der Ergebnisse der Validierung des Systems bezüglich der gestellten Anforderungen geschätzt. Der Entwickler kann feststellen, welches System mehr Anforderungen erfüllt, d.h. welches System eine größere Vollständigkeit aufweist. Wenn alle Anforderungen erfüllt sind, erreicht das System die Zielvollständigkeit.

Die Änderungen der Vollständigkeit eines System am Beispiel des evolutionären Lebenszyklus zeigt die Abb. 22.

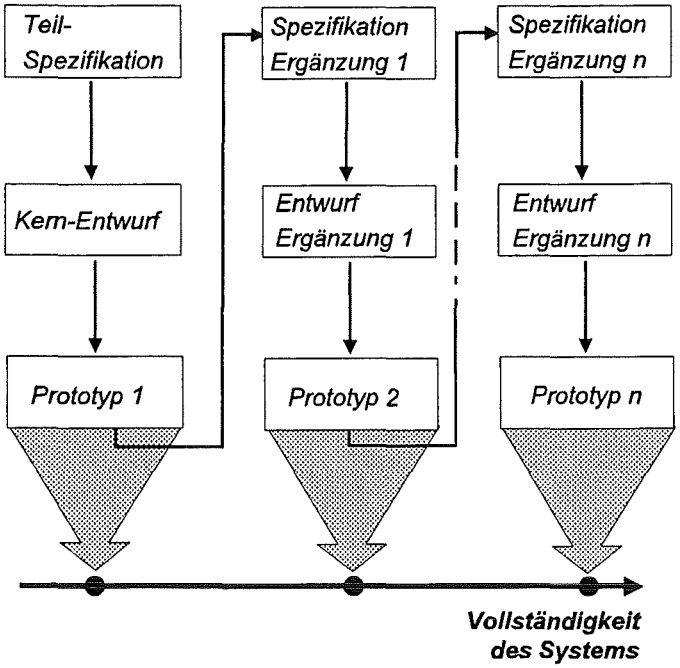


Abb. 22. Der evolutionäre Lebenszyklus: die Änderungen der Vollständigkeit des Systems

Als Ausgangspunkt der Entwicklung dient eine nicht vollständige Spezifikation des Systems, die im ersten Schritt in einen Kernentwurf umgewandelt wird. Der Kernentwurf wird für die Implementation des ersten Prototyps verwendet, dessen Vollständigkeit zwar relativ gering ist, jedoch ausreichend, um dem zukünftigen Benutzer eine Vorstellung über das System, seine Grundstruktur und -Funktionalität zu geben. In den nächsten Iterationen wird die Funktionalität schrittweise erweitert, wodurch die Vollständigkeit des zu entwickelnden Systems zunimmt. Beim letzten Prototyp erreicht das System die Zielvollständigkeit.

Die Methode, die zur Entwicklung des Systems eingesetzt wird, ändert sich in dem ganzen Lebenszyklus nicht - ihre Eigenschaften, wie z.B. Modellierungskonstrukte, Vorgehensweise bei der Entwicklung, werden vor dem Entwicklungsprozeß festgelegt und bleiben im Laufe dieses Prozesses unverändert. Es wird dabei angenommen, daß die Eigenschaften der Methode ausreichend sind, um ein System mit der Zielvollständigkeit zu entwickeln.

Wenn für die Entwicklung eines Systems eine falsche Methode ausgewählt wurde, oder wenn die Anforderungen an das System sich während der Entwicklung ändern, kann es vorkommen, daß die Eigenschaften der eingesetzten Methode nicht ausreichend sind, um die Zielvollständigkeit des Systems zu erreichen. In dem Fall werden nicht alle an das System gestellte Anforderungen erfüllt. Diese Angemessenheit der Methode bezüglich der Anforderungen wird mit dem Begriff "Mächtigkeit" beschrieben.

Die Mächtigkeit (M) einer Entwicklungsmethode beschreibt die Angemessenheit der Methode bezüglich der Anforderungen, die an das mit der Methode zu entwickelnde System gestellt werden. Die volle Mächtigkeit bedeutet, daß es mit Hilfe der Methode möglich ist, ein System mit der Zielvollständigkeit zu entwickeln. Mit einer Methode, die mit einer nicht vollen Mächtigkeit ausgezeichnet ist, läßt sich kein System mit der Zielvollständigkeit entwickeln.

Die Mächtigkeit einer Methode ist immer von dem zu entwickelnden System abhängig - eine Methode, welche bei der Entwicklung eines Systems die Erfüllung von allen Anforderungen gewährleistet (volle Mächtigkeit), muß nicht die Zielvollständigkeit bei der Entwicklung eines anderen Systems gewährleisten. Da die Mächtigkeit einer Methode auf der Basis der Vollständigkeit des zu entwickelnden Systems definiert wird, ist die Quantifizierung dieser Größe ebenfalls nicht möglich. Ähnlich wie bei der Vollständigkeit kann aber (während des Entwicklungsprozesses) festgestellt werden, welche Methode eine größere Mächtigkeit bezüglich der gestellten Anforderungen aufweist und welche Methode die Zielvollständigkeit des zu entwickelnden Systems gewährleisten kann (volle Mächtigkeit).

Wie bereits im Kapitel 2.1.7 erwähnt wurde, führt die Feststellung, daß die zur Entwicklung eines Systems eingesetzte Methode nicht die volle Mächtigkeit hat, zwangsläufig zum Einsatz einer anderen Methode, welche die Implementation eines Systems mit der Zielvollständigkeit ermöglichen soll. Dabei muß die bis jetzt entwickelte Systembeschreibung (Spezifikation bzw. Entwurf) entsprechend konvertiert werden, was zu einem Bruch im Verlauf des Entwicklungsprozesses führt. Diese Konvertierung ist jedoch nicht immer möglich, da die entsprechenden Schnittstellen zwischen Methoden bzw. zwischen Werkzeugen, die diese Methoden unterstützen, fehlen.

Das Problem der fehlenden Schnittstellen kann durch den Einsatz des Meta-Modellierungskonzepts gelöst werden, in dem verschiedene Methoden immer von einer Meta-Methode abgeleitet werden. Dies macht es möglich, die bereits erstellte Systembeschreibung beim Einsatz einer neuen Version der Methode weiter zu verwenden. Die Konvertierung der Beschreibungen bleibt aber wie vorher aufwendig und zeitraubend, wodurch sich der Entwicklungsprozeß deutlich verlangsamt. Eine weitere Schwachstelle liegt darin, daß das Konzept für die Entwicklung von abgeschlossenen Werkzeugen eingesetzt wird, wodurch der Systementwickler, der ein solches Werkzeug einsetzt, keine Möglichkeit hat, die von diesem Werkzeug unterstützte Methode zu ändern und an die neuen Anforderungen anzupassen. Mit anderen Worten, bei dem Meta-Konzept sind die Änderungen der Methode während des Entwicklungsprozesses nicht möglich bzw. sehr schwer zu realisieren. Die Methode wird vor dem Entwicklungsprozess vollständig definiert und es werden keine späteren Änderungen dieser

Methode vorgesehen. Der Verlauf der Methoden- und Systementwicklung sieht in diesem Fall wie in der Abb. 23 aus.

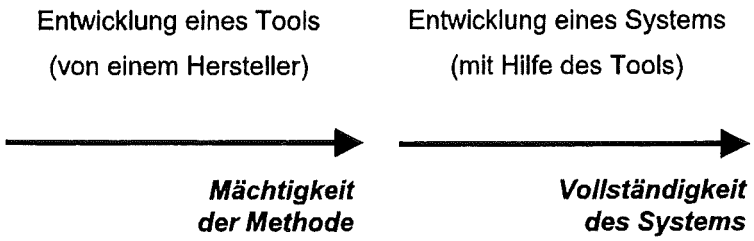


Abb. 23. Das Meta-Modellierungskonzept: die Methoden- und Systementwicklung

4.3.2 Entwicklungsraum und -pfad

Der in dieser Arbeit vorgeschlagene zweidimensionale Lebenszyklus umgeht die in den vorausgehenden Abschnitten dargestellten Probleme der Prototypenentwicklung, indem die Zunahme der Mächtigkeit einer Entwicklungsmethode parallel zu der mit der Methode realisierten Entwicklung eines Systems zugelassen wird. Der Entwicklungsprozeß erfolgt damit nicht mehr auf der Achse der Vollständigkeit des Systems, sondern in einem **zweidimensionalen Raum**, der durch die zwei Achsen der Mächtigkeit der Methode und der Vollständigkeit des Systems bestimmt wird (Abb. 24).

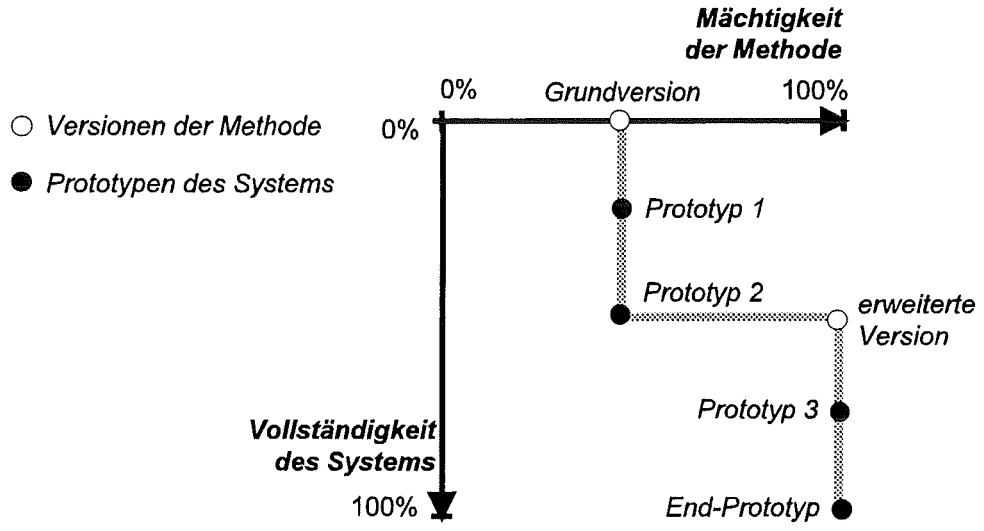


Abb. 24. Zweidimensionaler Lebenszyklus und ein exemplarischer Entwicklungspfad

Der zweidimensionale Raum wird als **Entwicklungsraum** bezeichnet. Die in dem Diagramm gezeigte Kurve wird **Entwicklungspfad** genannt. Ein Entwicklungspfad beschreibt daher den Verlauf des Entwicklungsprozesses im Entwicklungsraum. Der Entwicklungspfad wird durch die

Positionen der nacheinander folgenden Prototypen und der Versionen der Methode im Entwicklungsraum bestimmt.

Der zweidimensionale Lebenszyklus basiert auf dem evolutionären Modell der Entwicklung. Das bedeutet, daß ein Segment zwischen den nacheinander folgenden Prototypen (ein zu der Vollständigkeits-Achse paralleler Abschnitt) einem Iterationsschritt des evolutionären Lebenszyklus entspricht und daher die Erweiterung bzw. Modifikation der Funktionalität des Prototyps bedeutet. Ein zu der Mächtigkeits-Achse paralleles Segment, das die Änderung der Methode darstellt, hat dagegen keine Abbildung im evolutionären Lebenszyklus.

Der in der Abb. 24 präsentierte Entwicklungspfad stellt nur ein Beispiel dar. Der Verlauf dieses beispielhaften Pfades zeigt, daß die für die Entwicklung der ersten zwei Prototypen eingesetzte Methode eine unzureichende Mächtigkeit aufweist, um ein vollständiges System zu entwickeln. Aus diesem Grund wird nach der Entwicklung des zweiten Prototyps die bisher eingesetzte Methode erweitert, so daß sie die volle Mächtigkeit erreicht. Mit der erweiterten Methode läßt sich nun das Zielsystem vollständig entwickeln (die nächsten zwei Schritte). Andere mögliche Varianten des Entwicklungspfades werden in dem nächsten Kapitel vorgestellt.

4.3.3 Varianten des Entwicklungspfades

Der in dem vorherigen Kapitel präsentierte Entwicklungspfad stellt nur eine von mehreren möglichen Varianten dar. Diese Variante entspricht der Grundsituation, in der dem Systementwickler ein Werkzeug mit einer vordefinierten Methode zur Verfügung gestellt wird. Die Methode wird aber im Verlauf des Entwicklungsprozesses erweitert, um die angestrebte Funktionalität des Systems erreichen zu können. Die Mächtigkeit der zur Verfügung gestellten Methode war für den bestimmten Zweck nicht ausreichend und sie wurde daher an die speziellen Anforderungen durch entsprechende Erweiterungen angepaßt. Wie in den vorausgehenden Abschnitten erläutert wurde, wäre dies mit Werkzeugen, die andere (d.h. eindimensionale) Lebenszyklen unterstützten, nur sehr schwer durchzuführen. Das hier präsentierte Konzept des zweidimensionalen Lebenszyklus ist grundsätzlich für den Einsatz in solchen Situationen vorgesehen. Das Konzept kann aber auch in anderen Situationen eingesetzt werden, die im weiteren vorgestellt werden.

Wenn keine vordefinierte Methode zu Beginn des Entwicklungsprozesses zur Verfügung gestellt wird, muß sie zuerst entwickelt werden. Es muß dabei nicht die volle Mächtigkeit der Methode in

einem Schritt erreicht werden - die Entwicklung der Methode und der Prototypen des zu entwickelnden Systems kann verflochten sein (Abb. 25).

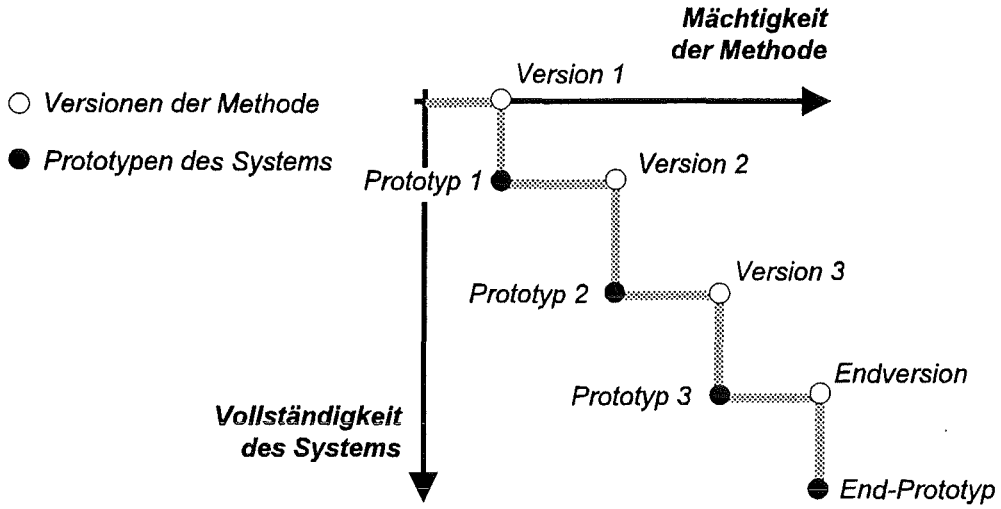


Abb. 25. Die Verflechtung der Entwicklung der Methode und der Prototypen

Der in dem Diagramm gezeigte "Treppen"-Entwicklungspfad entspricht der Situation, in der zuerst eine einfache Version der Methode und anschließend der erste Prototyp entwickelt wird. Im nächsten Schritt wird die Methode erweitert, so daß die Entwicklung des nächsten Prototyps möglich ist. In jedem Schritt erreicht der Prototyp die größte Vollständigkeit, die mit der aktuellen Version der Methode erzielt werden kann. Vor der Entwicklung des nächsten Prototyps muß deswegen die Methode entsprechend erweitert werden.

Unter vielen anderen Varianten des Entwicklungspfades sind noch zwei Sonderfälle zu erwähnen, in denen der Entwicklungsraum sich zu einem eindimensionalen Raum reduziert. Der erste Fall tritt auf, wenn die vorgegebene Methode bereits die volle Mächtigkeit aufweist. Damit sind jegliche Methodenänderungen überflüssig - die Zielvollständigkeit des Systems kann mit der vorgegebenen Methode erreicht werden. Der zweidimensionale Lebenszyklus reduziert sich in diesem Fall zum normalen evolutionären Lebenszyklus und die Entwicklung des Systems erfolgt ähnlich wie in den traditionellen CASE-Tools (Abb. 26).

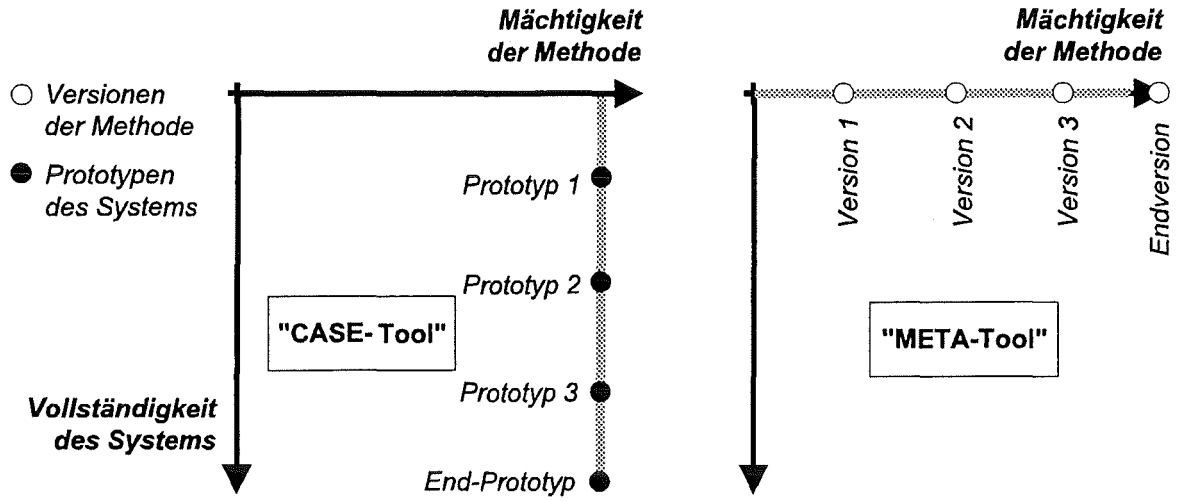


Abb. 26. Sonderfälle des Entwicklungspfades: "CASE-Tool" und "Meta-Tool"

Der zweite Sonderfall beschreibt die Situation, in der das Ziel der Entwicklung nicht ein System sondern eine Entwicklungsmethode ist. Dieser Fall entspricht dem Meta-Modellierungskonzept, das in verschiedenen Meta-Tools eingesetzt wird.

4.4 Bausteine

4.4.1 Grundsätzliche Überlegungen

Zur Unterstützung des zweidimensionalen Lebenszyklus wird ein Baustein-Konzept eingesetzt. In diesem Konzept wird ein System durch Erzeugen und Zusammensetzen von Bausteinen (in einem Modell) entwickelt. Die Definition einer Methode, die bei dieser Entwicklung eingesetzt wird, erfolgt durch die Definition verschiedener Baustein-Typen. Bei der Entwicklung eines Systems werden dabei Bausteine von den in dieser Methode definierten Baustein-Typen verwendet. Die Verwendung des Begriffs Typ ist jedoch in diesem Konzept nicht adäquat. Nach [Sch91] ist ein Typ "eine benannte Menge, deren Elemente eine gemeinsame Eigenschaft besitzen. Die Definition eines Typs kann durch eine Beschreibung dieser typbildenden Eigenschaften erfolgen...". In diesem Konzept werden die gemeinsamen Eigenschaften der Bausteine jedoch nicht nur in den erwähnten Baustein-Typen definiert. Bestimmte gemeinsame Eigenschaften werden auch in sogenannten Baustein-Templates festgelegt. Um eine eindeutige Benenung innerhalb dieser Arbeit zu ermöglichen, werden daher andere Begriffe verwendet. Anstelle des Baustein-Typs und des Bausteines werden die Begriffe Baustein-Instanz bzw. Baustein-Occurrence verwendet. Eine Menge von Baustein-Instanzen definiert damit eine Methode und eine Menge von Baustein-Occurrences ein System.

Sowohl für die Definition der Baustein-Instanzen als auch für das Erzeugen der Baustein-Occurrences werden die bereits erwähnten Baustein-Templates verwendet. Ein Baustein-Template ist eine Schablone (ein Formular) mit zahlreichen Attributen, die als Platzhalter für verschiedene Informationen dienen. Die Definition einer Baustein-Instanz und das Erzeugen einer Baustein-Occurrence ist gleichbedeutend mit dem Erstellen einer Kopie des entsprechenden Baustein-Templates und dem anschließenden Ausfüllen dieser Kopie, was durch Definition verschiedener Informationen in den von den Templates zur Verfügung gestellten Attributen (Platzhaltern) erfolgt. Das Kopieren und Ausfüllen der Baustein-Templates wird auch als Ableiten einer Baustein-Instanz bzw. einer Baustein-Occurrence bezeichnet. Die Ableitung der Baustein-Instanzen wird in der Abb. 27 dargestellt.

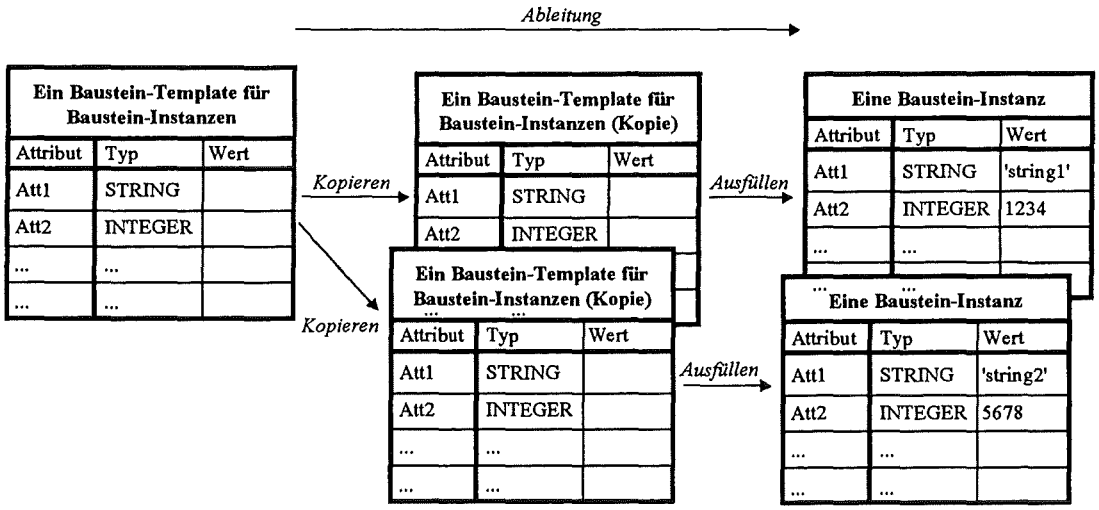


Abb. 27. Ableiten der Baustein-Instanzen[7]

Die Ableitung der Baustein-Occurrences erfolgt nach dem gleichen Schema. Die Informationen, die bei der Definition einer Baustein-Instanz und beim Erzeugen einer Baustein-Occurrence angegeben werden müssen, sind jedoch unterschiedlich. Bei der Baustein-Instanz werden die allgemeinen Eigenschaften der entsprechenden Baustein-Occurrences definiert, etwa:

- die Darstellung der Baustein-Occurrences,
- die Benutzerschnittstelle (z.B. Menüs) die von den Baustein-Occurrences zur Verfügung gestellt wird,
- die Verbindungsregeln, die bestimmen, mit welchen anderen Baustein-Occurrences eine Baustein-Occurrence Verbindungen haben kann,
- das Verhalten der Baustein-Occurrences,
- die Definition der Daten, die in den Baustein-Occurrences gespeichert werden können.

Bei der Baustein-Occurrence werden dagegen die Informationen angegeben, die für jede Baustein-Occurrence spezifisch sind, etwa:

- die existierenden Verbindungen mit anderen Baustein-Occurrences,
- die topologische Information (d.h. die räumlichen Koordinaten der Baustein-Occurrence),
- die konkreten Daten, die in der Baustein-Occurrence gespeichert sind.

Die Tatsache, daß die für die Baustein-Instanzen und die Baustein-Occurrences benötigten Informationen unterschiedlich sind, hat zur Folge, daß für die beiden Konstrukte unterschiedliche

[7] Für die Definition der Typen in dem Baustein-Template wird die EXPRESS-Notation verwendet

Templates verwendet werden müssen. Die Templates für die Baustein-Instanzen werden **als statische Baustein-Templates** bezeichnet, weil sie unveränderlich sind. Die Templates für die Baustein-Occurrences werden dagegen als **dynamische Baustein-Templates** bezeichnet, weil ihre Struktur zum Teil von der bei der Definition der entsprechenden Baustein-Instanz angegeben Informationen abhängig ist und sich daher ändern kann. Bei diesen Änderungen der Struktur der dynamischen Baustein-Templates handelt es sich vor allem um die Änderungen der Attribute für die Daten, die in einer Baustein-Occurrence gespeichert werden (siehe Kapitel 5.2).

Es werden drei statische Baustein-Templates vordefiniert: **statisches Component-Template** ($t_{s,C}$), **statisches Node-Template** ($t_{s,N}$) und **statisches Arc-Template** ($t_{s,A}$). Die Definition dieser Templates in EXPRESS-G wird im Kapitel 5.4 vorgestellt. Von diesen drei statischen Baustein-Templates werden verschiedene Baustein-Instanzen abgeleitet. Es gilt dabei:

- Vom statischen Component-Template wird nur eine Baustein-Instanz abgeleitet (Component-Instanz).
- Vom statischen Node- oder Arc-Template können mehrere Baustein-Instanzen (Node-Instanzen bzw. Arc-Instanzen) abgeleitet werden

Die abgeleiteten Baustein-Instanzen, welche die ausgefüllten statischen Baustein-Templates darstellen, bestimmen die Methode, die zur Entwicklung eines System-Modells eingesetzt wird.

Eine **Methode** m wird mit folgendem Tupel definiert:

$$m = (c_I, N_I, A_I).$$

Es gilt dabei:

1. a) c_I ist ein ausgefülltes statisches Component-Template $t_{s,C}$.
   b) c_I wird die **Component-Instanz** der Methode m genannt.
2. a) N_I ist eine Menge der ausgefüllten statischen Node-Templates $t_{s,N}$.
   b) $n_I \in N_I$ wird eine **Node-Instanz** der Methode m genannt
3. a) A_I ist eine Menge der ausgefüllten statischen Arc-Templates $t_{s,A}$.
   b) $a_I \in A_I$ wird eine **Arc-Instanz** der Methode m genannt
4. $I = \{c_I\} \cup N_I \cup A_I$ ist eine Menge der **Baustein-Instanzen** der Methode m

Die Information, die bei der Ableitung der Baustein-Instanzen spezifiziert wurde, bestimmt die gemeinsamen Eigenschaften der Baustein-Occurrences, die bei der Entwicklung eines System-Modells verwendet werden. Diese in allen Baustein-Instanzen definierten Eigenschaften bestimmen die Vorgehensweise und die Beschreibungssprache, die bei der Entwicklung eingesetzt wird - sie definieren daher die Methode (vergleiche die Definition einer Methode im Kapitel 2).

Die Baustein-Instanzen dienen als Vorlage für das Erzeugen von Baustein-Occurrences während der Entwicklung eines System-Modells. Die Baustein-Occurrences werden jedoch nicht direkt von den entsprechenden Baustein-Instanzen abgeleitet, sondern von den dynamischen Baustein-Templates. Wie bereits erwähnt wurde, ist die Struktur der dynamischen Baustein-Templates von der in den Baustein-Instanzen definierten Information abhängig und kann daher nicht vordefiniert werden. Es kann nur eine Rahmendefinition dieser Baustein-Templates vorgegeben werden. Die Rahmendefinition kann als ein fast vollständig definiertes dynamisches Baustein-Template gesehen werden (Rahmentemplate), in dem nur bestimmte Attribute (je nach Definition der entsprechenden Baustein-Instanz) hinzugefügt werden müssen. Eine ausführliche Beschreibung befindet sich im Kapitel 5.5.3.

Für jede Baustein-Instanz wird ein dynamisches Baustein-Template generiert. Die Zahl der dynamischen Baustein-Templates entspricht daher der Zahl der Baustein-Instanzen. Die Generierung eines dynamischen Baustein-Templates ist gleichbedeutend mit dem Erstellen einer Kopie des vorgegebenen Rahmentemplates und anschließendem Ergänzen mit zusätzlichen Attributen (falls notwendig). Wenn die Menge der dynamischen Baustein-Templates, die bei der Definition der Methode m (Definition der Baustein-Instanzen) generiert wurden, als T_d bezeichnet wird, kann die Funktion zur Generierung dieser Baustein-Templates folgendermaßen beschrieben werden:

Eine symmetrische Funktion

$$\gamma : I \rightarrow T_d$$

wird die **Generierungsfunktion** der dynamischen Baustein-Templates genannt. $\gamma(i)$ mit $i \in I$ bezeichnet dabei ein dynamisches Baustein-Template, das aus der Instanz i generiert wird.

Es gilt dabei:

1. $t_{d,C} = \gamma(c_I)$ wird das **dynamische Component-Template** der Methode m genannt
2. a) $T_{d,N} = \{\gamma(i) \mid i \in N_I\}$ ist eine Menge der dynamischen Baustein-Templates, die für die Node-Instanzen der Methode m generiert werden.
   b) $t_{d,N} \in T_{d,N}$ wird ein **dynamisches Node-Template** der Methode m genannt.
3. a) $T_{d,A} = \{\gamma(i) \mid i \in A_I\}$ ist eine Menge der dynamischen Baustein-Templates, die für die Arc-Instanzen der Methode m generiert werden.
   b) $t_{d,A} \in T_{d,A}$ wird ein **dynamisches Arc-Template** der Methode m genannt.
4. $T_d = \{t_{d,C}\} \cup T_{d,N} \cup T_{d,A}$

Von den generierten dynamischen Baustein-Templates werden bei der Entwicklung eines System-Modells die Baustein-Occurrences abgeleitet. Es gilt dabei:

- Vom dynamischen Component-Template wird nur eine Baustein-Occurrence abgeleitet (Component-Occurrence). Die Component-Occurrence repräsentiert einen Wurzelknoten des Graphen, der das zu entwickelnde System beschreibt.
- Von jeden dynamischen Node- oder Arc-Template können mehrere Baustein-Occurrences (Node-Occurrences bzw. Arc-Occurrences) abgeleitet werden. Die Node- und Arc-Occurrences stellen die Knoten bzw. Kanten eines Graphen dar, der das zu entwickelnde System beschreibt.

Alle Baustein-Occurrences bilden ein System-Modell:

Ein mit Hilfe der Methode m entwickeltes **System-Modell** s_m ist ein Tupel

$$s_m = (c_{O,s}, N_{O,s}, A_{O,s}).$$

Es gilt dabei:

1. a) $c_{O,s}$ ist ein ausgefülltes dynamisches Component-Template der Methode m.
   b) $c_{O,s}$ wird die Component-Occurrence in dem System-Modell s_m bzw. die **Komponente** des System-Modells s_m genannt.
2. a) $N_{O,s}$ ist eine Menge der ausgefüllten dynamischen Node-Templates der Methode m.
   b) $n_{O,s} \in N_{O,s}$ wird eine Node-Occurrence in dem System-Modell s_m bzw. ein **Knoten** des System-Modells s_m genannt.
3. a) $A_{O,s}$ ist eine Menge der ausgefüllten dynamischen Arc-Templates der Methode m.
   b) $a_{O,s} \in A_{O,s}$ wird eine Arc-Occurrence in dem System-Modell s_m bzw. eine **Kante** des System-Modells s_m genannt.
4. $O_s = \{c_{O,s}\} \cup N_{O,s} \cup A_{O,s}$ ist eine Menge der **Baustein-Occurrences** in dem System-Modell s_m.

4.4.2 Zusammenhänge zwischen Begriffen

Die Zusammenhänge zwischen den eingeführten Begriffen werden mit Hilfe der binären ER-Methode [ScN90] in der Abb. 28 dargestellt.

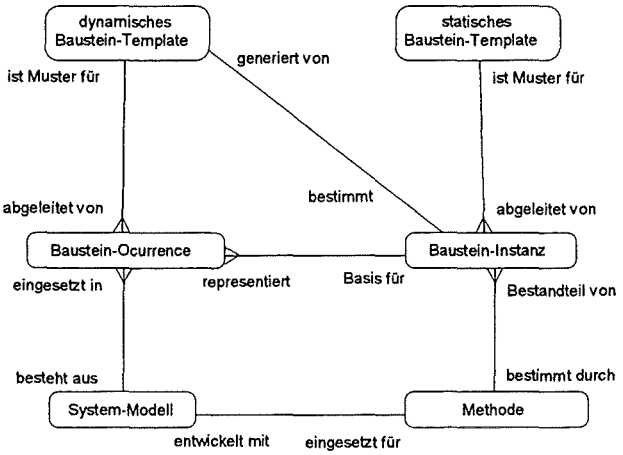


Abb. 28. Die Beziehungen zwischen den Begriffen

Diese Beziehungen können in zwei Sichten betrachtet werden. Aus der Sicht des zweidimensionalen Lebenszyklus gibt es zwei unabhängige "Beziehungspfade", die der Methode-Mächtigkeits-Achse und der System-Vollständigkeits-Achse zuzuordnen sind. Die Ableitung der Baustein-Instanzen und deren Änderungen entsprechen dem Entwicklungsprozeß einer Methode und die Ableitungen und Änderungen der Baustein-Occurrences dem Entwicklungsprozeß eines Systems (Abb. 29).

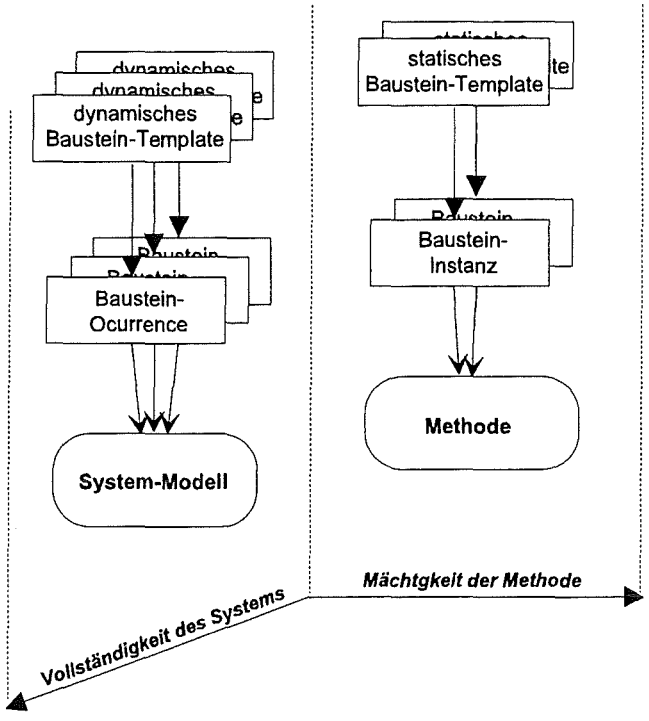


Abb. 29. Die Begriffe und der zweidimensionale Lebenszyklus

Diese getrennten "Beziehungspfade" sind von größter Bedeutung für das hier präsentierte Konzept. Die unabhängige Ableitung von Baustein-Instanzen und Baustein-Occurrences von den entsprechenden Templates schafft die Voraussetzung für die unabhängige Entwicklung der Methode und des Systems. Zwar werden die dynamischen Templates auf der Basis von Baustein-Instanzen generiert, jedoch sind die möglichen Änderungen der Template-Struktur begrenzt , so daß in vielen Fällen eine automatische Redefinition eines System-Modells möglich ist (siehe Kapitel 5.2 und 5.5.3).

Aus der Sicht der Konkretisierung gibt es nur einen durchgängigen Pfad, der von den statischen Baustein-Templates über die Baustein-Instanzen und damit auch über die dynamischen Baustein-Templates zu den Baustein-Occurrences führt. Eine Baustein-Instanz ist in dieser Sicht eine Konkretisierung eines statischen Templates und eine Baustein-Occurrence ist eine Konkretisierung einer Baustein-Instanz, mit denen entsprechende dynamische Baustein-Templates verbunden sind (Abb. 30).

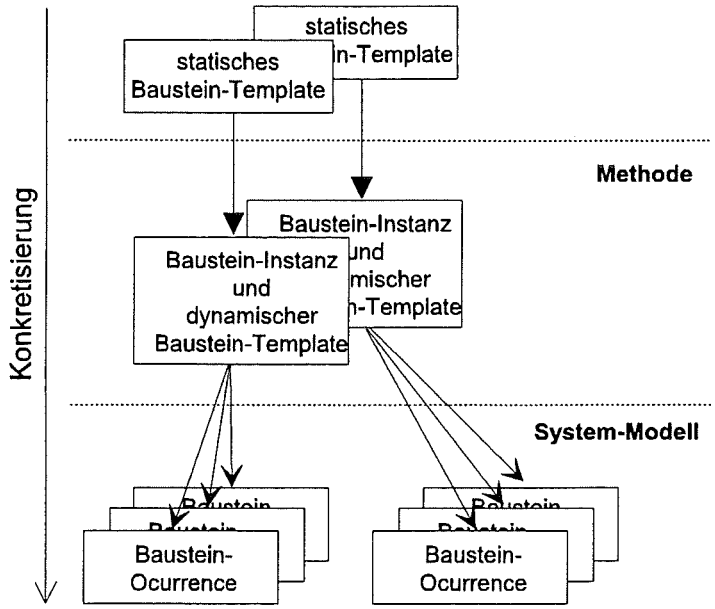


Abb. 30. Die Begriffe und die Konkretisierung

4.4.3 Ein Beispiel

Die Ableitung der Baustein-Instanzen von den statischen Baustein-Templates (Definition einer Methode), Generierung der dynamischen Baustein-Templates und Ableitung der Baustein-Occcurrences von diesen Baustein-Templates (Entwicklung eines System-Modells) wird anschaulich in der Abb.31 anhand eines Beispieles dargestellt. In diesem Beispiel wird eine

Methode definiert, die es ermöglicht, P/T-Netze (Place/Transition-Netze) zu erstellen. Es werden dabei entsprechende Petri-Netz-Konstrukte entwickelt (Place, Transition, Token und Arc), die für die Erstellung eines einfachen System-Modells verwendet werden.

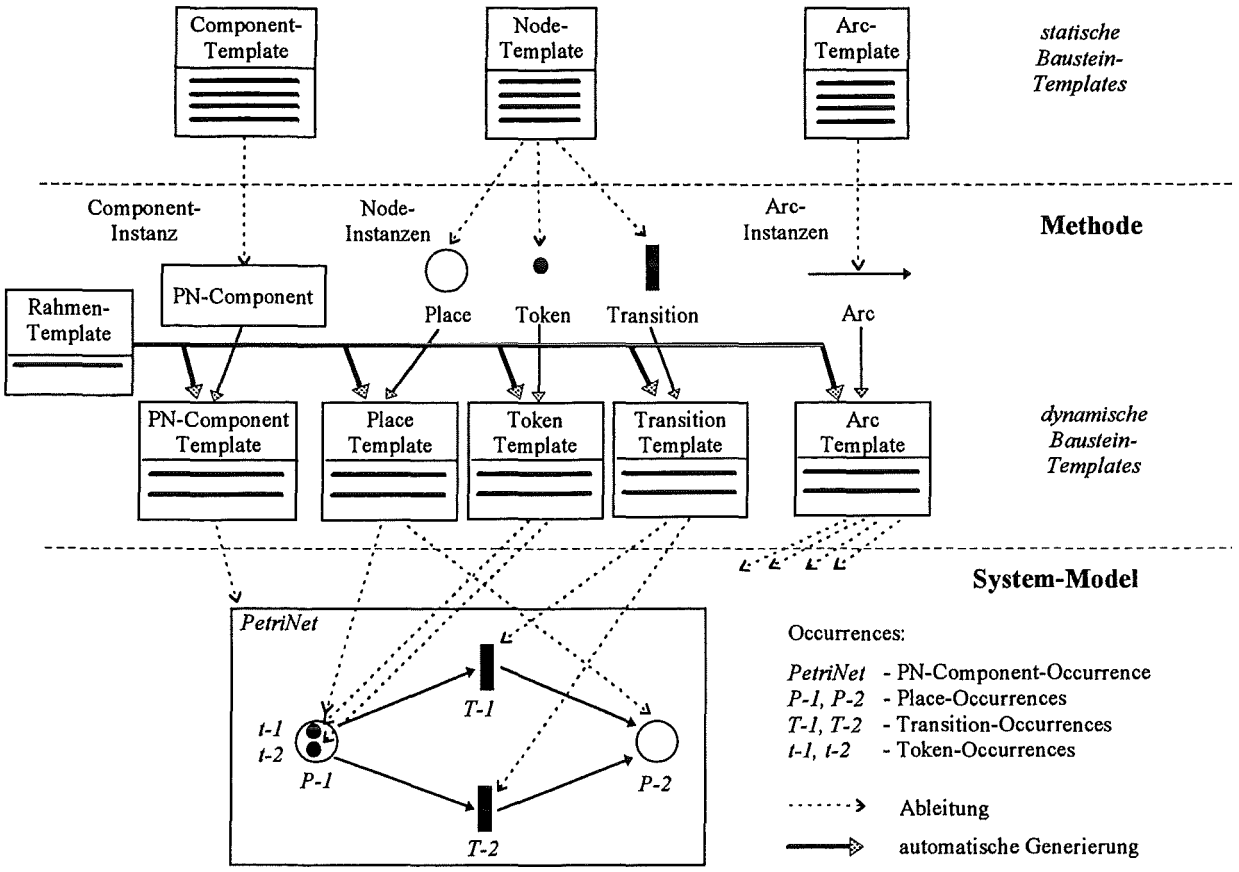


Abb. 31. Baustein-Templates, Baustein-Instanzen und Baustein-Occurrences für P/T-Netze (die "Ableitungskanten" für Arc-Occurrences werden der besseren Übersichtlichkeit des Bildes wegen nicht vollständig gezeigt)

Wie in dieser Abbildung dargestellt, werden bei der Definition der Methode von den statischen Baustein-Templates folgende Baustein-Instanzen abgeleitet:

- vom statischen Component-Template: die einzige Component-Instanz **PN-Component**,
- vom statischen Node-Template: drei Node-Instanzen: **Place**, **Transition** und **Token**,
- vom statischen Arc-Template: eine Arc-Instanz: **Arc**.

Bei der Ableitung der Baustein-Instanzen werden verschieden Informationen angegeben, welche die allgemeinen Eigenschaften der entsprechenden Baustein-Occurrences festlegen. Bei der Component-Instanz *PN-Component* wird definiert, welche Baustein-Occurrences die PN-Component-Occurrence beinhalten kann. In diesem Fall sind es: Place-, Transition- und Token-Occurrences. Bei den Node-Instanzen *Place*, *Transition* und *Token* werden die zulässige

Verbindungen mit anderen Baustein-Occurrences angegeben: Place-Occurrences dürfen nur mit Transition-Occurrences, Transition-Occurrences nur mit Place-Occurrences, und Token-Occurrences ebenfalls nur mit Place-Occurrences verbunden werden. Es werden dabei für die Repräsentation der Verbindungen zwischen Place- und Transition-Occurrences die Arc-Occurrences verwendet. Zusätzlich werden Informationen über der Darstellung und der Benutzerschnittstelle der entsprechenden Baustein-Occurrences angegeben. Es werden auch Daten definiert, die in den Baustein-Occurrences gespeichert werden können, wie etwa die Kapazität für die Place-Occurrences und Gewicht für die Arc-Occurrences.

Die Informationen aus den Baustein-Instanzen werden verwendet, um die folgenden dynamischen Baustein-Templates zu generieren:

- das einzige dynamische Component-Template **PN-Component-Template**,
- drei dynamische Node-Templates: **Place-**, **Transition-** und **Token-Template**,
- ein dynamisches Arc-Template: **Arc-Template**.

Dies geschieht durch Kopieren und eventuelle Vervollständigung des Rahmentemplates. Die Vervollständigung des Rahmentemplates betrifft nur das Place- und Arc-Template. Für diese Templates müssen zusätzliche Attribute für die Kapazität und das Gewicht eingefügt werden.

Von den dynamischen Baustein-Templates werden bei der Entwicklung des System-Modells (Petri-Netz-Modells) entsprechende Baustein-Occurrences abgeleitet, wobei gilt:

- vom dynamischen Component-Template **PN-Component-Template** wird nur eine Component-Occurrence **PN-Component-Occurrence** abgeleitet,
- von den übrigen dynamischen Baustein-Templates werden mehrere Baustein-Occurrences (**Place-**, **Transition-**, **Token-** und **Arc-Occurrences**) abgeleitet.

Bei der Entwicklung des System-Modells (durch die Ableitung der Baustein-Occurrences) werden in den dynamischen Baustein-Templates verschiedene Informationen angegeben, die für jede Baustein-Occurrence spezifisch sind, etwa:

- die existierenden Verbindungen mit anderen Baustein-Occurrences,
- die topologische Anordnung (d.h. die räumlichen Koordinaten der Baustein-Occurrences),
- die konkreten Daten, die in den Baustein-Occurrences gespeichert sind (die Kapazität für die Place-Occurrences und das Gewicht für die Arc-Occurrences).

Die Informationen aus den entsprechenden Baustein-Instanzen werden für die Unterstützung des Entwicklungsprozesses verwendet. Diese Informationen ermöglichen:

- die Baustein-Occurrences dem Entwickler zu präsentieren,
- eine entsprechende Benutzerschnittstelle (z.B. Menü) für jede Baustein-Occurrence zur Verfügung zu stellen,
- die zulässigen Verbindungen zwischen den Node-Occurrences zu erstellen,
- entsprechende Daten in den Baustein-Occurrences zu speichern.

Nach der Entwicklung des System-Modells wird die dabei eingesetzte Methode erweitert, so daß das Erstellen von Pr/T-Netzen (Prädikat/Tranistionsnetzen) möglich wird. Die bereits definierten Petri-Netz-Konstrukte (Baustein-Instanzen) werden dabei wie folgt geändert:

- in der Arc-Instanz wird die Definition des Gewichtes entfernt und die Definition der Kantenbeschriftung eingefügt,
- in der Token-Instanz wird die Definition der Liste der Token-Attributte eingefügt.

Die bereits existierenden dynamischen Baustein-Templates, die mit diesen Baustein-Instanzen verbunden sind (Arc-Template und Token-Template) werden durch Einfügen von neuen Attributen (die Kantenbeschriftung und die Liste der Token-Attribute) und Entfernen der nicht mehr gültigen Attribute (das Gewicht) automatisch geändert. Die im System-Modell bereits existierenden Arc- und Token-Occurrences, die als ausgefüllten Kopien des Arc- bzw. Token-Templates betrachtet werden können, werden ebenfalls geändert. Damit können sowohl in den existierenden als auch in den neu erzeugten Arc- und Token-Occurrences entsprechende Daten eingefügt und gespeichert werden. Nach diesen Änderungen kann die Entwicklung des System-Modells mit der erweiterten Methode (Pr/T-Netze) fortgeführt werden.

4.5 Zusammenfassung

Es existieren viele Ansätze zur Unterstützung der Entwicklung von Prototypen. In diesen Ansätzen werden zur Beschreibung der Entwicklung verschiedene Lebenszyklen verwendet. Die Lebenszyklen haben als gemeinsame Eigenschaft, daß sie den Einsatz einer vordefinierten Entwicklungsmethode voraussetzen. Ein System kann sich damit im Laufe der Entwicklung nur entlang der Vollständigkeit-Achse bewegen, die durch die nacheinanderfolgenden Prototypen des Systems bestimmt wird. In diesen Lebenszyklen werden daher nur sekundäre Änderungen unterstützt, welche die Änderungen der Spezifikation des Systems bedeuten. Primären Änderungen, die als Erweiterungen bzw. Modifikationen der Entwicklungsmethode verstanden werden, werden dagegen nicht unterstützt. Eine Methode wird dieser Arbeit mit der Beschreibungssprache zusammen mit der Vorgehensweise beim Einsatz der Sprache gleichgesetzt. Es wird dabei angenommen, daß die Erweiterungen bzw. Modifikationen der

Methode (d.h. die Evolution der Methode) hauptsächlich die Sprache betreffen. Der Einsatzbereich des in dieser Arbeit präsentierten Konzeptes beschränkt sich hierbei auf graphenbasierte Beschreibungssprachen.

Zur Unterstützung sowohl der sekundären als auch der primären Änderungen wurde in diesem Konzept der zweidimensionale Lebenszyklus eingeführt. In diesem Lebenszyklus erfolgt der Entwicklungsprozeß nicht mehr nur auf der Achse der Vollständigkeit des Systems, sondern in einem zweidimensionalen Raum, der durch die zwei Achsen der "Mächtigkeit der Methode" und der "Vollständigkeit des Systems" bestimmt wird. Die Mächtigkeits-Achse wird dabei durch die nacheinanderfolgenden Versionen der Methode bestimmt.

Die Vollständigkeit des Systems beschreibt den Erfüllungsgrad der an das System gestellten Anforderungen. Sie läßt sich nicht quantifizieren. Sie wird vom Entwickler geschätzt, um festzustellen, ob die Entwicklung fortgeführt oder beendet werden soll. Die Vollständigkeit wird daher nicht für den Vergleich der unterschiedlichen Systeme verwendet, sondern lediglich für die Abschätzung der Erfüllung der Anforderungen bei den nacheinanderfolgenden Prototypen des gleichen Systems.

Die Mächtigkeit der Methode beschreibt die Angemessenheit der Methode bezüglich der Anforderungen, die an das mit der Methode zu entwickelnde System gestellt werden. Die Mächtigkeit der Methode, ähnlich wie die Vollständigkeit des Systems, läßt sich ebenfalls nicht quantifizieren. Sie wird vom Entwickler geschätzt, um zu entscheiden, ob die bis dahin eingesetzte Methode für die Entwicklung der nächsten Prototypen des Systems weiter verwendet werden kann, oder ob sie zuvor erweitert werden soll. Die Mächtigkeit wird daher nicht für den Vergleich der unterschiedlichen Methoden angewandt, sondern lediglich für die Bestimmung der Angemessenheit der aktuell verwendeten Version der Methode für die weitere Entwicklung des Systems.

Zur Unterstützung des zweidimensionalen Lebenszyklus wurde ein Baustein-Konzept entwickelt. In diesem Konzept wird eine Methode durch die Entwicklung und Änderung der Baustein-Instanzen definiert bzw. erweitert. Mit Hilfe der Baustein-Instanzen werden Baustein-Occurrences erzeugt, die das zu entwickelnde System beschreiben. Die Abhängigkeit der Beschreibung des Systems (Baustein-Occurrences) von der Definition der Methode (Baustein-Instanzen) wurde dabei auf ein notwendiges Minimum reduziert. Die Änderungen der Methode haben damit keine bzw. eine geringe Auswirkung auf die bis dahin erstellte Beschreibung des Systems, was eine Voraussetzung für die Realisierung des zweidimensionalen Lebenszyklus ist.

In den nächsten Kapiteln wird die Grundstruktur und anschließend die formale Definition der Baustein-Templates präsentiert, die zur Entwicklung der Baustein-Instanzen (Definition der Methode) und zum Erzeugen der Baustein-Occurrences (Beschreibung des Systems) verwendet werden. Es wird des weiteren die prototypische Implementierung eines Werkzeugs vorgestellt, welches das Baustein-Konzept realisiert und damit den zweidimesnionalen Lebenszyklus unterstützt. Anschließend werden zwei beispielhafte Einsätze beschrieben, die zur Validierung des Baustein-Konzeptes und des Werkzeuges dienen.

5. Definition der Baustein-Templates

In diesem Kapitel werden die verschiedenen Baustein-Templates formal definiert. Am Anfang des Kapitels wird der allgemeine Aufbau der Baustein-Templates erläutert. Danach werden die Auswirkungen der Änderungen der Definition einer Methode (Änderungen der Definition der Baustein-Instanzen) auf die entsprechenden dynamischen Baustein-Templates und auf das bis dahin mit dieser Methode entwickelte System-Modell (Baustein-Occurrences) diskutiert. Anschließend wird die zur Definition der Baustein-Templates verwendete Notation (EXPRESS-G) dargestellt und die formale Definition aller Baustein-Templates präsentiert.

5.1 Die Grundstrukturen der Baustein-Templates

Sowohl die statischen als auch die dynamischen Baustein-Templates unterscheiden sich untereinander nur in einem begrenzten Umfang: für beide Gruppen der Baustein-Templates kann jeweils eine Grundstruktur identifiziert werden. Die Abweichungen der Struktur der konkreten statischen bzw. dynamischen Baustein-Templates von der entsprechenden Grundstruktur werden im weiteren besprochen.

Die Grundstrukturen sind hierarchisch mit Hilfe von mehreren "Unter-Templates" (die auch als Modelle bezeichnet werden) aufgebaut. Die hierarchische Struktur entsteht dadurch, daß die Attribute in einem Template (Baustein- bzw. Unter-Template) als die Unter-Templates definiert werden.

Die **statischen Baustein-Templates**, die zur Definition der Methode (durch die Ableitung der Baustein-Instanzen) dienen, bestehen aus drei Modellen: Implementations-, Sicht- und Verbindungsregel-Modell, die ihrerseits noch weiter unterteilt werden (Abb. 32). Die Konkretisierung dieser Modelle erfolgt im Instanziierungsprozeß einer Baustein-Instanz. Die konkretisierten Modelle (ausgefüllte "Unter-Templates") der Baustein-Instanz beschreiben Eigenschaften, die für alle auf der Basis dieser Baustein-Instanz erzeugten Baustein-Occurrences gemeinsam sind.

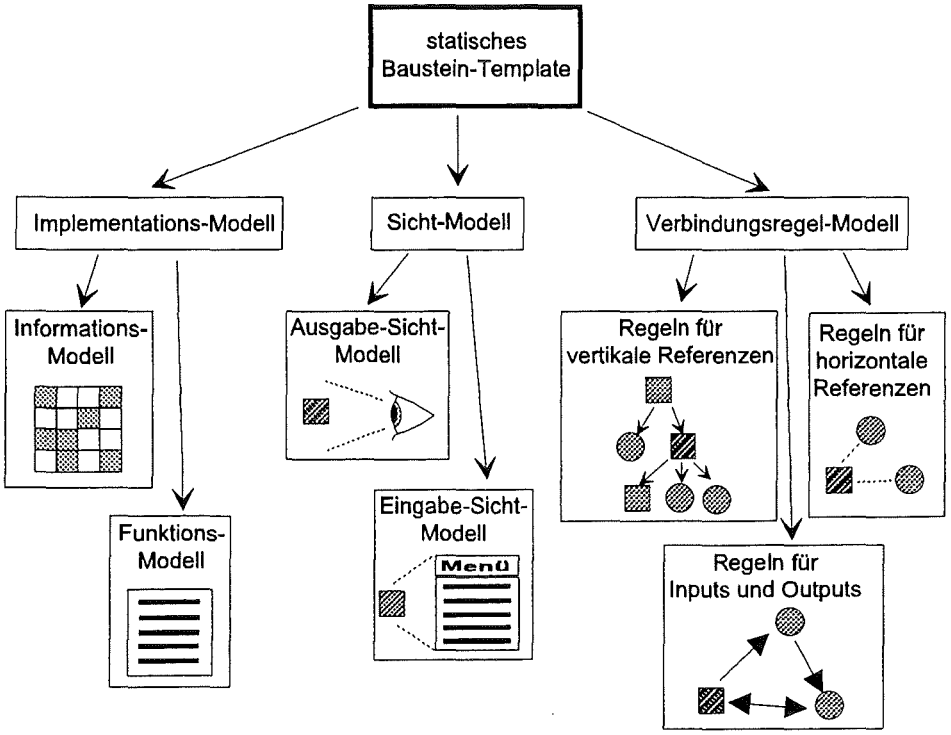


Abb. 32. Aufbau eines statischen Baustein-Templates

Das **Sicht-Modell** wird verwendet, um zu definieren, wie die Baustein-Occurrences dem Systementwickler präsentiert werden (Ausgabe) und welche Funktionalität ihm zur Verfügung gestellt wird (Eingabe). Das Modell wird in das Ausgabe-Sicht- und das Eingabe-Sicht-Modell unterteilt. Das **Ausgabe-Sicht-Modell** wird vor allem für die Definition des Layout der Baustein-Occurrences verwendet. Das **Eingabe-Sicht-Modell** wird für die Definition der verschiedenen Eingabeelemente (z.B. Menüs) der Baustein-Occurrences verwendet. Das Sicht-Modell dient daher zur Definition der Benutzerschnittstelle, die von der Baustein-Occurrences unterstützt wird.

Um die Entwicklung von komplexen Systemen zu erleichtern, können **verschiedene Sichten** verwendet werden, welche die Betrachtung des Systems nur unter einem bestimmten Aspekt ermöglichen. Die vorgesehen Sichten werden in der Component-Instanz bestimmt. Dazu wird ein **Allgemein-Sicht-Modell** verwendet, das in dem statischen Component-Template zusätzlich zur Verfügung steht.

Die eigentliche Definition der Sichten (d.h. der Ausgabe und Eingabe) erfolgt dagegen für jede Baustein-Instanz separat (mit Hilfe der oben beschriebenen Sicht-Modelle). Die Ausgabe und Eingabe kann dabei (muß aber nicht) für jede vorgesehene Sicht anders definiert werden. Damit erhalten die Baustein-Occurrences verschiedene Benutzerschnittstellen - in jeder Sicht werden sie anders präsentiert und sie stellen in jeder Sicht verschiedene Eingabefunktionalität zur Verfügung.

Das **Verbindungsregel-Modell** dient als Vorlage zur Definition von Regeln, die beschreiben, mit welchen Node-Occurrences welche Verbindungen zugelassen sind und welche Arc-Occurrences für die Repräsentation der Verbindungen verwendet werden. Insbesondere spezifizieren die Verbindungsregeln:

- mögliche **vertikale Referenzen** zu den Node-Occurrences, die in einem hierarchischen System-Modell untergeordnet sind
- mögliche **horizontale Referenzen** zu den Node-Occurrences aus der gleichen Hierarchieebene
- mögliche **Inputs** und **Outputs**, die Node-Occurrences (bzw. die Component-Occurrence) auch von verschiedenen Hierarchieebenen verbinden können

Sowohl die Input- und Output-Verbindungen als auch die horizontalen Referenz-Verbindungen dienen grundsätzlich zur Repräsentation von Verbindungen zwischen Node-Occurrences in einer Hierarchieebene des zu entwickelnden System-Modells. Sie können aber auch für die Darstellung von Verbindungen zwischen Node-Occurrences unterschiedlicher Hierarchieebenen verwendet werden. Für die Darstellung von Verbindungen zwischen Node-Occurrences unterschiedlicher Ebenen, die die Hierarchiestruktur des System-Modells beschreiben sollen, sind jedoch die vertikalen Referenz-Verbindungen vorgesehen. Mit der horizontalen Referenz-Verbindungen können alle Verbindungen dargestellt werden, die nicht als Input-, Output- oder Sub-Referenz-Verbindungen klassifiziert werden können.

Das Verbindungsregel-Modell steht nur in dem statischen Component- und Node-Template zur Verfügung. In dem statischen Arc-Template fehlt dieses Modell, da die Verbindungsregeln bei der Definition der Component-Instanz und der Node-Instanzen eindeutig beschrieben werden.

Die Verbindungsregeln unterstützen die Konsistenzhaltung des Modells des zu entwickelnden Systems, indem sie die zulässige Verbindungen zwischen Node-Occurrences (und der Component-Occurrence) beschreiben. In dem das hier präsentierte Konzept unterstützenden Tool soll die Erstellung nur von den so definierten zulässigen Verbindungen möglich sein.

Das **Implementations-Modell** dient als Vorlage zur Definition sowohl des Verhaltens von Baustein-Occurrences als auch der Information, die in den einzelnen Baustein-Occurrences gespeichert werden kann. Es werden dazu zwei Template-Modelle verwendet: Informations- und Funktions-Modell. Das **Informations-Modell** dient zur Definition der Daten einer Baustein-Occurrence (Namen, Typen) und das **Funktions-Modell** zur Definition der Funktionen, welche diese Daten manipulieren können. Dies entspricht dem Prinzip der Einkapselung von Daten und Funktionen in einem Objekt aus dem objektorientierten Paradigma.

Die genaue Struktur **der dynamischen Baustein-Templates**, die zur Entwicklung des System-Modells (durch die Ableitung der Baustein-Occurrences) dienen, ist nicht vordefiniert und sie kann sich dynamisch ändern. Diese möglichen Änderungen sind jedoch sehr gering (siehe Kapitel 5.5.3), so daß sich eine Grundstruktur der dynamischen Baustein-Templates identifizieren läßt.

In den dynamischen Baustein-Templates stehen zwei Modellen zur Verfügung: Verbindungs- und Informations-Modell. Das Verbindungs-Modell wird weiter unterteilt (Abb. 33). Die Konkretisierung dieser Modelle erfolgt im Instenziierungsprozess einer Baustein-Occurrence. Die konkretisierten Modelle aller Baustein-Occurrences eines System-Modells beschreiben die Struktur des Graphen, das dieses System-Modell repräsentiert.

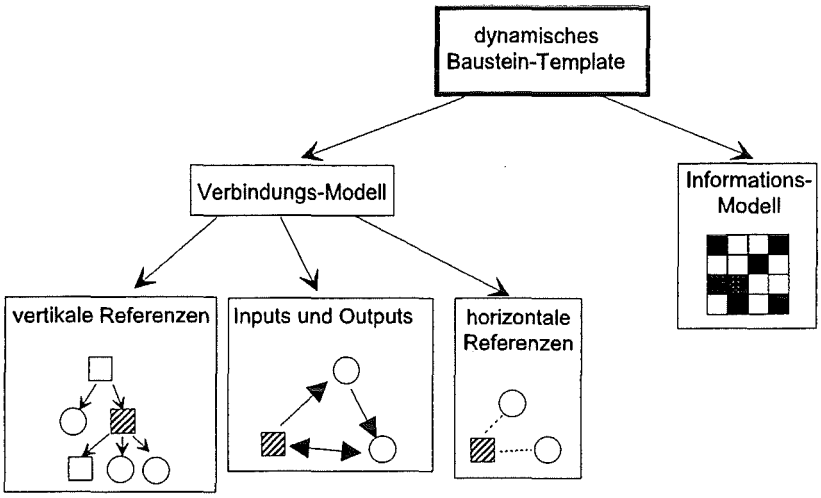


Abb. 33. Aufbau eines dynamischen Baustein-Templates

Das **Verbindungs-Modell** dient als Vorlage zur Definition von Verbindungen einer Baustein-Occurrence zu anderen Baustein-Occurrences. Insbesondere können mit Hilfe dieses Modells folgende Verbindungen definiert werden:

- **vertikale** Referenzen
- **horizontale** Referenzen
- **Inputs** und **Outputs**

Der Aufbau des Verbindungs-Modells der dynamischen Baustein-Templates ist ähnlich wie der Aufbau des Verbindungsregel-Modells der statischen Baustein-Templates. Bei den Baustein-Instanzen (die von den statischen Baustein-Templates abgeleitet werden) werden jedoch nur mögliche Verbindungen zwischen Baustein-Occurrences festgelegt. Bei den Baustein-Occurrences (die von den dynamischen Baustein-Templates abgeleitet werden) werden dagegen konkrete Verbindungen mit anderen Baustein-Occurrences beschrieben.

Das **Informations-Modell** dient zur Definition der konkreten Werte der Daten einer Baustein-Occurrence, im Unterschied zum Informations-Modell eines statischen Baustein-Templates, das nur für die Definitionen der Datenstruktur (d.h. Datennamen, Datentypen) verwendet wird.

Die genaue Struktur des Informations-Modell kann nicht vom Anfang an festgelegt werden, da sie von der Definition der Daten (Informations-Modell eines statischen Baustein-Templates) der entsprechenden Baustein-Instanzen abhängig ist. Diese dynamische Struktur wird aufgrund dieser vorliegenden Definition der Daten einer Baustein-Instanz automatisch generiert und nach jeder Änderung dieser Definition entsprechend aktualisiert.

5.2 Änderungen der Methode

Nachdem die Grundstrukturen der statischen und dynamischen Baustein-Templates vorgestellt wurden, können die Änderungen der Methode (Baustein-Instanzen) und ihre Auswirkung auf die dynamischen Baustein-Templates und auf das bis dahin erstellte System-Modell (Baustein-Occurrences) diskutiert werden.

Die Änderungen der Methode können folgende Aktivitäten umfassen:

1. Definition einer neuen Baustein-Instanz
2. Entfernen einer existierenden Baustein-Instanz
3. Änderungen der Informationen in einer existierenden Baustein-Instanz

Da mit jeder Baustein-Instanz ein dynamisches Baustein-Templates verbunden ist, von dem Baustein-Occurrences abgeleitet werden, haben diese Änderungen eine Auswirkung auf das bereits entwickelte System-Modell. Diese Auswirkungen werden im folgenden diskutiert.

5.2.1 Definition einer neuen Baustein-Instanz

Nach der Definition einer neuen Baustein-Instanz wird auch ein entsprechendes dynamisches Baustein-Template automatisch generiert. Dieses Baustein-Template wird anschließend zu der Liste der vorhandenen dynamischen Baustein-Templates hinzugefügt. Bei der weiteren Entwicklung des System-Modells können damit Baustein-Occurrences von dem neuen Typ (der durch die neue Baustein-Instanz beschrieben wird) eingesetzt werden.

5.2.2 Entfernen einer existierenden Baustein-Instanz

Nach dem Entfernen einer existierenden Baustein-Instanz wird auch das mit der Baustein-Instanz verbundene dynamische Baustein-Template aus der Liste der vorhandenen Templates entfernt. In dem bereits entwickelten System-Modell werden alle Baustein-Occurrences, die von diesem dynamischen Baustein-Templates abgeleitet wurden, ebenfalls entfernt. Wenn die entfernte Baustein-Instanz eine Node-Instanz ist (d.h. die entfernten Baustein-Occurrences Node-Occurrences sind), werden auch folgende Baustein-Occurrences gelöscht:

- alle Arc-Occurrences, die Verbindungen mit den entfernten Node-Occurrences repräsentieren,
- alle Node-Occurrences (zusammen mit den Arc-Occurrences, welche die Verbindungen mit diesen Node-Occurrences repräsentieren), die den entfernten Node-Occurrences untergeordnet sind (durch Sub-Referenzen).

Wenn die entfernte Baustein-Instanz eine Arc-Instanz ist, die zur Definition der Eigenschaften der Arc-Occurrences zur Darstellung der Sub-Referenzen dient, werden auch die Node-Occurrences gelöscht, die durch diese Sub-Referenzen mit der übergeordneten Node-Occurrences (bzw. mit der übergeordneten Component-Occurrence) verbunden sind. Bei dem Löschen dieser Node-Occurrences werden auch andere Baustein-Occurrences nach den oben aufgelisteten Regeln gelöscht.

5.2.3 Änderungen der Informationen in einer existierenden Baustein-Instanz

Hier sind folgende Änderungen möglich:

- Änderungen des Sicht-Modells
- Änderungen des Verbindungsregel-Modell
- Änderungen des Implementations-Modells

Änderungen des Sicht-Modells haben keine Auswirkung auf das entsprechende dynamische Baustein-Template und auf das bereits entwickelte System-Modell. Nach dieser Änderungen erhalten die betroffenen Baustein-Occurrences lediglich eine andere Benutzerschnittstelle. Das dynamische Baustein-Template bleibt unverändert. Eine Ausnahme ist die Änderung der Information im Sicht-Ausgabe-Modell, die bestimmt, ob die entsprechenden Baustein-Occurrences ihre Position speichern sollen (siehe Kapitel 5.4.1.2). Hier wird das entsprechende dynamische Baustein-Template umdefiniert und die Informations-Modelle der betroffenen Baustein-Occurrences werden entsprechend umstruktuiert (es werden zusätzliche Attribute für das Speichern der Position hinzugefügt bzw. die existierenden Attribute für das Speichern der Position entfernt; siehe Kapitel 5.5.3).

Nach **Änderungen des Verbindungsregel-Modell** kann eine Situation entstehen, in der bestimmte Verbindungen zwischen Baustein-Occurrences in dem bereits entwickelten System-Modell nicht mehr zulässig sind. Hier wird eine Umstrukturierung des System-Modells durchgeführt, indem die Arc-Occurrences, welche die betroffenen Verbindungen repräsentieren, gelöscht werden. Es werden dabei auch andere Baustein-Occurrences nach den im vorherigen Abschnitt beschriebenen Regeln gelöscht. Das dynamische Baustein-Template bleibt dabei unverändert.

Änderungen des Implementations-Modells, die das **Funktions-Modell** betreffen, haben keine Auswirkung auf das entsprechende dynamische Baustein-Template und auf das bis dahin

entwickelte System-Modell. Nach diesen Änderungen erhalten die betroffenen Baustein-Occurrences lediglich ein anderes Verhalten, z.B. während der Simulation. Das dynamische Baustein-Template bleibt dabei unverändert.

Die Änderungen, die das **Informations-Modell** einer Baustein-Instanz betreffen, haben dagegen immer eine Auswirkung auf das mit der Baustein-Instanz verbundene dynamische Baustein-Template und damit auch auf die entsprechenden Baustein-Occurrences. In dem Fall ändert sich die Definition der Daten, die in den Baustein-Occurrences gespeichert werden sollen. Hier sind zwei Typen von Änderungen möglich:

1. Hinzufügen oder Entfernen einer Definition eines Datums - in einem dynamischen Baustein-Template muß die Definition eines neuen Attributes hinzugefügt bzw. die existierende Definition eines Attributes entfernt werden. In den bereits existierenden Baustein-Occurrences werden dabei dentsprechende Datenstrukturen hinzugefügt bzw. entfernt.
2. Änderung einer existierenden Definition eines Datums
    a) Änderung des Namens des Datums - in einem dynamischen Baustein-Template wird der Name des entsprechenden Attributes geändert. In den bereits existierenden Baustein-Occurrences werden dabei die entsprechenden Datenstrukturen umbenannt.
    a) Änderungen des Typs des Datums - in einem dynamischen Baustein-Template wird der Typ des entsprechenden Attributes geändert. In den bereits existierenden Baustein-Occurrences werden dabei entsprechende Datenstrukturen wenn möglich angepaßt. Wenn die Anpassung nicht möglich ist, muß der Entwickler entscheiden, wie die Änderung behandelt werden soll.

5.3 Die verwendete Notation

Zur Definition der Baustein-Templates wird die formale Beschreibungssprache EXPRESS [ISO92a] eingesetzt. In den folgenden Kapiteln wird zum besseren Verständnis die graphische Notation EXPRESS-G verwendet und es werden nur die wichtigsten Teile der Definitionen dargestellt. Die vollständige Definitionen, die mit EXPRESS erstellt wurden, befinden sich im Anhang A, B und C.

Die wichtigsten Symbole der Notation von EXPRESS-G werden in der Abb. 34 dargestellt.

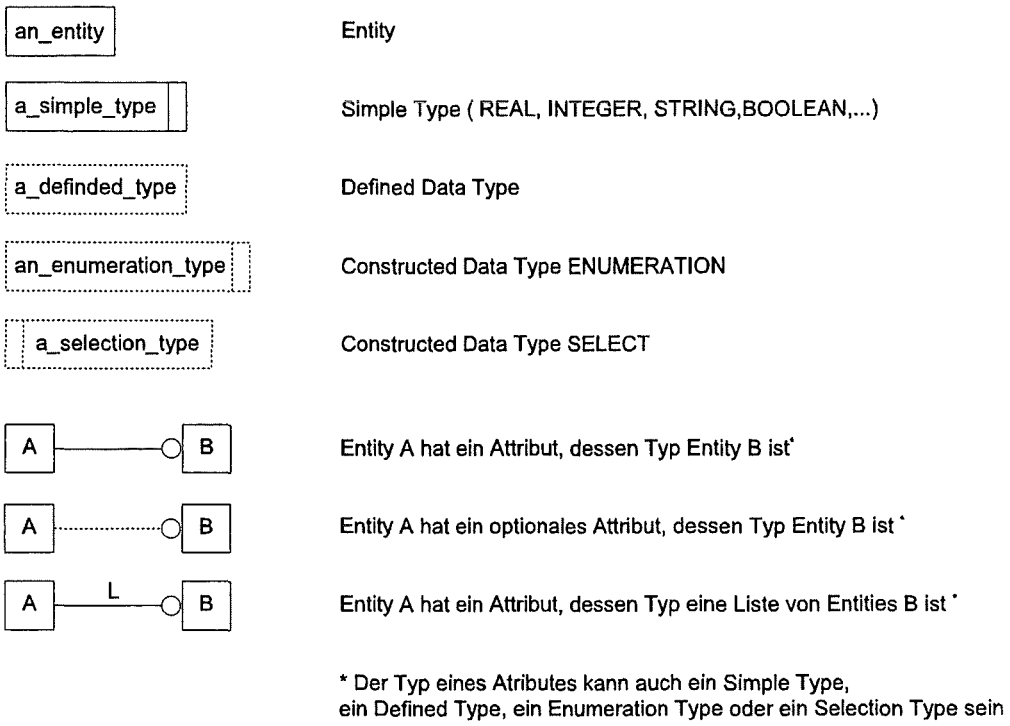


Abb. 34. Die wichtigsten Symbole von EXPRESS-G [ISO92a]

Das Hauptkonstrukt "Entity" mit seinen Attributen und die zahlreichen vordefinierten Datentypen der EXPRESS-Sprache bilden ein ideales Werkzeug zur Definition von Baustein-Templates. Die Instanzen von Entities, die statt Attributdefinitionen konkrete Werte beinhalten, können als Darstellung von ausgefüllten Templates (Baustein-Objekte) dienen. Die Definitionen der Entities werden in der sogenannten Schemas zusammengefaßt. Die Beschreibungen der Entity-Instanzen werden dagegen in der sogenannten Exchange Structure gespeichert [ISO92b]. Die Exchange Structure kann Entity-Instanzen mehrerer Schemas beinhalten. Hier wird jedoch angenommen, daß eine Exchange Struktur Entity-Instanzen nur eines Schemas beinhaltet. Die Exchange Structure kann daher als eine Schema-Instanz gesehen werden. Dabei können sowohl die Schemas als auch die Exchange Structures als Text gespeichert werden. Das Konzept des Schemas und der

Exchange Structure und die Abbildung auf das Baustein-Konzept wird in der Abb. 35 anschaulich dargestellt.

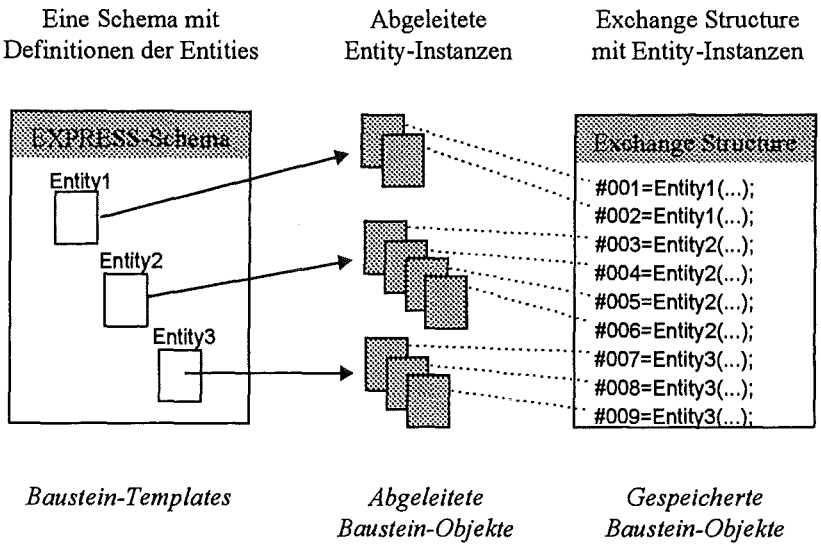


Abb. 35. EXPRESS-Schema, Exchange Structure und die Abbildung auf das Baustein-Konzept

Der Einsatz von EXPRESS erlaubt eine standardisierte Definition (und Dokumentation) von

1. **Baustein-Templates** - als ein **Schema** mit Definitionen der entsprechenden Entities,
2. **Baustein-Objekten** - als eine **Exchange Structure** mit Beschreibungen der entsprechenden Entity-Instanzen, wobei die Exchange Structure mit der Beschreibungen von:

   ⇒ **Baustein-Instanzen** zugleich die Definition der Methode darstellt,

   ⇒ **Baustein-Occurrences** zugleich die Definition des mit der Methode entwickelten System-Modells darstellt.

5.4 Die statischen Baustein-Templates[8]

Von den statischen Baustein-Templates (statisches Component-, Node- und Arc-Template) werden Baustein-Instanzen (Component-, Node- und Arc-Instanzen) abgeleitet, die eine Methode definieren, die zur Entwicklung eines Systems eingesetzt werden kann. Eine abgeleitete Baustein-Instanz beschreibt bestimmte Eigenschaften, die alle Baustein-Ocurrences dieser Baustein-Instanz besitzen. Sie dient auch als Vorlage bei der Generierung eines entsprechenden dynamischen Baustein-Templates, von dem diese Baustein-Ocurrences abgeleitet werden.

5.4.1 Das statische Component-Template

Die Definition des statischen Component-Templates stellt Abb. 36 dar.

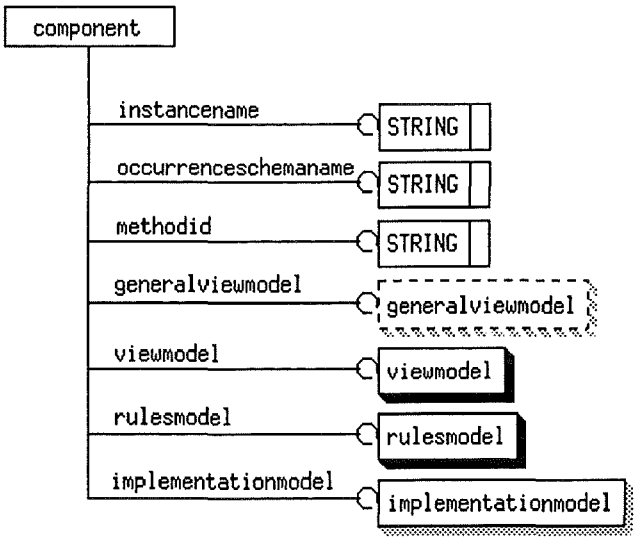


Abb. 36. Definition des statischen Component-Template[9]

InstanceName[10] spezifiziert den Namen der von dem Baustein-Template abgeleiteten Component-Instanz. *OccurrenceSchemaName* definiert einen Namen des Schemas, das die Definitionen von dynamischen Baustein-Templates beinhalten soll (Dieses Schema wird automatisch generiert).

---

[8] Für die vollsttändige Definition der statischen Baustein-Templates siehe Anhang A.

[9] Schattierte Kästchen stellen ein Element mit weiteren Unterelementen dar, die aber zur besseren Anschaulichkeit weggelassen sind

[10] Die graphische Darstellungen wurden mit einem EXPRESS-G-Editor erstellt. Da EXPRESS-Namen "case-insensitive" sind, werden sie in der Abbildungen immer kleingeschrieben - in dem Text werden sie jedoch zur besseren Verständnis großgeschrieben

MethodId ist ein Identifikator der Methode, die durch Ableitung der Baustein-Instanzen definiert wird. Die übrigen Elemente stellen hierarchische Template-Modelle dar, die im weiteren erklärt werden. Das Template-Modell *GeneralViewModel* tritt nur in dem Component-Template auf. Die Template-Modelle *ViewModel*, *RulesModel* und *ImplementationModel* sind Bestandteile der übrigen statischen Baustein-Templates (Node-Klasse und Arc-Template; das letzte jedoch ohne *RulesModel*) - sie werden daher allgemein, ohne jegliche Verweise auf die konkreten statischen Templates, beschrieben.

5.4.1.1 Allgemeines Sicht-Modell

Das allgemeine Sicht-Modell (*GeneralViewModel*) einer Component-Instanz stellt eine Liste mit der Beschreibungen der vorgesehenen Sichten dar. Diese Sichten können bei der Entwicklung eines System-Modells verwendet werden, um das System-Modell unter verschiedenen Aspekten darzustellen. In dem *GeneralViewModel* werden nur allgemeine Eigenschaften der Sichten festgelegt (Abb. 37). Die vollständige Definitionen der Sichten erfolgen in den Sicht-Modellen (*ViewModel*) einzelner Baustein-Instanzen, in denen die Art und Weise der Präsentation (Sicht-Ausgabe-Modell) von Baustein-Ocurrences dieser Baustein-Instanzen und die von den Baustein-Ocurrences zur Verfügung gestellte Eingabe-Funktionalität (Eingabe-Sicht-Modell) beschrieben wird.

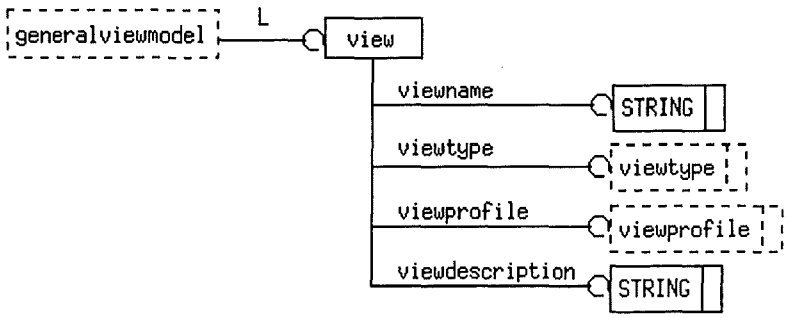


Abb. 37. Definition des Template-Modells GeneralViewModel

Für die Beschreibung jeder Sicht muß jeweils eine Entity *View* definiert werden. Diese Definition umfaßt u.a. die Spezifikation des Sichtnamen (*ViewName*) und der Sichtbeschreibung (*ViewDescription*). Die Sichtbeschreibung dient nur als Hilfe bei der Auswahl einer entsprechenden Sicht während der Entwicklung eines System-Modells - sie hat keinen Einfluß auf die Eigenschaften der betroffenen Sicht. Die Attribute *ViewType* und *ViewProfile* sind von dem Aufzählungstyp ENUMERATION. Für diese Attribute sind daher nur bestimmte Werte

zugelassen. Im EXPRESS-G wird die Darstellung von diesen Werten nicht unterstützt - sie werden deswegen in der Abbildung nicht gezeigt. Mit dem Attribut *ViewType* wird festgelegt, ob die Elemente eines System-Modells (Knoten und Kanten) bei der Präsentation automatisch positioniert werden. Bei dem Wert ***none*** findet kein automatisches Positionieren statt. Bei dem Wert ***tree*** werden die Knoten und Kanten wenn möglich so positioniert, daß sie eine Baumstruktur bilden. Eine Sicht von dem letzten Typ kann z.B. für die Darstellung der Hierarchieebenen eines hierarchischen System-Modells verwendet werden. Mit dem Attribut *ViewProfile* wird ein Profil der Sicht definiert, das einen Einfluß auf das automatische Positionieren der Elemente hat. Bei dem Wert ***horizontal*** werden als Grundlage für das Positionieren von Knoten nur die Verbindungen auf der gleichen Hierarchieebene, d.h. Inputs, Outputs und horizontale Referenzen, berücksichtigt - bei dem Wert ***vertical*** werden dagegen nur die vertikalen Verbindungen, d.h. Sub-Referenzen in Betracht gezogen.

5.4.1.2 Sicht-Modell

Das Sicht-Modell (*ViewModel*) einer Baustein-Instanz, das für die Definition der Benutzerschnittstelle von Baustein-Ocurrences der Baustein-Instanz vorgesehen ist, wird in zwei Untermodelle aufgeteilt (Abb. 38). *ViewOutputModel* dient zur Definition der Ausgabe-Sicht der Baustein-Ocurrences, d.h. es wird in dem Modell festgelegt, wie die Baustein-Ocurrences dem Systementwickler präsentiert werden. *ViewInputModel* wird verwendet, um die Eingabe-Sicht der Baustein-Ocurrences zu definieren, d.h. es wird in dem Modell festgelegt, welche Eingabe-Funktionalität dem Systementwickler von den Baustein-Ocurrences zur Verfügung gestellt wird.

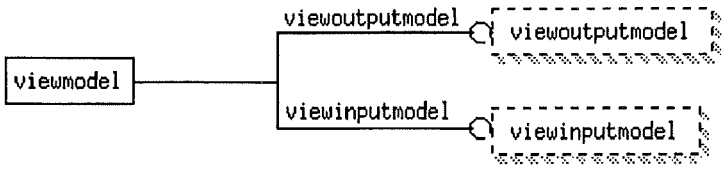


Abb. 38. Definition des Modells ViewModel

Ausgabe-Sicht-Modell

Das Ausgabe-Sicht-Modell (*ViewOutputModel*) stellt eine Liste mit Definitionen von Sicht-Ausgaben für verschiedene Sichten dar (Abb. 39), die in dem *GeneralViewModel* einer Component-Instanz aufgelistet sind. Es brauchen nicht alle Sichten bei diesem Modell berücksichtigt zu werden. Baustein-Ocurrences einer Baustein-Instanz, für die keine Sicht-Ausgabe für eine Sicht definiert wird, werden in der betroffenen Sicht nicht angezeigt.

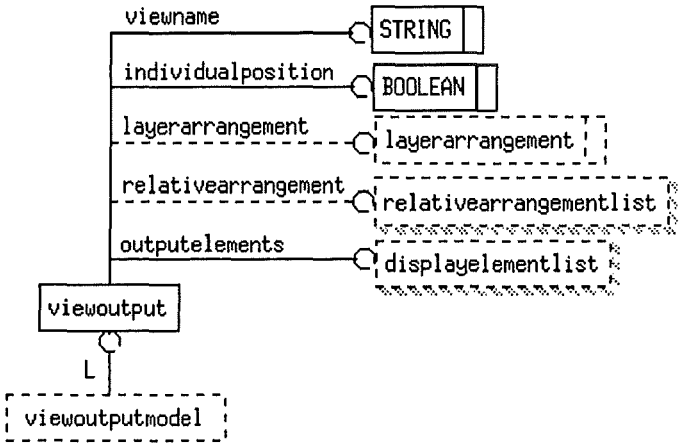


Abb. 39. Definition des Template-Modells ViewOutputModel

In der Entity *ViewOutput*, die zur Definition einer Sicht-Ausgabe dient, wird mit dem Attribut *ViewName* der Name der betroffenen Sicht genannt. Mit dem Attribut *IndividualPosition* wird es definiert, ob jede Baustein-Ocurrence der Baustein-Instanz ihre individuelle Position speichern soll. Dieses Attribut soll den Wert FALSE haben, wenn die Positionen der Baustein-Ocurrences automatisch bestimmt werden sollen. Das automatische Positionieren kann entweder durch eine entsprechende Definition einer Sicht in dem *GeneralViewModel* einer Component-Instanz oder durch die Spezifikation des optionalen Attributes *RelativeArrangementList* der Entity *ViewOutput* bestimmt werden. Mit diesem Attribut kann eine automatische Ausrichtung der Baustein-Ocurrences relativ zu der anderen Baustein-Ocurrences bestimmt werden, mit denen sie mit einer horizontalen Referenz verbunden sind. Das optionale Attribut *LayerArrangement* kann verwendet werden, um festzulegen, ob eine zuletzt erzeugte Baustein-Ocurrence im Vordergrund (der Wert ***top***) oder im Hintergrund (der Wert ***bottom***) präsentiert werden soll. Die graphische Elemente, die zur Präsentation der Baustein-Ocurrences verwendet werden, werden in dem Attribut *OutputElements* aufgelistet.

Die verfeinerte Definition des Attributes *RelativeArrangement* der Entity *ViewOutput* wird in der Abb. 40 dargestellt.

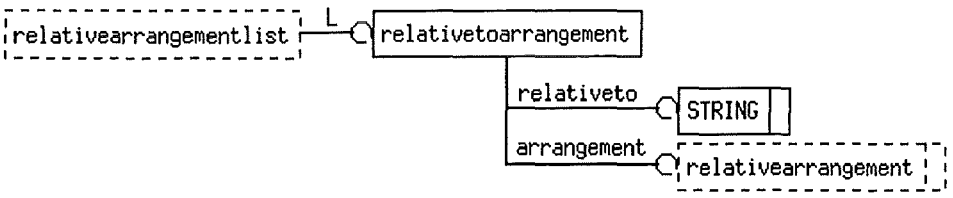


Abb. 40. Verfeinerte Definition des Attributes RelativeArrangement (vom benutzerdefinierten Typ RelativeArrangementList)

In der Liste können Entities *RelativeArrangement* definiert werden, in denen der Instanz-Name der Bezugs-Baustein-Ocurrence (Attribut *RelativeTo*) und die Art der Ausrichtung (Attribut *Arrangement*) spezifiziert wird. Die zulässige Werte des zweiten Attributes, das von dem Aufzählungstyp ENUMERATION ist, sind:

- ***inside*** - die Baustein-Ocurrence wird in der geometrischen Mitte der Bezugs-Baustein-Ocurrence positioniert
- ***bottom***, ***top***, ***right***, ***left*** - die Baustein-Ocurrence wird unterhalb, oberhalb, an der rechten Seite bzw. an der linken Seite der Bezugs-Baustein-Ocurrence positioniert
- ***ring*** - wenn mehrere Baustein-Ocurrence relativ zu der gleichen Bezugs-Baustein-Ocurrence ausgerichtet werden sollen, werden sie in einem Kreis um die Bezugs-Baustein-Ocurrence positioniert
- ***invisible*** - die Baustein-Ocurrence, die mit einer Referenz mit der Bezugs-Baustein-Ocurrence verbunden ist, wird nicht angezeigt

Die relative Ausrichtung ist vor allem für die Baustein-Ocurrences vorgesehen, deren horizontale Referenzen zu der Bezugs-Baustein-Ocurrences sich dynamisch ändern können (z.B. Tokens in einem Petri-Netz, die von einer Stelle zu anderen Stelle beim Schalten einer Transition fließen; für ein Beispiel siehe Kapitel 7.2.5).

In der Abb. 41 wird die verfeinerte Definition des Attributes *OutputElements* der Entity *ViewOutput* dargestellt.

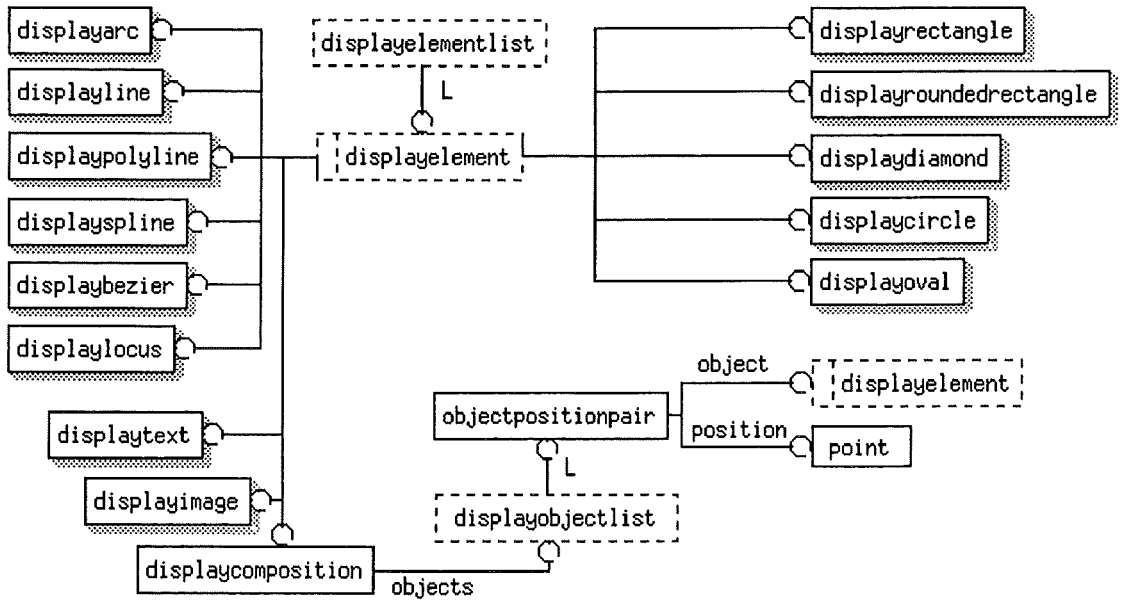


Abb. 41. Verfeinerte Definition des Attributes OutputElements (vom benutzerdefinierten Typ DisplayElementsList)

Die Liste *DisplayElementList* kann Elemente (*DisplayElement*) von dem Auswahltyp SELECTION beinhalten. Das bedeutet, daß einzelne Elemente der Liste verschiedene Entities repräsentieren können. Die zulässige Entities für diese Liste sind:

- *DisplayArc*, *DisplayLine*, *DisplayPolyline*, *DisplaySpline*, *DisplayBezier*, *DisplayLocus* - für die Darstellung von verschiedenen Arten von Kurven
- *DisplayRectangle*, *DisplayRoundedRectangle*, *DisplayDiamond*, *DisplayCircle*, *DisplayOval* - für die Darstellung von verschiedenen Arten von geometrischen Figuren
- *DisplayText* - für die Darstellung vom Text
- *DisplayImage* - für die Darstellung von Ikonen
- *DisplayComposition* - für die Darstellung von Kompositionen von allen hier aufgelisteten Elementen (inklusive *DisplayComposition*)

Jedes Element der Liste *DisplayObjectList* in dem Attribut *Objects* der Entity *DisplayComposition* ist eine Entity (*ObjectPositionPair*) mit zwei Attribute:

- Attribut *Object*, das eine der oben aufgelisteten Entities repräsentieren kann
- Attribut *Position*, das die Position des in dem Attribut *Object* spezifizierten Ausgabeelementes innerhalb der Komposition bestimmt

Die Tatsache, daß in der mit der Entity *DisplayComposition* definierten Komposition wiederum Entities vom Typ *DisplayComposition* auftreten können, erlaubt den Entwurf von nahezu beliebigen und auch sehr komplizierten Darstellungen der Baustein-Ocurrence.

Weil die Darstellung der Definition aller Ausgabe-Elemente zu umfangreich wäre, wird hier nur als Beispiel die Definition des Ausgabeelementes *DisplayRectangle* gezeigt (Abb. 42).

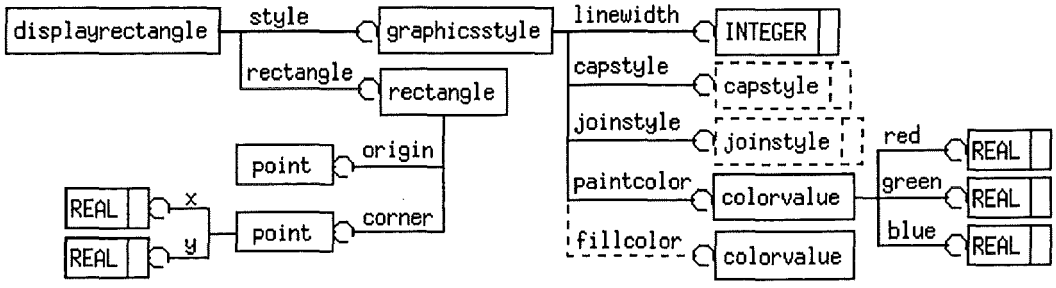


Abb. 42. Definition der Entity DisplayRectangle

Die Definition des Ausgabeelementes *DisplayRectangle* umfaßt die Spezifikation von verschiedenen graphischen Eigenschaften (Entity *GraphicsStyle*) sowie die Bestimmung von räumlichen Koordinaten des Rechteckes (Entity *Rectangle*). Andere Ausgabe-Elemente werden auf ähnliche Weise definiert. Bei dem Ausgabe-Element *DisplayText* müssen zusätzlich der

entsprechende Text und seine Formatierung und bei dem Ausgabe-Element *DisplayImage* die entsprechende Ikone spezifiziert werden.

Eingabe-Sicht-Modell

Das Eingabe-Sicht-Modell (*ViewInputModel*) stellt eine Liste mit Definitionen von Sicht-Eingaben für verschiedene Sichten dar (Abb. 43). Nicht alle Sichten brauchen bei diesem Modell berücksichtigt zu werden. Baustein-Ocurrences einer Baustein-Instanz, für die keine Sicht-Eingabe für eine Sicht definiert wird, stellen dem Systementwickler keine spezifische Eingabe-Funkionalität in der betroffenen Sicht zur Verfügung.

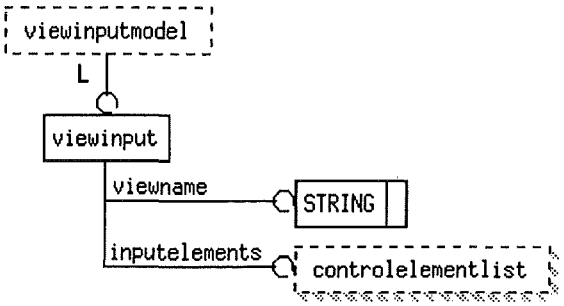


Abb. 43. Definition des Template-Modells ViewInputModel

In der Entity *ViewInput*, die zur Definition einer Sicht-Eingabe dient, wird mit dem Attribut *ViewName* der Name der betroffenen Sicht genannt. Die Kontrollelemente, die die Eingabe-Funktionalität der Baustein-Ocurrences in der mit dem Attribut *ViewName* spezifizierten Sicht unterstützen sollen, werden in dem Attribut *InputElements* aufgelistet.

In der Abb. 44 wird die verfeinerte Definition des Attributes *InputElements* dargestellt.

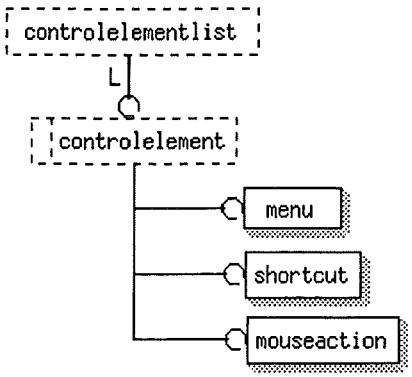


Abb. 44. Verfeinerte Definition des Attributes InputElements (vom benutzerdefinierten Typ ControlElementList)

Die Liste *ControlElementList* kann Elemente (*ControlElement*) von dem Auswahltyp SELECTION beinhalten. Das bedeutet, daß einzelne Elemente der Liste verschiedene Entities repräsentieren können. Die zulässige Entities für diese Liste sind:

- *Menu* - zur Definition von Menüs
- *ShortCut* - zur Definition von Tastenkürzeln
- *MouseAction* - zur Definition von Maus-Aktionen

Weil die Darstellung der Definition aller Kontrollelemente zu umfangreich wäre, wird hier nur als Beispiel die Definition des Kontrollelementes *Menu* gezeigt (Abb. 45).

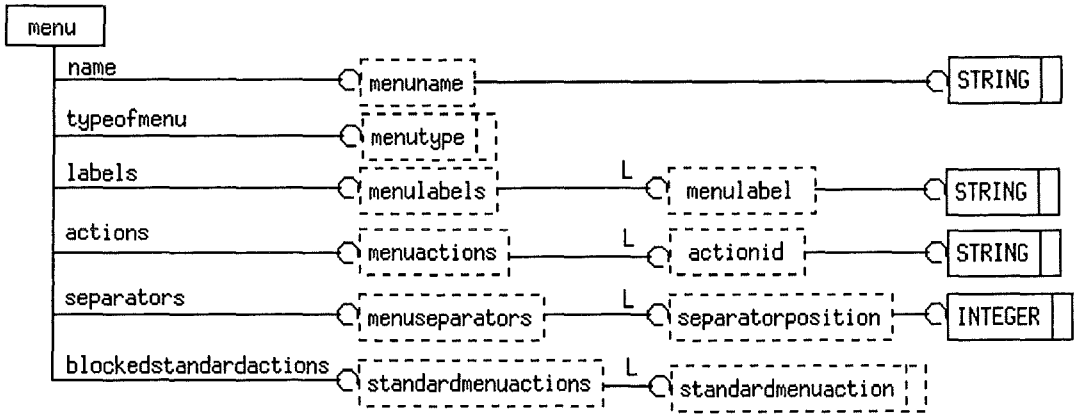


Abb. 45. Definition der Entity Menu

In dem Attribut *Name* wird der Name des Menüs definiert. Mit dem Attribut *TypeOfMenu* (vom Aufzählungstyp ENUMERATION) kann der Typ des Menüs bestimmt werden. Die zulässige Werte sind:

- ***context*** - für ein Kontext-Menü, das zur Verfügung steht, wenn die betroffene Baustein-Ocurrence selektiert ist
- ***background*** - für ein Hintergrund-Menü, das für Baustein-Ocurrences vorgesehen ist, die ein Element repräsentieren, das weiter verfeinert werden kann (in einem hierarchisch strukturierten System-Modell). Das Hintergrund-Menü steht in dem nächsten Untermodell einer Baustein-Ocurrence zur Verfügung, wenn keine Baustein-Ocurrences in dem Untermodell selektiert sind.
- ***submenu*** - für ein Sub-Menü, das ein Teil eines hierarchisch aufgebauten übergeordneten Menü ist (Kontext- bzw. Hintergrund-Menü).

In dem Attribut *Labels* werden die einzelnen Menüpunkte aufgelistet. In dem Attribut *Actions* werden Identifikatoren der Aktionen aufgelistet (*ActionId*), die nach dem Auswahl eines entsprechendem Menüpunktes ausgeführt werden sollen. Die eigentliche Definition der Aktionen erfolgt mit Hilfe des Template-Modells *FunctionModel* des Modells *ImplementationModel* (siehe

Kapitel 5.4.1.4). Wenn in *ActionId* ein Name von einem Menu angegeben ist, wird nach der Auswahl des entsprechenden Menüpunktes keine Aktion aus dem erwähnten *FunctionModel* ausgeführt, sondern es wird das mit dem Namen identifizierte Menü (das als Sub-Menü definiert werden muß) gestartet. Damit lassen sich hierarchisch aufgebaute Menüs definieren.

In dem Attribut *Separators* wird eine Liste von Positionen (*SeparatorPosition*) von Trennungslinien eines Menü definiert. Damit können verwandte Menüpunkte in Gruppen zusammen gefaßt werden. In jedem Kontext- und Hintergrund-Menü sind bestimmte Aktionen als Standardaktionen vorgesehen, die in dem Menü unabhängig von den in dem Attribut *Actions* definierten Aktionen standardmäßig verfügbar sind Das letzte Attribut *BlockedStandardActions* erlaubt eine Auflistung von diesen Standardaktionen, die in einem Menü nicht erreichbar sein sollen. Die vordefinierte Standardaktionen (d.h. die zulässige Werte des Aufzählungstyps *StandardMenuAction*) sind z.B.: ***clear***, ***refresh***, ***connect_input***, ***connect_output***, ***open_submodel***, ***open_supermodel***.

5.4.1.3 Verbindungsregel-Modell

Das Verbindungsregel-Modell (*ConnectionsRulesModel*) einer Baustein-Instanz dient zur Definition von Regeln, mit denen mögliche Verbindungen zwischen Baustein-Ocurrences dieser Baustein-Instanz beschrieben werden. In diesem Modell können vier verschiedene Listen von Regeln definiert werden (Abb. 46).

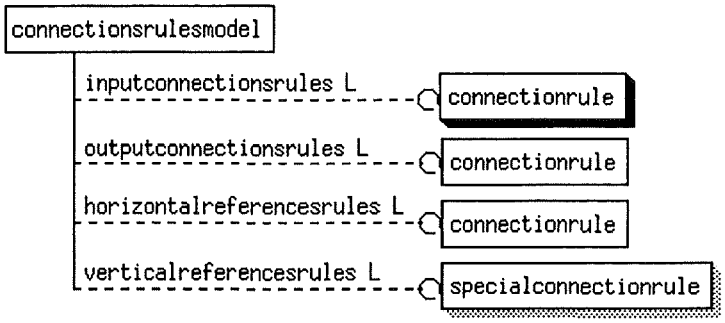


Abb. 46. Definition des Template-Modells RulesModel

In den optionalen Attributen *InputConnectionsRules* und *OutputConnectionsRules* werden Regeln aufgelistet, mit denen bestimmt wird, mit welchen anderen Baustein-Ocurrences die Baustein-Ocurrences der betroffenen Baustein-Instanz Input- bzw. Output-Verbindungen haben können. In dem optionalen Attribut *HorizontalReferencesRules* werden solche Regeln für horizontale Referenz-Verbindungen aufgelistet und in dem optionalen Attribut *VerticalReferencesRules* Regeln für vertikale Referenz-Verbindungen. Für die Beschreibung einer

vertikalen Referenz-Regel wird die Entity *SpecialConnectionRule* verwendet und für die übrigen Regeln - die Entity *ConnectionRule*. Entsprechende Definitionen von beiden Entities werden in der Abb. 47 dargestellt.

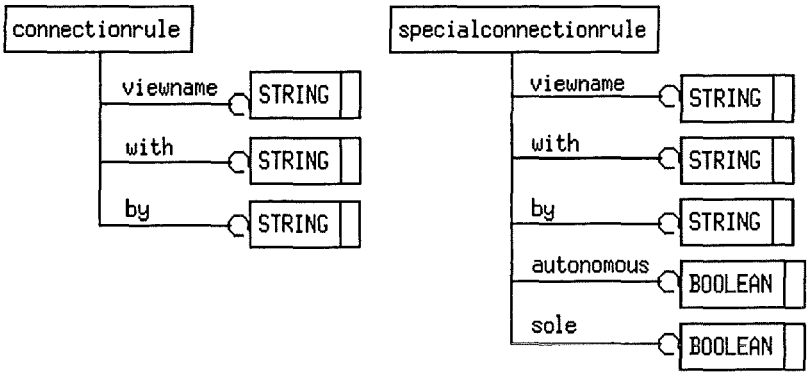


Abb. 47. Definitionen der Entities ConnectionRule und SpecialConnectionRule.

In beiden Entities wird in dem Attribut *ViewName* der Name einer Sicht angegeben, in der die betroffene Regel gelten soll. Damit können unterschiedliche Verbindungen in verschiedenen Sichten verwendet werden können. In dem Attribut *With* wird der Instanz-Name der Baustein-Ocurrences bestimmt, mit denen die betroffene Verbindung möglich ist. In dem Attribut *By* wird der Instanz-Name der Baustein-Ocurrences definiert, die für die Repräsentation der betroffenen Verbindungen verwendet werden. In der Entity *SpecialConnectionRule* stehen zwei zusätzliche Attribute zur Verfügung. Die boolische Werte des Attributes *Autonomous* haben folgende Bedeutung:

- ***true*** - die in dem Attribut *With* definierten Baustein-Ocurrences können in der durch das Attribut *ViewName* bestimmten Sicht direkt vom Systementwickler in dem Untermodell der Baustein-Ocurrence, für die diese vertikale Referenz-Regel gilt, erzeugt werden. Dies bedeutet, daß in einem Werkzeug, mit dem ein System-Modell entwickelt wird, bei dem Entwurf eines Untermodells der Baustein-Ocurrence, für die diese vertikale Referenz-Regel gilt, die in dem Attribut *With* definierte Instanz-Name in einer Auswahlliste mit verfügbaren Baustein-Instanzen erscheinen soll.
- ***false*** - die in dem Attribut *With* definierten Baustein-Ocurrences können nicht in der durch das Attribut *ViewName* bestimmten Sicht direkt von dem Systementwickler in dem Untermodell der Baustein-Ocurrence, für die diese vertikale Referenz-Regel gilt, erzeugt werden. Hier sind zwei Fälle möglich. In dem ersten Fall kann diese Baustein-Ocurrence in einer anderen Sicht direkt erzeugt werden. In dem zweiten Fall kann diese Baustein-Ocurrences in keiner Sicht direkt erzeugt werden und sie werden nur indirekt als Folge anderer Operationen (z.B. Erzeugen anderer Baustein-Ocurrences) bzw. anderer Ereignisse (z.B. während einer Simulation) erzeugt.

Die boolische Werte des Attributes *Sole* haben folgende Bedeutung:

- ***true*** - es kann nur eine Baustein-Occurrence, die in dem Attribut *With* definiert ist, in dem Modell erzeugt werden. Diese Baustein-Occurrence kann jedoch in dem entsprechenden Werkzeug an mehreren Stellen verwendet (und angezeigt) werden, Sie repräsentiert dabei immer die gleiche Baustein-Occurrence. Die Verwendung solcher Baustein-Occurrence wird im Kapitel 7.2.4 vorgestellt.
- ***false*** - es können mehrere Baustein-Occurrences, die in dem Attribut *With* definiert sind, in dem Modell erzeugt werden (Standardwert).

5.4.1.4 Implementations-Modell

Das Implementations-Modell (*ImplementationModel*) einer Baustein-Instanz, das für die Definition von Daten und Funktionen von Baustein-Ocurrences dieser Baustein-Instanz vorgesehen ist, wird in zwei Untermodelle aufgeteilt (Abb. 48). Das Template-Modell *InformationModel* dient zur Definition von Daten, die in den Baustein-Ocurrences gespeichert werden und die auch für die Verwendung in dem Funktions-Modell (*FunctionModel*) zur Verfügung gestellt werden. In dem Informations-Modell einer Baustein-Instanz werden nur entsprechende Namen und Typen von Daten definiert - die konkrete Werte werden in einem entsprechendem Modell jeder Baustein-Ocurrence bestimmt (siehe Kapitel 5.5.3). Das Modell *FunktionModel* dient zur Definition von Funktionen, die beschreiben, wie die in dem Informations-Modell definierte Daten der Baustein-Ocurrences manipuliert werden. Diese Funktionen können auch in der Definition von Funktions-Modellen anderer Baustein-Instanzen verwendet werden. Sie können ebenfalls in dem Sicht-Eingabe-Modell der gleichen Baustein-Instanz benutzt werden, etwa bei der Definition eines Menüs (siehe Kapitel 5.4.1.2). Beide Template-Modelle, *InformationModel* und *FunktionModel*, werden als optionale Attribute der Entity *ImplementationModel* spezifiziert - die Definition dieser Modelle kann daher weggelassen werden.

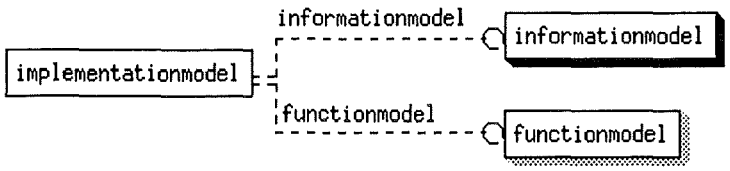


Abb. 48. Definition des Template-Modells ImplementationModel

Informations-Modell

Bei dem Template-Modell *InformationModel* können zwei optionale Listen definiert werden (Abb. 49). In dem Attribut *Elements* werden sogenannte Informationselemente und in dem Attribut *Substitutions* sogenannte Substitutionen aufgelistet.

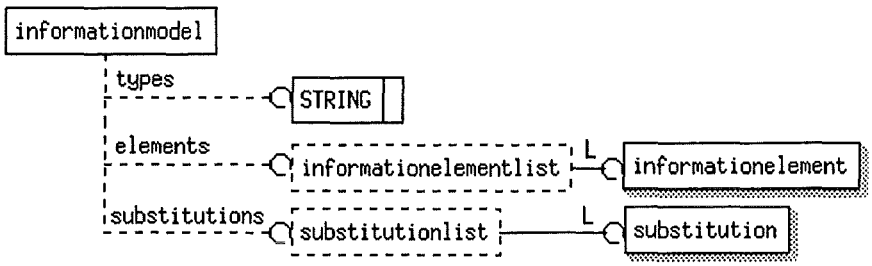


Abb. 49. Definition des Template-Modells InformationModel

Ein Informationselement einer Baustein-Ocurrence dient zum Speichern von Daten. Der Name und der Typ eines Informationselements wird in der Entity *InformationElement* im Informations-Modell der entsprechenden Baustein-Instanz festgelegt. Diese Entity besitzt zwei Attribute: *AccessName* - zur Definition des Namen des Informationselements, mit dem das Element in einem Funktions-Modell identifiziert wird, und *DataType* - zur Definition des Typs des Informationselementes. Der Typ wird in einer textuellen Form (einfacher Typ STRING) mit EXPRESS beschrieben. Es können dabei folgende EXPRESS-Typen verwendet werden:

- einfache Typen - NUMBER, REAL, INTEGER, LOGICAL, BOOLEAN, STRING, BINARY
- Aggregationstypen - ARRAY, LIST, BAG, SET
- alle genannten Typen (Entities und definierte Typen), die in der vollständigen Definition der statischen Templates beinhaltet sind - z.B. geometrische Elemente, Farbe, Menu usw.
- ein beliebiger Typ, der in dem Attribut *Types* mit der EXPRESS-Sprache definiert wird
- ein spezieller Typ *ValueString*

Der spezielle Typ *ValueString* ist für die Definition der Daten vorgesehen, deren Typ nicht festgelegt ist. Ein Beispiel für solche Daten können Attribute eines Tokens in einem Petri Netz sein. Diese Attribute können während der Simulation Daten (bzw. Objekte) unterschiedlicher Typen beinhalten. Bei der Implementation eines Werkzeuges, das das Baustein-Konzept unterstützt, muß es dafür gesorgt werden, daß während des Zugriffes auf solche Daten eine

entsprechende Umsetzung stattfindet (vom Objekt zu der textuellen Repräsentation beim Speichern und von der textuellen Repräsentation zum Objekt beim Lesen)[11].

Die Definition der Entity *InformationElement* zusammen mit der Definition der Entity *Substitution* wird in der Abb. 50 dargestellt.

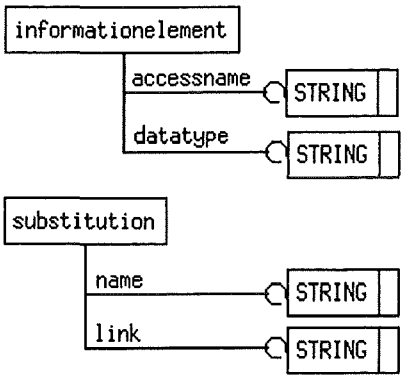


Abb. 50. Definition der Entities InformationElement und Substitution

Mit Hilfe einer Substitution kann ein Attribut eines ausgewählten Ausgabeelements so betrachtet werden, als wäre es ein Informationselement des Informationsmodells. Ein solches Element kann dann in dem Funktions-Modell, genauso wie ein Informationselement, referenziert werden. Damit ist es möglich, dynamische Änderungen der Darstellung einer Baustein-Ocurrence in einer Funktion des Funktions-Modells zu programmieren. In dem Attribut *Name* der Entity *Substitution* wird der Ersatzname des Ausgabeelements definiert. In dem Attribut *Link* wird dieser Element in dem Ausgabe-Sicht-Modell eindeutig identifiziert. Die Syntax der Definition des Attributes *Link* wird mit Hilfe der Wirth Syntax Notation [Wir77] wie folgend dargestellt:

```
Link       = ViewName '[' Nr ']' ':' Attribute { ':' Attribute } .
ViewName   = simple_id .
Nr         = integer_literal .
Attribute  = simple_id .
```

ViewName definiert den Namen der Sicht, *Nr* bestimmt die Position des Ausgabeelementes in der Liste, die in dem Attribut *OutputElements* der Entity *ViewOutput* definiert wird (siehe Abb. 39

[11] Diese Anforderung beschränkt die Auswahl der Implementationssprache für das Werkzeug auf die Sprachen, die eine solche Umsetzung unterstützen. Eine solche Sprache ist Smalltalk-80, die auch für die Implementation des Prototyps des Werkzeuges ausgewählt wurde.

und Abb. 41) und *Attribut* entspricht dem Namen eines Attributes der Entity, die das Ausgabe-Element repräsentiert, oder dem Namen einer untergeordneten Entity. Durch aufeinander folgenden Attributnamen (getrennt durch einen Doppelpunkt) kann ein Pfad zusammengestellt werden, der zu dem gewünschten Attribut führt. Wenn z.B. in der Sicht mit dem Namen ***"Standard"*** eine Substitution für die Farbe eines Ausgabeelements, das die erste Position in der Liste *OutputElements* hat, definiert werden soll (siehe Abb. 41), sieht die Definition des Attributs *Link* folgendermaßen aus:

```
Link        = 'Standard[0]:style:paintColor'
```

Mit Hilfe dieser Substitution können die dynamischen Änderungen der Farbe eines Ausgabeelementes in der Standard-Sicht in dem Funktions-Modell einer Baustein-Instanz programmiert werden. Die Definitionen von *simple_id* und *integer_literal* ist in [ISO92a] enthalten.

Funktions-Modell

Bei dem Template-Modell *FunktionModel* können vier optionale Listen definiert werden (Abb. 51), die für verschiedene Typen der Aktionen, die das Verhalten von Baustein-Ocurrences bestimmen, vorgesehen sind.

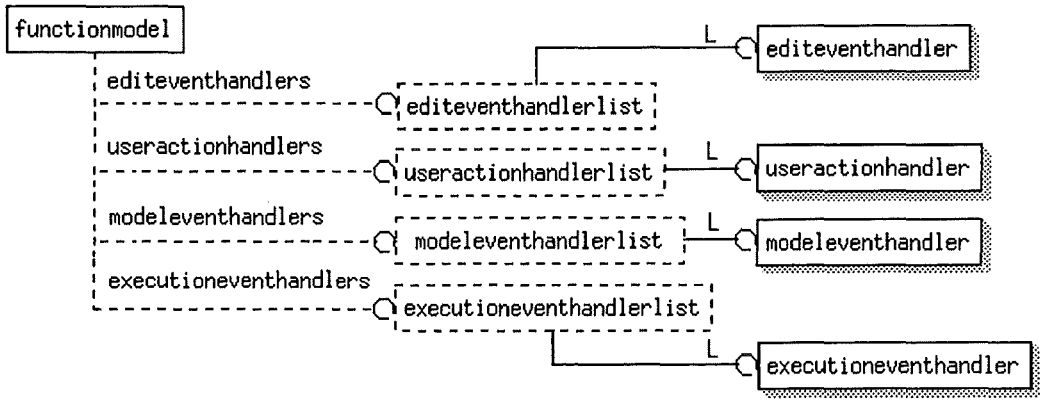


Abb. 51. Definition des Template-Modells FunktionModel

Folgende Attribute stehen zur Verfügung:

- *EditEventHandlers* - zur Definition von Funktionen (Entities *EditEventHandler*), deren Ausführung durch bestimmte vordefinierte Edit-Ereignisse (z.B. Löschen, Erzeugen, Verschieben usw.) verursacht wird

- *UserActionHandlers* - zur Definition von Funktionen (Entities *UserActionHandler*), deren Ausführung durch Benutzereingaben ausgelöst wird. Diese Funktionen werden daher im Eingabe-Sicht-Modell verwendet (z.B. in einem Menü)
- *ModelEventHandlers* - zur Definition von Funktionen (Entities *ModelEventHandler*), deren Ausführung durch bestimmte, vordefinierte Modell-Ereignisse ausgelöst werden, die die Struktur des zu entwickelnden System-Modells ändern (z.B. Erzeugen oder Löschen einer Verbindung). Diese Modell-Ereignisse stimmen in manchen Fällen mit der oben erwähnten Edit-Ereignissen überein. Die Edit-Ereignisse werden jedoch nur als Folge von Edit-Aktionen (also vom Benutzer) generiert. Die Modell-Ereignisse können dagegen auch ohne Beteiligung des Benutzer generiert werden, z.B. als Folge einer Aktion, die während der Ausführung einer anderen Funktion durchgeführt wurde. Damit ist die Möglichkeit geschaffen, die Änderungen des System-Modells im Funktionsmodell einer Baustein-Instanz zu programmieren, was wiederum bedeutet, daß diese Änderungen z.B. während der Simulation von dem System selbst, ohne Beteiligung des Benutzers, durchgeführt werden können.
- *ExecutionEventHandlers* - zur Definition von Funktionen (Entities *ExecutionEventHandler*), deren Ausführung von den übrigen Funktionen des Funktionsmodells initiiert wird. Da eine Funktion dieser Gruppe auch in anderen Funktion dieser Gruppe verwendet werden kann, lassen sich beliebige Hierarchiestrukturen der Funktionen definieren und dadurch auch komplizierte Algorithmen zur Beschreibung des Verhaltens von Baustein-Ocurrences der betroffenen Baustein-Instanz implementieren.

Die Definitionen der Elemente der oben beschrieben Listen werden in der Abb. 52 gezeigt.

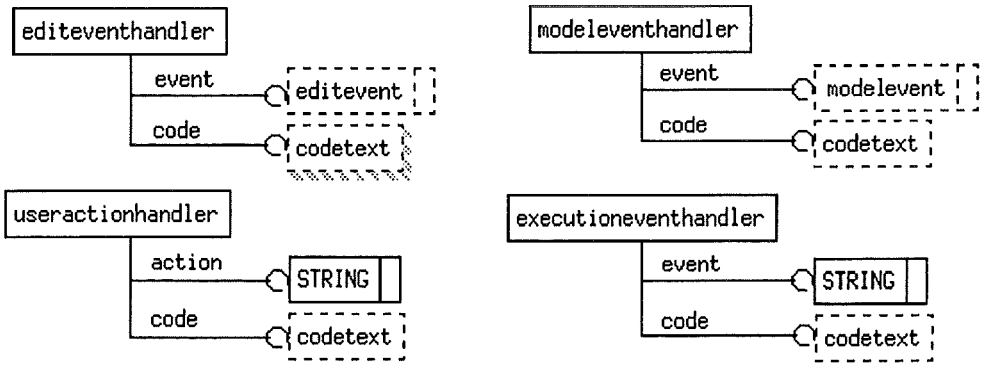


Abb. 52. Definition der Entities EditEventHandler, UserActionHandler, ModelEventHandler und ExecutionEventHandler.

Alle in der Abbildung gezeigten Entities sind ähnlich aufgebaut: sie besitzen jeweils zwei Attribute *Event* bzw. *Action* und *Code*.

In dem Attribut *Event* (Aufzählungstyp *EditEvent*) der Entity *EditEventHandler* wird der Edit-Ereignis genannt, der die Ausführung der betroffenen Funktion auslösen soll. Es sind zahlreiche zulässige Werte für dieses Attribut möglich (wie z.B. ***pre_clear***, ***post_clear***, ***post_create***, ***pre_remove***, ***pre_display***, ***post_display*** usw.). In dem Attribut *Action* der Entity

UserActionHandler wird der Name angegeben, mit dem die betroffene Funktion in dem Eingabe-Sicht-Modell identifiziert wird. In dem Attribut *Event* (Aufzählungstyp *ModelEvent*) der Entity *ModelEventHandler* wird das Modell-Ereignis definiert, das die Ausführung der betroffenen Funktion initiieren soll. Ähnlich wie bei den Edit-Ereignissen werden auch hier zahlreiche zulässige Werte unterstützt (etwa ***pre_connect_input***, ***post_connect_input***, ***created_input_node***, ***removed_input_node*** usw.).

In dem Attribut *Code* wird die entsprechende Funktion (mit Hilfe einer Programmiersprache) definiert. Bei der Implementation eines Werkzeuges, das das Baustein-Konzept unterstützt, muß es dafür gesorgt werden, daß nach jeder Änderung, spätestens direkt vor dem Aufruf der Funktion, eine erneute Übersetzung (recompiling) der Funktion stattfindet[12].

5.4.2 Das statische Node-Template

Die Definition des statischen Node-Template stellt die Abb. 53 dar.

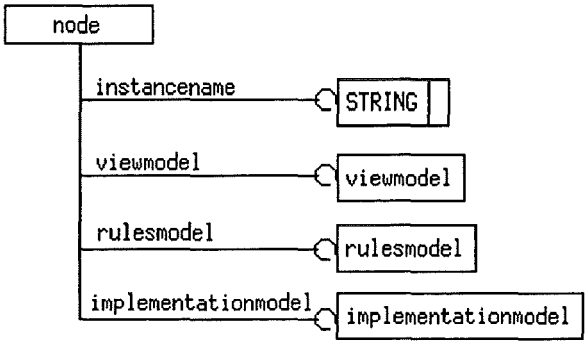


Abb. 53. Definition des statischen Node-Template

In dem Attribut *InstanceNames* wird der Name der von dem Template abgeleiteten Node-Instanz definiert. In dem übrigen Attributen, *ViewModel*, *RulesModel* und *ImplementationModel*, werden entsprechend das Sicht-, Verbindungsregel- und Implementations-Modell spezifiziert. Die Definitionen dieser Template-Modelle sind identisch mit der Definitionen in dem statischen Component-Template (siehe Kapitel 5.4.1.2, 5.4.1.3, 5.4.1.4).

[12] In Smalltalk-80 findet die Übersetzung eines geänderten Code-Abschnittes transparent und unmittelbar vor dem Aufruf statt.

5.4.3 Das statische Arc-Template

Die Definition des statischen Arc-Template stellt die Abb. 54 dar.

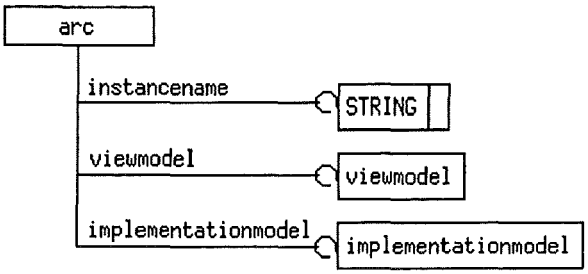


Abb. 54. Definition des statischen Arc-Template

In dem Attribut *InstanceNames* wird der Name der vom Arc-Template abgeleiteten Arc-Instanz definiert. In dem übrigen Attributen, *ViewModel* und *ImplementationModel*, werden entsprechend das Sicht- und Implementations-Modell spezifiziert. Die Definitionen dieser Template-Modelle sind identisch mit den Definitionen im statischen Component-Template (siehe Kapitel 5.4.1.2, 5.4.1.4). Im statischen Arc-Template entfällt das Verbindungsregel-Modell, da die entsprechende Regeln vollständig mit Hilfe von Verbindungsregel-Modellen des Component- und Node-Template definiert werden.

5.5 Die dynamischen Baustein-Templates[13]

Von den dynamischen Baustein-Templates werden Baustein-Ocurrences abgeleitet, die ein Modell des zu entwickelnden Systems bilden. Bei der Ableitung einer Baustein-Ocurrence werden die Verbindungen mit anderen Baustein-Ocurrences, die Position (falls erwünscht) und die Werte eventuell vorhandener Informationselemente definiert. Diese Informationen können auch nach der Ableitung einer Baustein-Ocurrence im Verlauf des Entwicklungsprozesses des Systems geändert werden.

Die dynamischen Baustein-Templates werden aufgrund der Definitionen der Baustein-Instanzen automatisch generiert. Ihre vollständige Struktur ist daher nicht festgelegt. Es gelten jedoch bestimmte Regeln, die bei der Generierung dieser Baustein-Templates berücksichtigt werden:

- für jede Baustein-Instanz wird jeweils ein Baustein-Template generiert, dessen Name gleich dem Instanz-Namen dieser Baustein-Instanz ist.
- in jedem Baustein-Template werden zwei Template-Modelle definiert: ein Verbindungs-Modell und ein Informations-Modell
- die Struktur des Verbindungs-Modells ist in jedem Baustein-Template gleich (siehe Kapitel 5.5.2)
- die Struktur des Informations-Modells ist abhängig von der Definition des Informations-Modells der entsprechenden Baustein-Instanz. (siehe Kapitel 5.5.3)

Die vollständige Präsentation der dynamischen Baustein-Templates wird anhand eines Beispieles durchgeführt. In diesem Beispiel wird angenommen, daß die bereits zur Verfügung stehende Baustein-Instanzen eine Methode definieren, die für das Erstellen von einfachen Petri-Netz-Modellen dient, die nur aus Transitionen, Stellen und Kanten bestehen. Es sind Baustein-Instanzen mit folgenden Instanz-Namen definiert worden:

- *PetriNetComponent* - zur Repräsentation des ganzen Petri-Netz-Modells
- *Transition* - zur Repräsentation von Transitionen
- *Place* - zur Repräsentation von Stellen. In dem Informations-Modell dieser Baustein-Instanz ist ein Informationselement (mit den Namen *Capacity* und von dem Typ *INTEGER*) definiert, das die Kapazität einer Stelle repräsentiert
- *Arc* - zur Darstellung von Kanten

[13] Für die vollständigen Definitionen der dynamischen Baustein-Templates für zwei Testszenarien, die in Kapitel 7 definiert werden, siehe Anhang B und C.

Es wird angenommen, daß im *GeneralViewModel* der Component-Instanz eine Standard-Sicht definiert ist, in der die Baustein-Ocurrences der Baustein-Instanzen Transition und Place individuelle Positionen speichern können (siehe Kapitel 5.4.1.2). In den folgenden Kapiteln werden die aufgrund der Informationen aus dieser Baustein-Instanzen generierte dynamischen Baustein-Templates vorgestellt.

5.5.1 Die Grundstruktur der dynamischen Baustein-Templates

Die Grundstruktur der dynamischen Baustein-Templates ist immer gleich (Abb. 55).

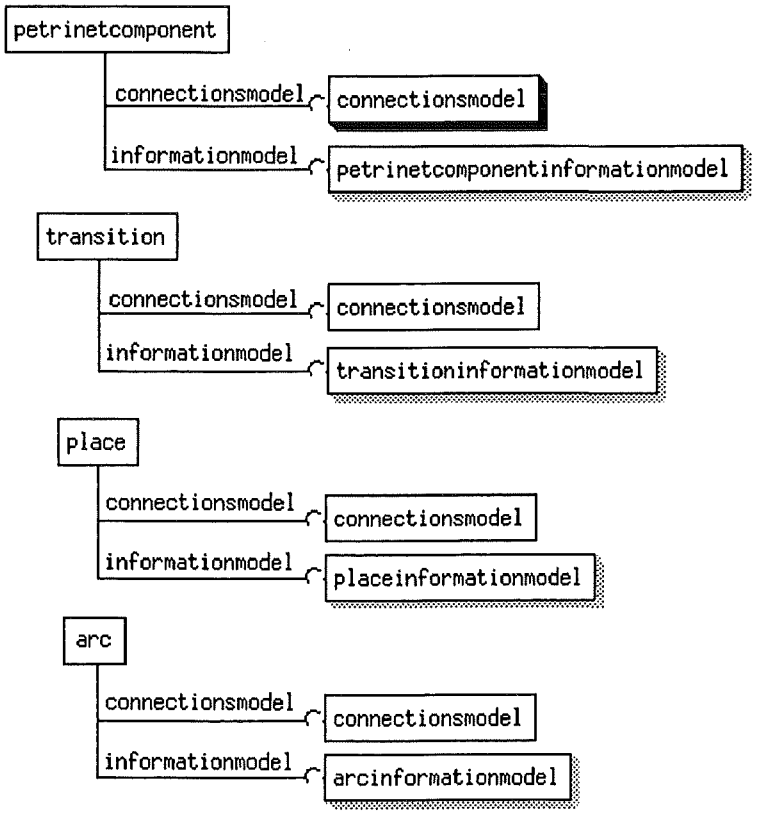


Abb. 55. Die Grundstruktur der dynamischen Baustein-Templates PetriNetComponent, Transition, Place und Arc

Jedes dynamische Baustein-Template besitzt jeweils zwei Attribute: *ConnectionsModel* und *InformationModel*. *ConnectionsModel* ist von dem vordefinierten, nicht änderbaren Typ *ConnectionsModel* und *InformationModel* ist von einem dynamisch generierten Typ, dessen Name durch Zusammensetzen von dem entsprechenden Instanz-Namen und dem Wort "InformationModel" gebildet wird.

5.5.2 Verbindungs-Modell

Die festgelegte Definition des Verbindungs-Modells (*ConnectionsModell*), die für alle dynamische Baustein-Templates gleich ist, wird in der Abb. 56 dargestellt.

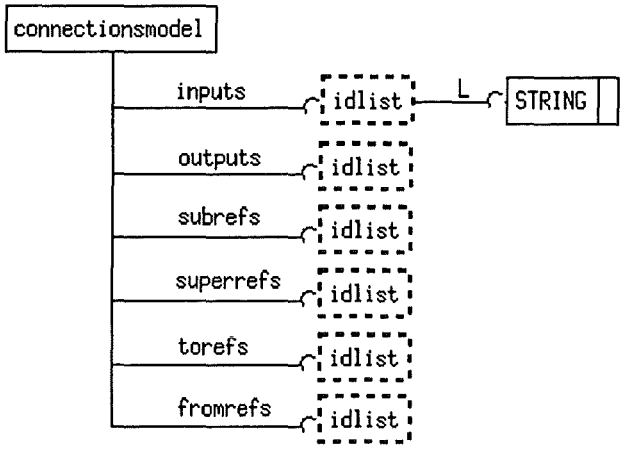


Abb. 56. Definition des Template-Modells ConnectionsModell

In den einzelnen Attributen des Modells werden Listen (*IdList*) definiert, die die verschiedene Verbindungen mit anderen Baustein-Ocurrences bestimmen:

- *Inputs* und *Outputs* - für die Inputs- und Outputs-Verbindungen
- *SubRefs* und *SuperRefs* - für die vertikale Referenz-Verbindungen
- *ToRefs* und *FromRefs* - für die horizontale Referenz Verbindungen

Jedes Element in diesen Listen ist vom einfachen Typ STRING und stellt einen Identifikator der Baustein-Ocurrence dar, die mit der betroffenen Baustein-Ocurrence eine entsprechende Verbindung hat. Diese Identifikatoren, die jede Baustein-Instanz eindeutig identifizieren, werden im Informations-Modell jeder Baustein-Ocurrence definiert.

5.5.3 Varianten des Informations-Modells

Das Informations-Modell einer Baustein-Ocurrence hat keine feste Struktur - es kann in den Baustein-Occurrences unterschiedlicher Baustein-Instanzen auch unterschiedliche Definitionen haben. In der Abb. 57 wird die einfachste Variante dieses Template-Modells dargestellt, die das Informations-Modell der Baustein-Ocurrence der beispielhaften Baustein-Instanz *PetriNetComponent* repräsentiert (diese Variante ist auch für das Template-Modell *ArcInformationModel* gültig).

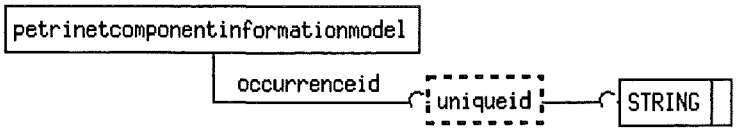


Abb. 57. Die einfachste Variante des Informations-Modells: PetriNetComponentInformationModel

Das Attribut *OcurrenceId* definiert einen eindeutigen Identifikator der Baustein-Ocurrence. Dieses Attribut tritt in jeder Variante des Informationsmodells auf. Da in der Baustein-Instanz *PetriNetComponent* keine Informationselemente definiert wurden und die Baustein-Ocurrence dieser Instanz in keiner Sicht eine individuelle Position speichern soll, sind in dieser Variante des Informations-Modells keine weitere Attribute vorhanden.

Die nächste Variante des Informations-Modells der Baustein-Ocurrences (der beispielhaften Baustein-Instanz *Transition*), die ihre individuelle Positionen speichern können, wird in der Abb. 58 dargestellt.

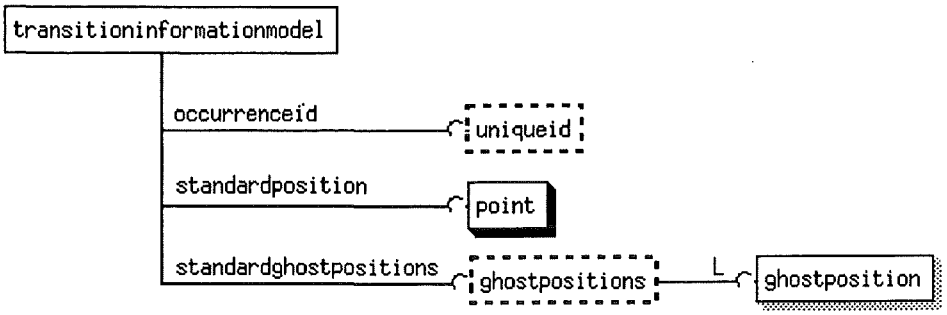


Abb. 58. Die Variante des Informations-Modells mit der individuellen Position: TransitionInformationModel

Für jede Sicht, in der die Baustein-Ocurrences ihre Positionen speichern sollen, werden jeweils zwei Attribute definiert. Das erste Attribut, dessen Name durch Zusammensetzten des Namen der Sicht und des Wortes "Position" gebildet wird, dient zur Definition der eigentlichen Position einer Baustein-Ocurrence. Das zweite Attribut, dessen Name durch Zusammensetzten des Namen der Sicht und des Wortes "GhostPosition" gebildet wird, wird für die Definition der Positionen der sogenannten Ghosts einer Baustein-Ocurrence verwendet. Ein Ghost einer Baustein-Ocurrence repräsentiert diese Baustein-Ocurrence in einem Untermodell einer anderen Baustein-Ocurrence, mit der die betroffene Baustein-Ocurrence verbunden ist (Abb. 59). Die andere Baustein-Ocurrence, mit der die Verbindung besteht, wird als Haunted-Node bezeichnet.

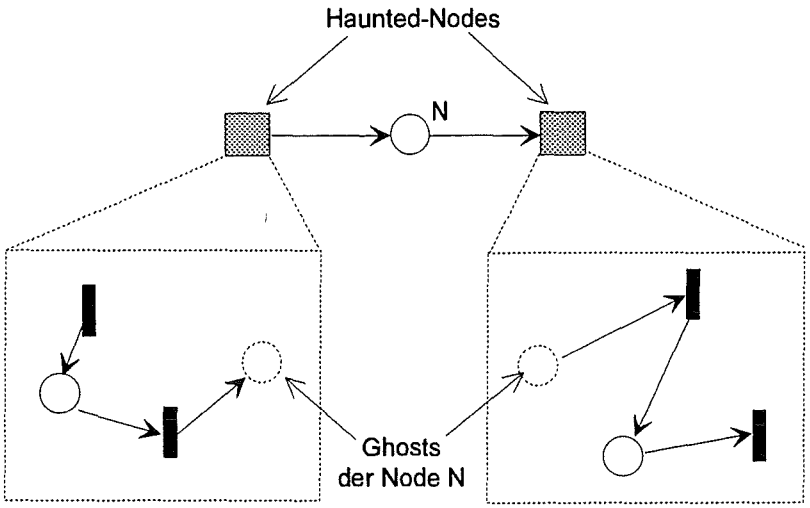


Abb. 59. Ghosts und Haunted-Nodes

Da eine Baustein-Ocurrence mehr als einen Ghost haben kann (sie kann Verbindungen mit mehreren hierarchisch strukturierten Baustein-Ocurrences haben), stellt das Ghost-Position-Attribut eine Liste dar, in der die einzelne Ghost-Positionen (Entity *GhostPosition*) definiert werden. Die Entity *GhostPosition*, deren Definition in der Abb. 60 präsentiert wird, verfügt über zwei Attribute: *HauntedNode* - für den Identifikator einer Baustein-Ocurrences, die eine Haunted-Node ist, und *Position* - für die Ghost-Position in dem Untermodel dieser Baustein-Ocurrence.

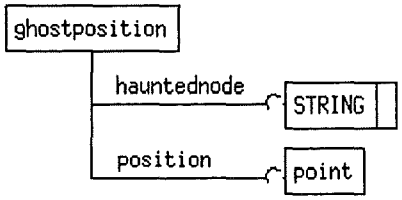


Abb. 60. Definition der Entity GhostPosition

Da in dem hier betrachteten Beispiel keine Hierarchisierung der Petri-Netz-Modelle vorgesehen ist, bleibt die Liste der Ghost-Positionen in jeder Baustein-Occurrence (der Baustein-Instanz *Transition* und *Place*) leer.

Die nächste Variante des Informations-Modells der Baustein-Ocurrences (der beispielhaften Baustein-Instanz *Place*), die ihre individuelle Positionen speichern können und bestimmte Informationselemente besitzen, wird in der Abb. 61 dargestellt.

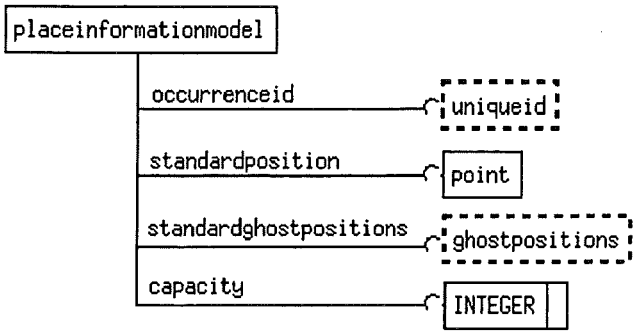


Abb. 61. Die Variante des Informationsmodells mit der individuellen Position und einem Informationselement: PlaceInformationModell

Die Zahl der zusätzlichen Attribute hängt von der Definition des Informations-Modells der entsprechenden Baustein-Instanz ab. In diesem Beispiel wird angenommen, daß in dem Informations-Modell der Baustein-Instanz *Place* nur ein Informationselement für die Repräsentation der Kapazität der Stelle definiert ist. In dem Informations-Modell der Baustein-Ocurrences dieser Baustein-Instanz erscheint daher nur das zusätzliche Attribut *Capacity*.

Die letzte Variante des Informations-Modells entspricht der Situation, in der die Baustein-Ocurrences bestimmte Informationselemente besitzen, aber keine individuelle Position speichern. In diesem Fall entfallen die Attribute, die für die Definition der Positionen dienen. Es werden daher zusätzlich zu dem Attribut für den Identifikator der Baustein-Ocurrence nur die Attribute definiert, welche die Informationselemente der Baustein-Ocurrence repräsentieren.

6. Die software-technische Umsetzung

In diesem Kapitel wird die prototypische Implementierung eines Werkzeuges vorgestellt, welches das in den vorausgegangen Kapiteln vorgestellte Baustein-Konzept realisiert und damit den zweidimensionalen Lebenszyklus eines Systems unterstützt.

Für die Implementation des Werkzeuges wurde die objektorientierte Sprache Smalltalk-80 [GoR89] ausgewählt, die für die prototypische Entwicklung sehr gut geeignet ist. Die dabei verwendete Entwicklungsumgebung (*ObjectWorks 4.0*[14]) ist für verschiedene Plattformen verfügbar und erlaubt plattformunabhängige SmallTalk-Anwendungen zu realisieren. Damit wird die Portabilität des Werkzeug gewährleistet. Die Verwendung von Smalltalk-80 ermöglicht auch die Erfüllung der in den Kapitel 5.4.1.4 formulierten Anforderungen (automatische Umsetzung einer textuellen Beschreibung eines Objektes in die binäre Repräsentation und umgekehrt; automatische Übersetzung eines geänderten Code-Abschnittes).

Die Entwicklungsumgebung *ObjectWorks 4.0* verfügt über zahlreiche vordefinierte Klassen für die Realisierung einer komfortablen Benutzerschnittstelle. Um die Realisierung der Benutzerschnittstelle des Werkzeuges zu beschleunigen wurde zusätzlich die Klassenbibliothek *Mei*[15] verwendet, von der vor allem die Klassen für die Erstellung von Editoren verwendet wurden.

Der in Rahmen dieser Arbeit entwickelte Prototyp stellt mehrere Module der Benutzerschnittstelle zur Verfügung, die dem Entwickler das Definieren der Baustein-Instanzen (Entwicklung und Änderung der Methode) und Erzeugen der Baustein-Occurrences (Entwicklung des Systems) in einer komfortablen Weise ermöglichen. Andere Module des Werkzeuges dienen zur Verwaltung der EXPRESS-Schemas, welche die Definitionen der Methoden und der System-Modelle beinhalten.

Im Rahmen der prototypischen Implementation des Werkzeuges wurden nur die wichtigsten Module entwickelt, mit denen die Realisierbarkeit des Baustein-Konzeptes nachgewiesen werden

---

[14] Die kommerzielle Entwicklungsumgebung ObjectWorks und die Programmiersprache Smalltalk-80 wurde am XEROX Palo Alto Research Center entwickelt

[15] Mei ist eine Smalltalk-Klassenbibliothek für die Erstellung von graphischen Benutzeroberflächen. Sie wurde am Software Engineering Laboratory Inc. entwickelt und wird im Rahmen der GNU General Public License freigegeben.

kann. Die Funktionalität und die Benutzerschnittstellen dieser Module werden in den folgenden Kapiteln näher vorgestellt.

6.1 Funktionalität des Werkzeuges

Das Werkzeug besteht aus zahlreichen Modulen, die für die Realisierung verschiedener Aufgaben vorgesehen sind. Es werden vier Aufgabenbereiche unterschieden (Abb. 62):

1. Verwaltung der Templates
2. Editieren der Baustein-Instanzen
3. Editieren und Simulation des System-Modells
4. Verwaltung der Module des Werkzeuges

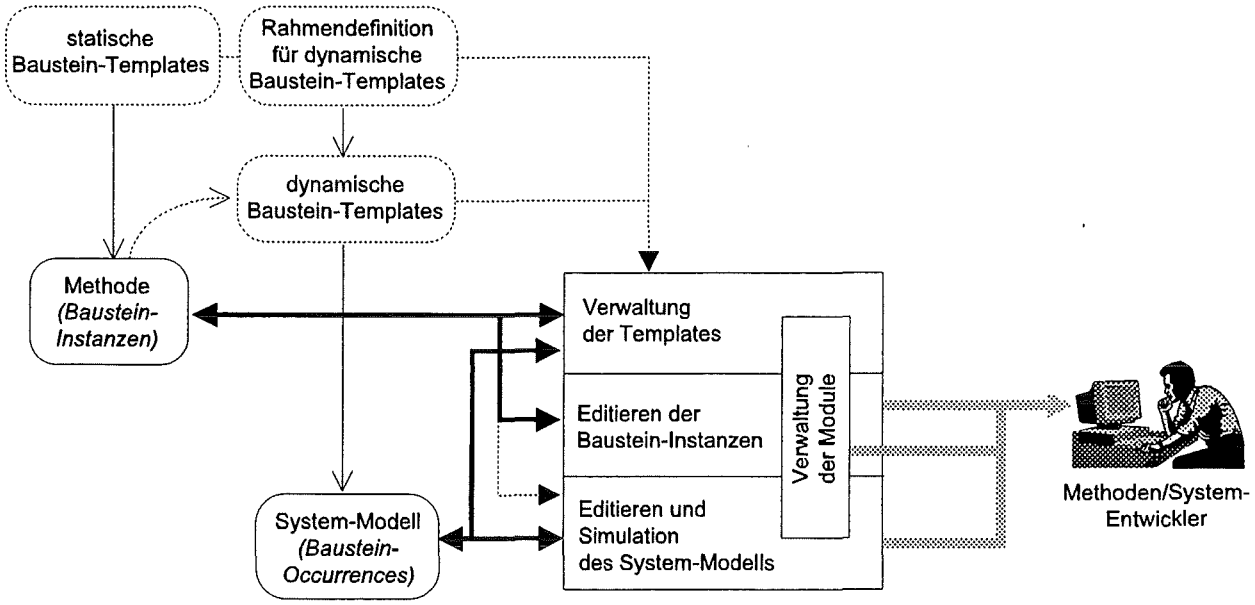


Abb. 62. Aufgabenbereiche des Werkzeuges

In den folgenden Abschnitten werden die einzelne Aufgabenbereiche näher vorgestellt.

6.1.1 Verwaltung der Templates

In diesem Bereich werden Aufgaben realisiert, die mit der Verwaltung der Template-Strukturen verbunden sind. Diese Template-Strukturen stellen die ausgefüllten Baustein-Templates zusammen mit ihrer Definition dar. Es werden zwei Typen der Template-Strukturen differenziert: die ausgefüllten statischen Baustein-Templates, d.h. die Baustein-Instanzen (die eine Methode definieren) und die ausgefüllten dynamischen Baustein-Templates, d.h. die Baustein-Occurrences (die ein System-Modell bilden). Da die Definitionen der statischen und dynamischen Baustein-Templates in gleicher Form (EXPRESS-Definitionen) repräsentiert werden, sind die

Verwaltungsaufgaben für die beiden Typen der Template-Strukturen gleich. Insbesondere umfassen sie die folgenden Aktivitäten:

- Erzeugen, Löschen und Ändern der ausgefüllten bzw. zur Ausfüllung vorgesehenen Baustein-Templates
- Laden und Umsetzen der Definitionen der Baustein-Templates in die interne Repräsentation (Parsen der EXPRESS-Definitionen)
- Umsetzen der internen Repräsentation der Baustein-Templates in die EXPRESS-Definition und Speichern dieser Definition
- Laden und Umsetzen der Beschreibung der ausgefüllten Baustein-Templates in die interne Repräsentation (Parsen der EXPRESS Exchange Structure)
- Umsetzen der internen Repräsentation der ausgefüllten Baustein-Templates in die Beschreibung (EXPRESS Exchange Structure) und Speichern dieser Beschreibung

Zusätzlich werden für die Template-Strukturen, welche die Baustein-Occurrences repräsentieren, folgende Aktivitäten durchgeführt:

- Generierung der Definition der dynamischen Baustein-Templates durch die Vervollständigung der Rahmendefinition aufgrund der Informationen aus den ausgefüllten statischen Baustein-Templates.
- Anpassung der Definition der dynamischen Baustein-Templates nach der Änderung der Informationen in den Baustein-Instanzen.
- Anpassungen der Daten in den ausgefüllten dynamischen Baustein-Templates (d.h. in den Baustein-Occurrences), deren Definition sich geändert hat. Diese Anpassungen werden automatisch oder mit Hilfe des Benutzers durchgeführt[16].

Die Module des Werkzeugs, welche die oben aufgelisteten Aktivitäten realisieren, stellen keine Benutzerschnittstelle zur Verfügung. Sie führen diese Aktivitäten nur auf Anforderungen der Module aus, welche die Aktivitäten der anderen Aufgabenbereiche realisieren.

6.1.2 Editieren der Baustein-Instanzen

Die Module des Werkzeuges, die zum Editieren der Baustein-Instanzen dienen, ermöglichen dem Benutzer:

- neue Baustein-Instanzen (Kopien der statischen Baustein-Templates) zu erzeugen
- in den erzeugten Baustein-Instanzen entsprechende Informationen anzugeben (Ausfüllen der Kopien der statischen Baustein-Templates)

---

[16] Andere Anpassungen (siehe dazu Kapitel 5.2) wurden bei dieser prototypischen Implementation des Werkzeuges nicht realisiert.

- existierende Baustein-Instanzen zu entfernen
- Informationen in den existierenden Baustein-Instanzen zu ändern

Diese Module stellen nur verschiedene (textuelle und graphische) Benutzerschnittstellen zur Verfügung, welche die Anforderungen der Durchführung dieser Aktivitäten vom Benutzer annehmen. Die eigentliche Ausführung dieser Aktivitäten erfolgt (auf Veranlassung von einer Benutzerschnittstelle) in den entsprechenden Template-Strukturen.

6.1.3 Editieren und Simulation des System-Modell

Ein System-Modell wird durch die Menge der Baustein-Occurrences definiert. Diese Baustein-Occurrences stellen die Knoten und Kanten des Graphen dar, der das System-Modell repräsentiert. Die Definition der Struktur dieses Graphen ist auf alle Baustein-Occurrences des System-Modells verteilt. Jede Baustein-Occurrence speichert die Information über Verbindungen zu den anderen Baustein-Occurrences. Das Editieren des System-Modells bedeutet daher, neben dem Erzeugen bzw. Löschen den Baustein-Occurrences, auch Änderungen der Information, die in den einzelnen Baustein-Occurrences gespeichert ist.

Die Module des Werkzeuges, die zum Editieren des System-Modells dienen, ermöglichen daher dem Benutzer:

- neue Baustein-Occurrences (Kopien der dynamischen Baustein-Templates) zu erzeugen
- in den erzeugten Baustein-Occurrences entsprechende Informationen anzugeben (Ausfüllen der Kopien der dynamischen Baustein-Templates)
- existierende Baustein-Occurrences zu entfernen
- Informationen in den existierenden Baustein-Occurrences zu ändern

Ähnlich wie beim Editieren der Baustein-Instanzen stellen diese Module nur verschiedene (textuelle und graphische) Benutzerschnittstellen zur Verfügung, welche die Anforderungen der Durchführung dieser Aktivitäten vom Benutzer annehmen. Die eigentliche Ausführung dieser Aktivitäten erfolgt (auf Veranlassung von einer Benutzerschnittstelle) in den entsprechenden Template-Strukturen.

Welche Baustein-Occurrences erzeugt und welche Informationen in diesen Baustein-Occurrences angegeben werden können, hängt von der Definition der entsprechenden Baustein-Instanzen ab. Die Module zum Editieren des System-Modells müssen daher auch den Zugriff auf die Template-Strukturen haben, welche die Baustein-Instanzen repräsentieren. Die Informationen in diesen Template-Strukturen können jedoch von diesem Modulen nicht geändert werden.

Zusätzlich zu den oben aufgelisteten Aktivitäten können die Module zum Editieren des System-Modells auch die Anforderungen der Durchführung einer Simulation vom Benutzer annehmen. Diese Anforderungen werden an ein Simulationsmodul weitergeleitet. Das Simulationsmodul stellt keine Benutzerschnittstelle zur Verfügung. Es nimmt die Anforderungen, die mit der Simulation verbunden sind, von den Modulen zum Editieren an und führt entsprechende Funktionen aus, die in den Funktions-Modellen der Baustein-Instanzen definiert sind. Das Simulationsmodul muß daher den Zugriff auf die Template-Strukturen haben, welche die Baustein-Instanzen repräsentieren.

6.1.4 Verwaltung der Module

Das Modul des Werkzeuges, das zur Verwaltung der übrigen Module dient, ermöglicht dem Benutzer:

- Erzeugen einer neuen Template-Struktur (für die Definition einer neuen Methode oder eines neuen System-Modells)
- Laden der Informationen in eine Template-Struktur (d.h. Definition einer Methode oder eines System-Modells)
- Speichern der Informationen aus einer Template-Struktur (d.h. Definition einer Methode oder eines System-Modells)
- Entfernen einer existierenden Template-Struktur
- Erzeugen oder Entfernen der Module (bzw. Instanzen der Module) zum Editieren der Baustein-Instanzen und zum Editieren und zur Simulation des System-Modells

Das Verwaltungsmodul kann dabei parallel mehrere Template-Strukturen (Methoden und System-Modelle) und mehrere Module (bzw. Instanzen der Module) zum Editieren der Baustein-Instanzen und zum Editieren und zur Simulation des System-Modells verwalten.

6.2 Module des Werkzeuges

Die einzelnen Module, welche die im vorherigen Kapiteln beschriebenen Aktivitäten durchführen, werden in der Abb. 63 zusammen mit ihren Aufgabenbereichen dargestellt.

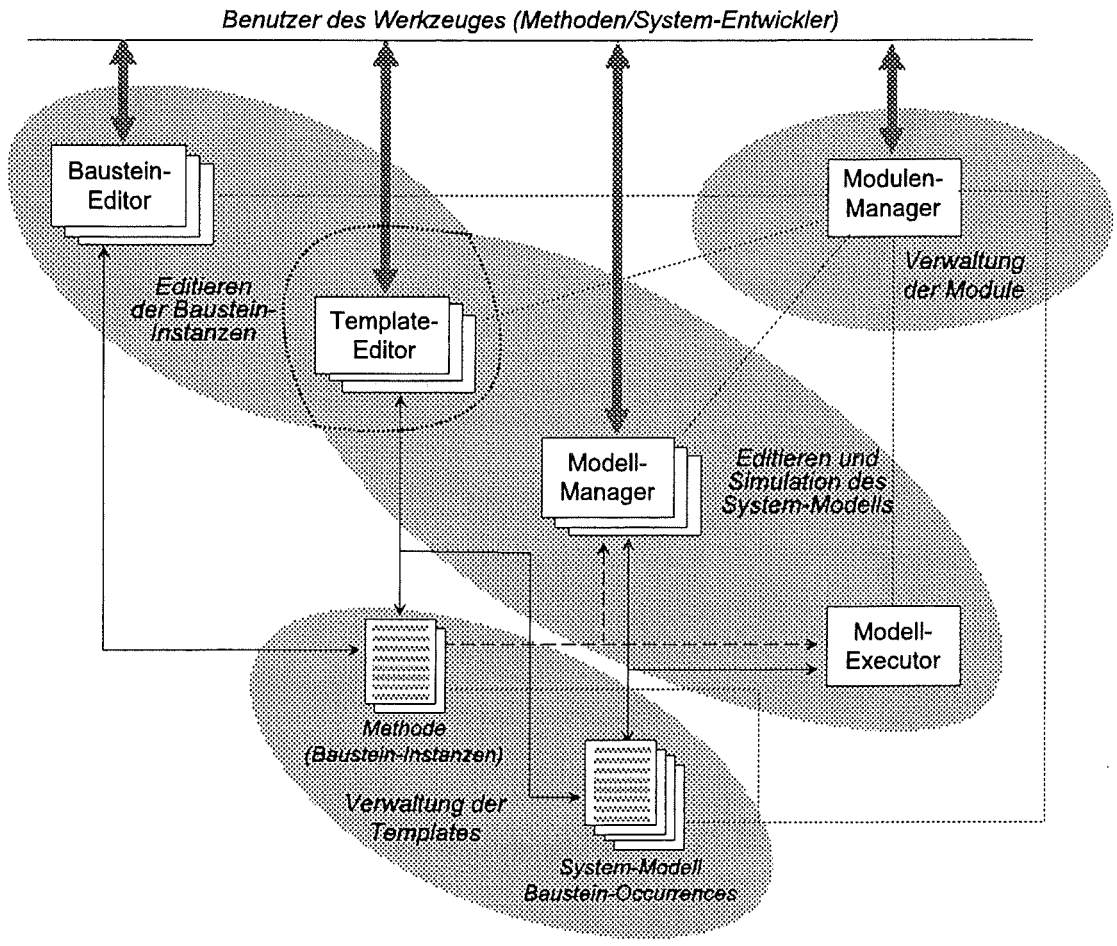


Abb. 63. Module des Werkzeuges und die Aufgabenbereiche

6.2.1 Template-Strukturen

Die Template-Strukturen sind Platzhalter für alle Informationen, die bei der Entwicklung einer Methode und eines System-Modells erzeugt und geändert werden. Sie stellen den übrigen Modulen zahlreiche Funktionen zur Verfügung, welche die Verwaltung dieser Informationen ermöglichen (siehe Kapitel 6.1.1). Die Template-Strukturen stellen die ausgefüllten Baustein-Templates zusammen mit ihrer Definition dar. Die ausgefüllten Baustein-Templates repräsentieren entweder die Baustein-Instanzen (ausgefüllte statische Baustein-Templates) oder die Baustein-Occurrences (ausgefüllte dynamische Baustein-Templates). Die Template-Strukturen stellen daher entweder die Definition einer Methode oder die Definition eines System-Modells dar.

Da die Definition sowohl der statischen als auch der dynamischen Baustein-Templates in der gleichen Form (EXPRESS-Definition) vorliegt, ist die Implementation und die interne Repräsentation für beide Typen der Template-Strukturen gleich. Für die interne Repräsentation der EXPRESS-Definition wurden die wichtigsten EXPRESS-Typen auf entsprechende, speziell dafür entwickelte Smalltalk-Klassen abgebildet. Die Objekte dieser Klassen repräsentieren einzelne Baustein-Templates. Die Instanzvariablen dieser Objekte entsprechen den Attributen der Templates und können weitere Objekte beinhalten, die entweder wiederum andere Unter-Templates (für Entities und Aggregationstypen, wie z.B. LIST oder ARRAY) oder konkrete Werte (für einfache Typen, wie z.B. INTEGER, REAL, BOOLEAN, STRING usw.) repräsentieren. Die Objekte bilden damit eine hierarchische Template-Struktur, deren Wurzel ein Objekt zur Repräsentation eines EXPRESS-Schemas ist.

Jedes Objekt verfügt über zahlreiche Funktionen (Methoden) zur Manipulation, zum Laden und Abspeichern der in den Instanzvariablen enthaltenen Information. Jedes Objekt stellt diese Funktionen seinem übergeordneten Objekt zur Verfügung. Bei der Ausführung dieser Funktionen in einem Objekt werden die zur Verfügung gestellten Funktionen der untergeordneten Objekte rekursiv aufgerufen. Damit können die gleichen Funktionen auf jeder Hierarchieebene einer Template-Struktur aufgerufen werden. Diese Funktionen werden von den übrigen Modulen des Werkzeugs verwendet.

6.2.2 Template-Editor

Ein Template-Editor stellt die gesamte Information eines Baustein-Templates aus einer Template-Struktur in Form einer Liste dar. Diese Liste enthält Namen, Typen und Werte der Attribute dieses Baustein-Templates und der Attribute aller untergeordneten Baustein-Templates. Eine Ausnahme ist die Darstellung der gesamten Templates-Struktur (des Wurzel-Objektes). In dem Fall wird eine Liste aller Baustein-Templates (ohne Attribute) angezeigt[17] (Abb. 64)

[17] Die Abbildungen in diesem Kapitel zeigen einen Zustand bei der Entwicklung einer Petri-Netz-Methode und eines Petri-Netz-Modells, wie in dem Kapitel 7.2 beschrieben.

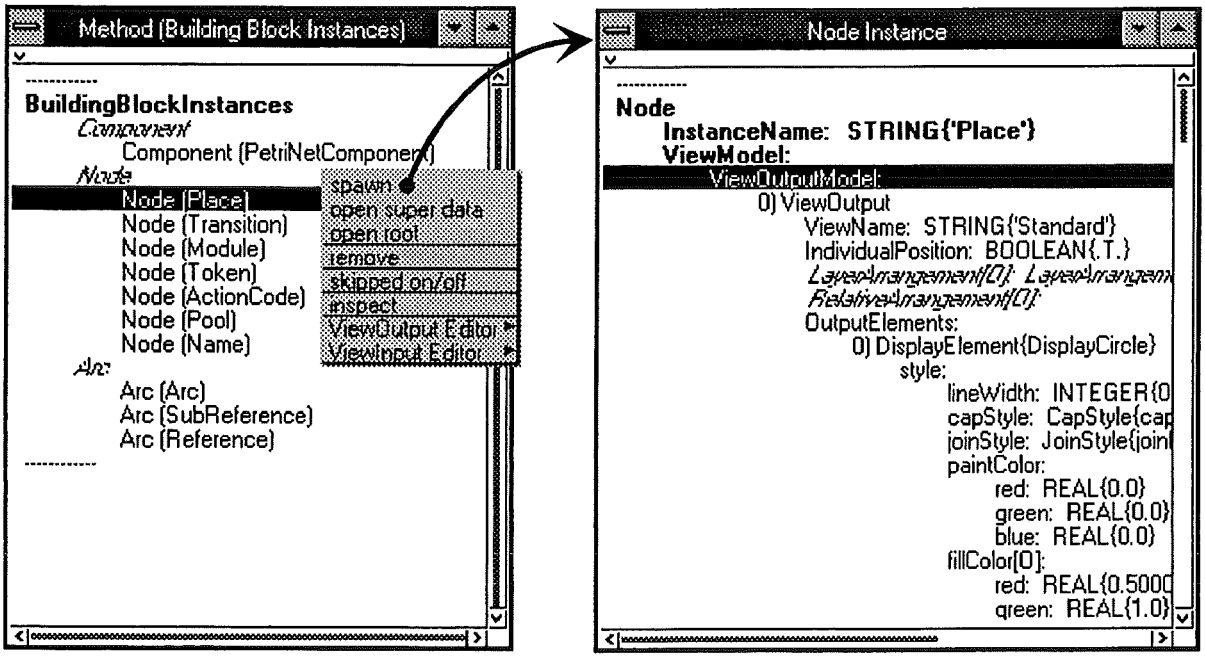


Abb. 64. Die Darstellung der gesamten Template-Struktur und eines Baustein-Templates (für die Definition einer Methode)

Für jedes Attribut in der Liste steht ein Popup-Menü zur Verfügung, dessen Einträge abhängig vom Typ des Attributes sind. Bei den Attributen, die Elemente einer Liste sind, können einzelne Attribute erzeugt oder entfernt werden (*add element*, *remove*). Bei den Attributen, die weitere untergeordnete Templates repräsentieren, steht eine Funktion zur Verfügung (*spawn*), mit der eine neue Liste mit der Attributen dieser untergeordneten Templates dargestellt wird. Bei den Attributen, die einfache Typen repräsentieren, steht dagegen eine Funktion zur Verfügung (*change value*), die ein Fenster zur Eingabe eines entsprechendes Wertes öffnet (Abb. 65).

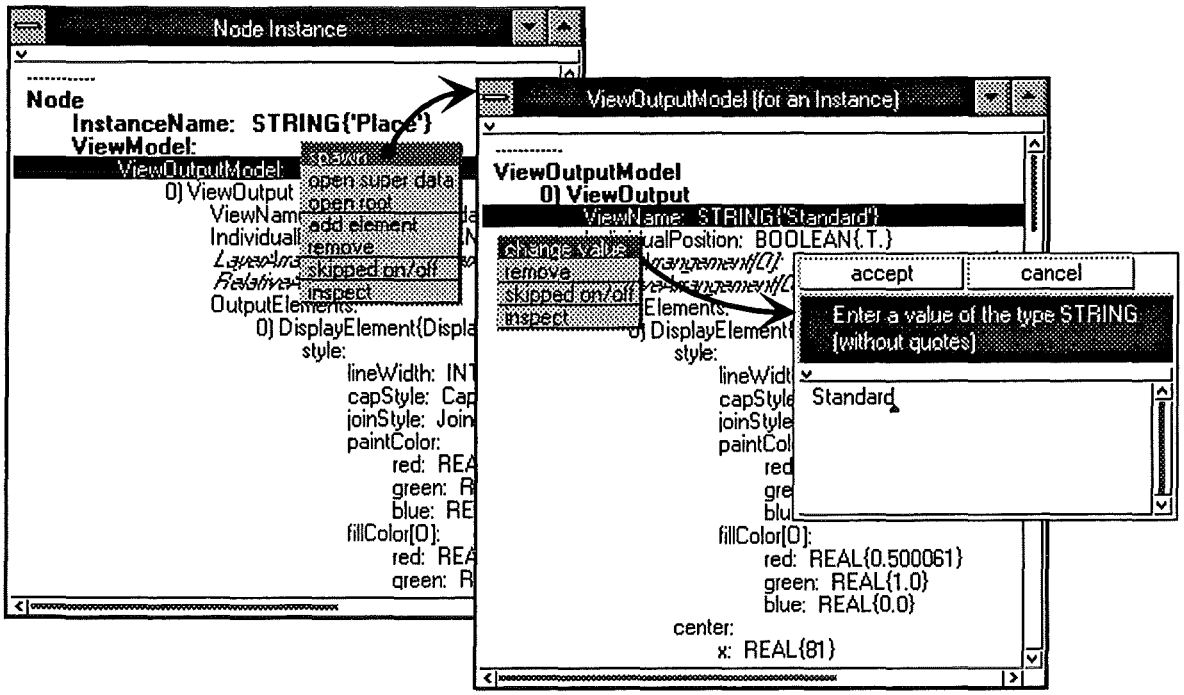


Abb. 65. Darstellung der hierarchisch geordneten Templates und die Fenster zur Eingabe eines Wertes (für die Definition einer Methode)

Die bisher präsentierten Abbildungen stellen die Template-Struktur einer Definition einer Methode (ausgefüllte statische Baustein-Templates, d.h. Baustein-Instanzen) dar. Da die Definition der statischen und der dynamischen Baustein-Templates in der gleichen Form zur Verfügung steht, wird für eine Template-Struktur eines System-Modells (ausgefüllte dynamische Baustein-Templates, d.h. Baustein-Occurrences) die gleiche Darstellung verwendet (Abb. 66)

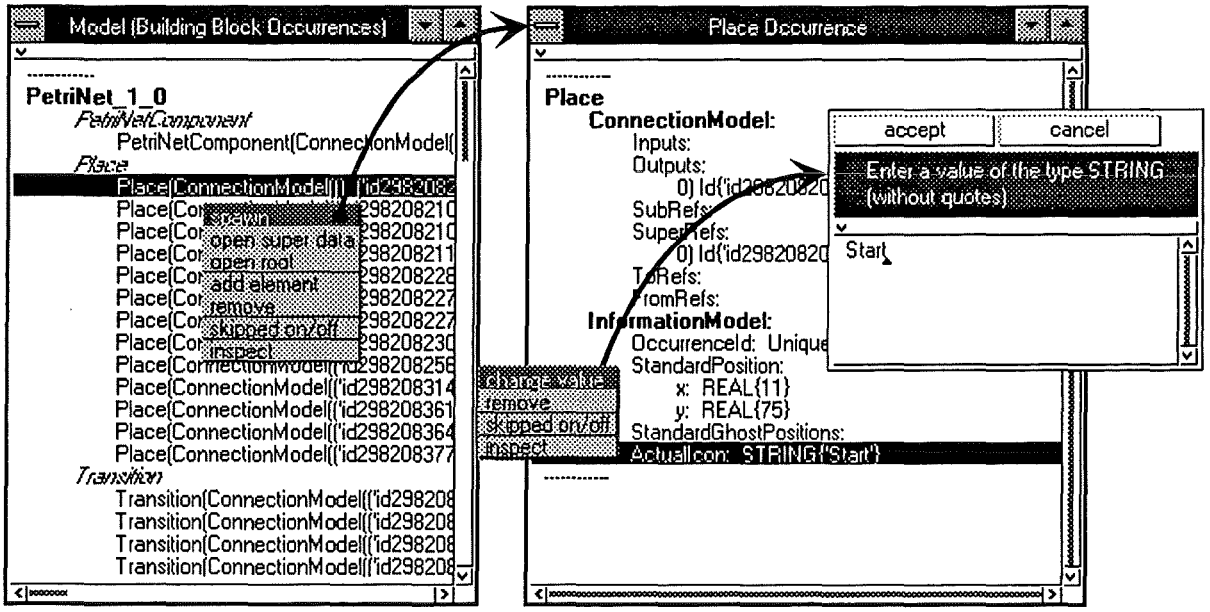


Abb. 66. Die Darstellung der gesamten Template-Struktur und eines Baustein-Templates (für die Definition eines System-Modells)

Der Template-Editor ist vor allem für kleinere Änderungen in einer Template-Struktur (z.B. Hinzufügen oder Löschen eines Attributes, Ändern des Wertes eines Attributes) geeignet. Bei der umfangreichen Änderungen (z.B. Definition des Sicht-Modells in einer Baustein-Instanz) muß eine relativ große Menge an Informationen spezifiziert werden. Dies wäre bei der Benutzung des Template-Editors sehr aufwendig. Aus diesem Grund wurden zusätzliche Module implementiert, welche die Information aus der ganzen Template-Struktur oder nur aus einem ausgewählten Ausschnitt der Template-Struktur (z.B. ein Ausgabe-Sicht-Modell) in einer komfortableren Form präsentieren und die Manipulation dieser Information ermöglichen. Diese Module werden in den nächsten zwei Abschnitten vorgestellt.

6.2.3 Baustein-Editor

Der Baustein-Editor soll mehrere Module zur Verfügung stellen, welche die unterschiedlichen Informationen aus der Template-Struktur der Definition einer Methode in einer angemessenen Form präsentieren und Änderungen dieser Informationen ermöglichen sollen. Im Rahmen der prototypischen Implementation des Werkzeugs wurde ein Modul (Ausgabe-Sicht-Editor) realisiert, das die Definition und Änderungen des Ausgabe-Sicht-Modells einer Baustein-Instanz ermöglicht (Abb. 67).

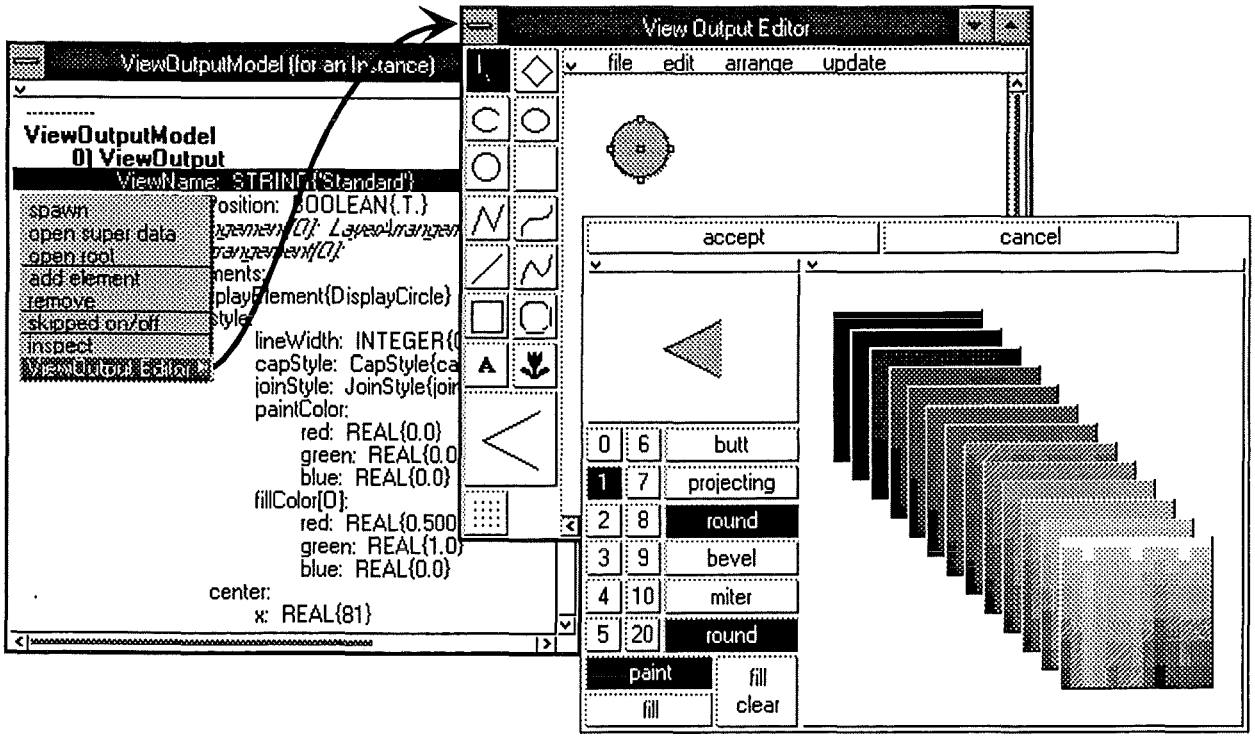


Abb. 67. Der Ausgabe-Sicht-Editor (aufgerufen von einem Template-Editor)

Im Ausgabe-Sicht-Editor können verschiedene geometrische Elemente verwendet werden, um die Darstellung der entsprechenden Baustein-Occurrences (also das Ausgabe-Sicht-Modell der Baustein-Instanz) zu definieren. Jedes Element kann über unterschiedliche graphische Eigenschaften (wie z.B. Dicke der Linie, Farbe der Linie und der Ausfüllung usw.) verfügen. Außer der geometrischen Elementen können auch Elemente zur Darstellung eines Textes oder einer Ikone verwendet werden. Alle diese Elemente können dabei zu einem Element (eine Komposition) zusammengefaßt werden.

Nach der Definition der graphischen Darstellung kann eine Funktion aufgerufen werden, welche diese Darstellung in eine Reihe von Attributen umsetzt und in die entsprechende Template-Struktur überträgt. Falls bereits ein oder mehrere Template-Editoren mit dieser Template-Struktur geöffnet sind, wird die Information in den offenen Template-Editoren entsprechend aktualisiert. Durch die Verwendung des Ausgabe-Sicht-Editors kann die Definition eines Sicht Modells einer Baustein-Instanz wesentlich beschleunigt werden. Bei komplexen Sicht-Modellen kann die Zahl der Attribute, die definiert werden müssen, sehr groß werden. Mit Hilfe des Ausgabe-Sicht-Editors kann diese große Menge an Information aus der graphischen Darstellung automatisch (und damit auch sehr schnell) generiert werden.

Die übrigen Modelle einer Baustein-Instanz (Implementations-Modell, Verbindungsregeln-Modell) erfordern die Definition wesentlich weniger Attributen. Aus diesem Grund wurden keine anderen Module des Baustein-Editors implementiert. Die Definition dieser Attribute kann mit dem Template-Editor in einer akzeptablen Zeit (bei der Verwendung eines prototypischen Werkzeugs) durchgeführt werden.

6.2.4 Modell-Manager

Der Modell-Manager dient zur graphischen Darstellung und Manipulation der Information aus einer Template-Struktur, welche die Definition eines System-Modells repräsentiert (Baustein-Occurrences). Einzelne Baustein-Occurrences werden als Knoten oder Kanten eines Graphen dargestellt (Abb. 68). Die Information über die Darstellung der Knoten und Kanten, über die Elemente der Benutzerschnittstelle, die von den Knoten und Kanten zur Verfügung gestellt wird (z.B. ein Menü), und über die Verbindungsregeln wird von der Template-Struktur abgelesen, welche die Definition der Methode darstellt (Baustein-Instanzen), mit der das System-Modell entwickelt wird. Die Information darüber, welche Arten von Knoten und Kanten (also Baustein-Instanzen) in der Methode zur Verfügung stehen, und welche Sichten in der Methode bei der

Entwicklung eines System-Modells verwendet werden können, wird ebenfalls von dieser Template-Struktur entnommen.

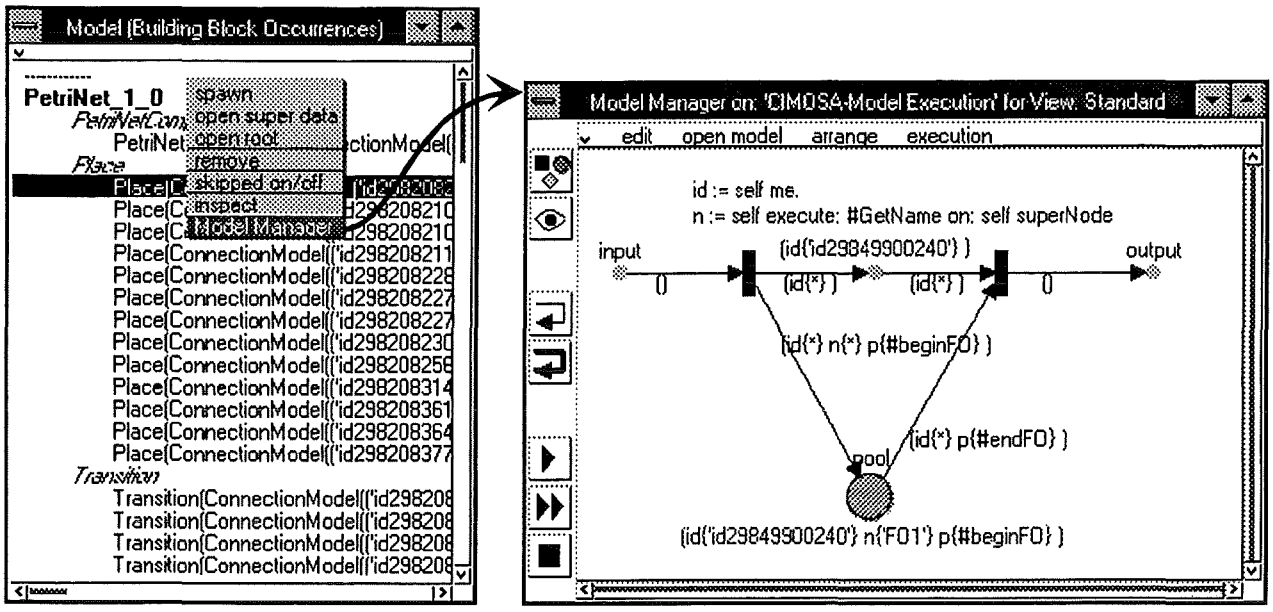


Abb. 68. Der Modell-Manager (aufgerufen von einem Template-Editor)

Diese Informationen werden dabei von der Template-Struktur der Methode jedesmal, wenn sie gebraucht werden, gelesen. Dies bedeutet, daß die Änderungen der Definition der Methode bei der weiteren Entwicklung des System-Modells umgehend berücksichtigt werden. Um die Effizienz zu verbessern, werden die meisten komplexen Informationen (z.B. die graphische Darstellung der Baustein-Occurrences) im Modell-Manager zwischengespeichert und nur dann von der Template-Struktur der Methode neu gelesen, wenn sie geändert wurden.

Der Modell-Manager stellt zahlreiche Funktionen zur Verfügung, die das Editieren des System-Modells ermöglichen. Insbesondere können neue Baustein-Occurrences erzeugt und die existierende Baustein-Occurrences gelöscht werden. Eine Baustein-Occurrence, die einen Knoten des Graphen darstellt, wird durch Auswahl aus einer Liste der verfügbaren Knotenarten erstellt und anschließend im Graph plaziert. Eine Baustein-Occurrence, die eine Kante des Graphen darstellt, wird erzeugt, indem eine Verbindung zwischen existierenden Knoten definiert wird. Wenn unterschiedliche Verbindungen für ein zu verbindendes Paar von Knoten zugelassen sind, wird eine Liste mit verfügbaren Kantenarten präsentiert, aus der ein entsprechendes Element ausgewählt werden kann.

Andere Funktionen des Modell-Managers ermöglichen:

- Verschieben der Elemente

- automatische Ausrichtung der Elemente
- Öffnen eines untergeordneten bzw. übergeordneten Teilmodells (bei hierarchisch strukturierten System-Modellen)
- Umschalten zwischen verschiedenen Sichten - nach dem Umschalten wird das System-Modell neu gezeichnet, wobei für die Darstellung der Baustein-Occurrences die für die gegebene Sicht in den entsprechenden Baustein-Instanzen definierten Ausgabe-Sicht-Modelle verwendet werden
- Starten und Stoppen der Simulation - diese Funktionen generieren entsprechende Ereignisse, die an das Simulationsmodul (Modell-Executor) weitergeleitet werden. Sie haben nur dann eine Wirkung, wenn ein entsprechendes Verhalten in den Baustein-Instanzen der Methode definiert wurde, das die Reaktion auf diese Ereignisse beschreibt.

Nach der Ausführung der Funktionen, welche die Änderung der Information über die Struktur des System-Modells, über die räumliche Anordnung der Elemente oder über die Daten, die in den Baustein-Occurrences gespeichert werden, zur Folge hat, wird diese geänderte Information in die entsprechenden Template-Struktur des System-Modells übertragen. Falls dabei ein oder mehrere Template-Editoren mit dieser Template-Struktur geöffnet sind, wird die Information in diesen Template-Editoren entsprechend aktualisiert.

6.2.5 Modell-Executor

Der Modell-Executor ist eine vereinfachte Variante des Modell-Managers, die keine Benutzeroberfläche zur Verfügung stellt. Ähnlich wie der Modell-Manager hat der Modell-Executor den vollen Zugriff auf die Template-Struktur eines System-Modells. Die Informationen in dieser Template-Struktur können jedoch vom Benutzer nicht geändert werden. Änderungen können aber durch die Ausführung der Funktionen, die in den Funktions-Modellen der Baustein-Instanzen definiert wurden, verursacht werden.

Die Definitionen dieser Funktionen werden aus der Template-Struktur der Methode gelesen und in eine interne Repräsentation umgesetzt. Um die Effizienz zu verbessern, wird diese Umsetzung nur dann durchgeführt, wenn eine entsprechende Definition sich geändert hat. Damit ist es möglich, das Verhalten der Baustein-Occurrences, das durch diese Funktionen beschrieben ist, jederzeit zu ändern. Ein Simulationsalgorithmus, der durch diese Funktionen bestimmt wird, kann somit schrittweise, während der Entwicklung des System-Modells, erweitert und getestet werden.

Im Modell-Executor wird kein Simulationsalgorithmus implementiert. Dieser Algorithmus wird durch die Definition der Funktions-Modelle der Baustein-Instanzen bestimmt. Der Modell-Executor interpretiert lediglich diese Funktionen und führt sie aus. Wenn die Ausführung dieser

Funktionen Änderungen der Informationen über die Struktur des System-Modells, über die räumliche Anordnung der Elemente oder über die Daten, die in den Baustein-Occurrences gespeichert werden, zur Folge hat, wird diese geänderte Information in die entsprechende Template-Struktur des System-Modells übertragen. Falls dabei ein oder mehrere Template-Editoren mit dieser Template-Struktur geöffnet sind, wird die Information in diesen Template-Editoren entsprechend aktualisiert. Die graphische Darstellung des System-Modells im Modell-Manager wird ebenfalls aktualisiert. Damit kann der Verlauf der Simulation im Modell-Manager beobachtet werden.

Da der Modell-Executor und der Modell-Manager als getrennte Module implementiert wurden, die nur durch die entsprechende Template-Struktur (als gemeinsame Datenbasis) verbunden sind, kann die Simulation eines System-Modells auch ohne den Modell-Manager durchgeführt werden. Dies bedeutet, daß der Modell-Manager nach dem Starten der Simulation geschlossen werden kann. Dadurch kann die Simulation beschleunigt werden, weil die graphische Darstellung des System-Modells nicht aktualisiert werden muß. Der Modell-Manager kann aber jederzeit wieder geöffnet werden, um Zwischenergebnisse der Simulation zu beobachten und/oder um die Simulation zu stoppen.

6.2.6 Modul-Manager

Der Modul-Manager ist ein Zentralmodul des Werkzeuges, das die Verwaltung aller anderen Module ermöglicht. In der Benutzeroberfläche dieses Moduls werden zwei Listen zur Verfügung gestellt (Abb. 69). In der ersten Liste werden die Namen aller Methoden angezeigt, deren Definitionen in das Werkzeug geladen bzw. mit dem Werkzeug erstellt wurden. Nach dem Auswählen einer Methode erscheinen in der zweiten Liste die Namen aller mit dieser Methode erstellten System-Modelle, die momentan zur Verfügung stehen. Die Definitionen dieser System-Modelle können ähnlich wie die Definitionen der Methoden in das Werkzeug geladen oder mit dem Werkzeug erstellt werden.

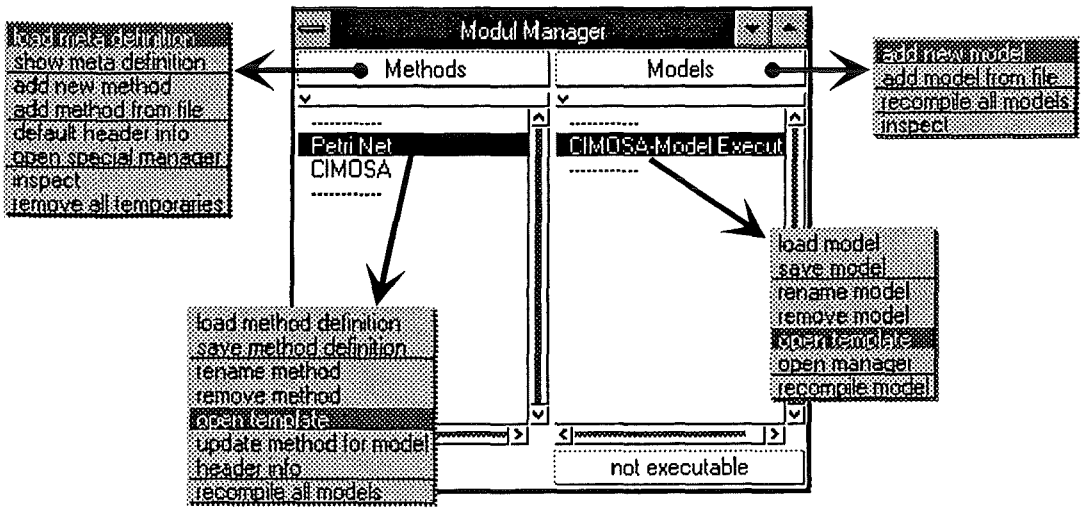


Abb. 69. Der Modul-Manager mit Definitionen der Methoden und der System-Modelle

In der Liste der Methoden werden in einem Menü mehrere Funktionen zur Verfügung gestellt. Diese Funktionen erlauben u.a.:

- Laden, Speichern, Entfernen der Definition einer Methode,
- Umbenennen einer Methode,
- Öffnen eines Template-Editors für die ausgewählte Methode,
- Aktualisieren aller Definitionen der System-Modelle, die mit dieser Methode entwickelt wurden. Diese Funktion soll nach Änderungen der Methode aufgerufen werden, die eine Auswirkung auf die Definition des System-Modells haben (Änderungen der Definition der Daten, die in den Baustein-Occurrences gespeichert werden sollen, Definition einer neuen Baustein-Instanz, Entfernen der Definition einer existierenden Baustein-Instanz).

Die Funktionen, die bei der Liste der System-Modelle zur Verfügung gestellt werden, ermöglichen u.a.:

- Laden, Speichern, oder Entfernen der Definition eines System-Modell
- Umbenennen eines System-Modells
- Öffnen eines Template-Editors für das ausgewählte System-Modell
- Öffnen des Modell-Managers
- Aktualisieren der Definition des ausgewählten System-Modells. Diese Funktion entspricht der Aktualisierungsfunktion in der Liste der Methoden. Die Aktualisierung wird jedoch nur für die Definition des ausgewählten System-Modells durchgeführt.
- Erzeugen oder Entfernen eines Modell-Executors für das ausgewählte System-Modell

Jedes Modul, das von dem Modul-Manager erzeugt wird, wird in einem Modul-Register eingetragen. Die Information aus diesem Register wird verwendet, um die Änderungen der Informationen in einer Template-Struktur allen Modulen, die mit dieser Template-Struktur

verbunden sind, mitteilen zu können. Damit wird es gewährleistet, daß alle geöffnete Module korrekte Informationen über die Definition einer Methode bzw. eines System-Modells präsentieren.

7. Die beispielhaften Einsätze

Für die Validierung der prototypischen Implementation des Werkzeuges werden zwei Testszenarien definiert. In diesen Szenarien werden unterschiedliche Methoden definiert. Mit diesen Methoden werden dann beispielhafte System-Modelle entwickelt, wobei während dieser Entwicklung die dabei eingesetzte Methode erweitert wird. In dem ersten Szenario wird eine vereinfachte CIMOSA-Modellierungsmethode definiert, die das Erstellen von Unternehmensmodellen in zwei CIMOSA-Sichten (Funktions- und Informationssicht) ermöglicht. Mit dieser Methode wird ein beispielhaftes Modell entwickelt, das die Papierherstellung in einem mittelständigen deutschen Unternehmen darstellt. Im zweiten Szenario wird eine Petri-Netz-Methode definiert, die zum Erstellen und Simulieren von Pr/T-Netzen (Prädikat/Transitionsnetzen) dient. Mit dieser Methode wird eine Softwarekomponente entwickelt, die die vereinfachte Funktionalität des Business-Services der CIMOSA-Integrationsplattform implementiert. Die Aufgabe des Business-Services ist die Ausführung der CIMOSA-Modelle. Diese Ausführung wird durch den Einsatz der Petri-Netz-Methode ermöglicht. Beide Szenarien basieren auf den Testszenarien, die für die Validierung des CIMOSA-Konzeptes im Rahmen des ESPRIT Projektes VOICE definiert wurden. Der Verlauf der Definition der Methoden und der Entwicklung der System-Modelle in diesen zwei Szenarien wird in den folgenden Kapiteln vorgestellt.

7.1 CIMOSA-Modellierung

7.1.1 Definition des Szenarios

7.1.1.1 Methode

Es wird eine vereinfachte CIMOSA-Modellierungmethode mit zwei CIMOSA-Sichten, Funktions- und Informationssicht) definiert. Für die Funktionssicht werden folgende Modellierungskonstrukte entwickelt:

- **Domain Process** - zur Beschreibung der Funktionalität innerhalb eines Domains (Geschäftsbereichs)
- **Enterprise Activity** - zur Beschreibung der funktionalen Abläufe innerhalb eines Domain Process
- **Functional Operation**[18] - zur Darstellung der Implementation einer Enterprise Activity
- **Procedural Rules** - zur Beschreibung des Kontrollflusses innerhalb eines Domain Process oder einer Enterprise Activity. Eine Procedural Rule kann als ein Synchronisationspunkt angesehen werden.
- **Event** - zur Beschreibung der Ereignisse, welche die Procedural Rules innerhalb eines Domain Process triggern. Events werden von Enterprise Activities und Functional Operations generiert
- **Start** und **Finish** - zur Darstellung der Eingangs- bzw. Ausgangspunkte der Enterprise Activities bzw. der Ausgangspunkte der Domain Processes

Es werden dabei folgende Regeln berücksichtigt:

⇒ Auf der obersten Hierarchieebene eines Funktionsmodells können nur Domain Processe und Events definiert werden

⇒ Ein Domain Process kann durch Erstellen (im Untermodell dieses Prozesses) von Enterprise Activities, Procedural Rules, Start- und Finish-Konstrukten verfeinert werden

⇒ Eine Enterprise Activity kann durch Erstellen (im Untermodell dieser Aktivität) von Functional Operations, Procedural Rules, Start und Finish verfeinert werden

⇒ Die Enterprise Activities bzw. Functional Operations können nur mittels der Procedural Rules miteinander verbunden werden

⇒ Domain Processes können nur durch Events verbunden werden, wobei die Output-Verbindungen zu den Events als Control Outputs bezeichnet werden

⇒ Start bzw. Finish kann nur mit einer Procedural Rule als Input bzw. als Output verbunden werden

⇒ Ein Event kann innerhalb eines Domain Process Output-Verbindungen mit Procedural Rules haben

⇒ Ein Event kann innerhalb eines Domain Process oder einer Enterprise Function eine Input-Verbindung mit einer Enterprise Activity bzw. einer Functional Operation haben

⇒ Jede Enterprise Function kann benannt werden

---

[18] Für die Domain Processes, Enterprise Activities und Functional Operations wird auch der Sammelbegriff *Enterprise Function* verwendet

Für die Informationssicht werden folgende Modellierungskonstrukte entwickelt:

- **Object View** - zur Definition der Daten
- **Information Element** - zur Verfeinerung der Definition der Daten. Information Elements können als Attribute der Object Views angesehen werden

Es werden dabei folgende Regeln berücksichtigt:

⇒ Eine Object View kann mehrere Attribute haben.

⇒ Ein Information Element kann in mehreren Object Views als Attribut auftreten

Die Methode soll zusätzlich ermöglichen, Querverbindungen zwischen den zwei CIMOSA-Sichten (Funktions- und Informationssicht) zu erstellen. Mit diesen Querverbindungen kann ermittelt werden, welche Object Views in welchen Enterprise Functions verwendet werden. Zu den Querverbindungen gehören:

- **Function Input** und **Function Output** - zur Definition der Information, die in einer Enterprise Function verwendet bzw. erzeugt wird.
- **Control Input** - zur Definition der Steuerungsinformation, die in einer Enterprise Function verwendet wird. **Control Output** wird in der Funktionssicht mit einem Event definiert.
- **Resource Input** und **Resource Output** - zur Definition der Information über die Betriebsmittel, die in der Enterprise Function verwendet bzw. generiert werden

Es werden dabei folgende Regeln berücksichtigt:

⇒ Eine Enterprise Function kann durch Querverbindungen mit mehreren Object Views verbunden werden

⇒ Eine Object View kann durch Querverbindungen mit mehreren Enterprise Functions verbunden werden

⇒ Eine Enterprise Function kann mehrere Querverbindungen vom gleichen Typ (z.B. mehrere Function-Input-Verbindungen) haben

7.1.1.2 System-Modell

Mit der oben definierten Methode wird ein System-Modell entwickelt, das einen Ausschnitt eines umfangreichen Unternehmensmodells darstellt. Dieses Unternehmensmodell wurde im Rahmen einer Kooperation mit dem Produktionswirtschaftlichen Zentrum (PWZ) der Fachhochschule Offenburg entwickelt. Es beschreibt die Geschäftsprozesse, Informationen und Betriebsmittel bei der Papierherstellung in einem mittelständischen deutschen Unternehmen [Neu95]. In dieser Arbeit werden nur Teilmodelle aus zwei Sichten, der Funktions- und der Informationssicht, erstellt. In der Funktionssicht wird ein Domain Process *Herstellen* modelliert. Es werden darin entsprechende Enterprise Activities und Functional Operations definiert, welche die Funktionalität dieses Domain Process beschreiben. In der Informationssicht werden die benötigten Object Views

und Information Elements definiert, die im Domain Process *Herstellen* verwendet werden. Anschließend werden die Querverbindungen zwischen der Funktions- und Informationssicht der erstellt, welche die Object Views und die Enterprise Functions miteinander verbinden.

7.1.1.3 Entwicklung

Die Entwicklung der CIMOSA-Modellierungsmethode und des CIMOSA-Modells wird in sechs Schritten (drei Schritte der Methodendefinition und drei Schritte der Systementwicklung) realisiert. Diese Schritte bilden ein Entwicklungspfad, in dem die Definition der Methode und die Entwicklung des System-Modells stark verflochten sind. Nach jedem Schritt der Methodendefinition wird ein Prototyp des System-Modells entwickelt. Es werden insgesamt drei aufeinander folgende Prototypen entwickelt (Abb. 70).

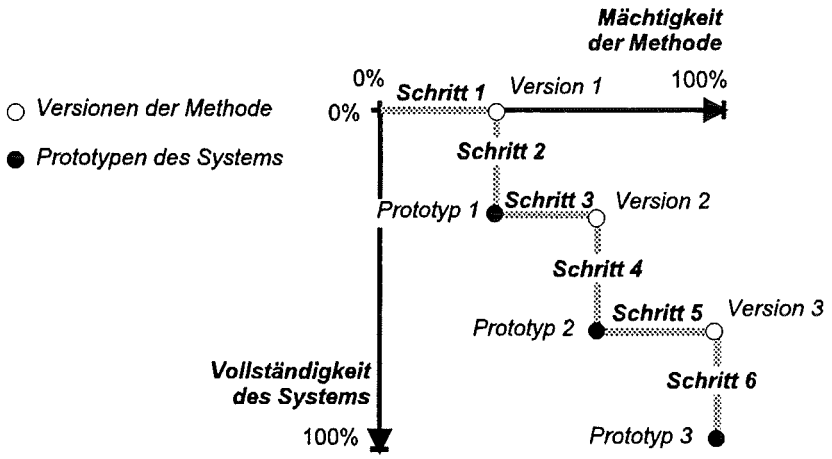


Abb. 70. Entwicklungspfad des ersten Szenarios

Die einzelnen Schritte werden im weiteren näher beschrieben.

7.1.2 Schritt 1: Definition der Funktions-Sicht

In dem ersten Schritt werden die Modellierungskonstrukte der CIMOSA-Funktionssicht definiert. Die CIMOSA-Funktionssicht wird auf zwei Werkzeug-Sichten (die im weiteren nur als Sichten bezeichnet werden) abgebildet:

- *Funktion*-Sicht zur Definition der CIMOSA-Funktionsmodelle auf einer Hierarchieebene
- *FunctionTree*-Sicht zur Darstellung der Hierarchie der CIMOSA-Funktionsmodelle. Diese Sicht wird auch zur Navigation innerhalb der Hierarchie verwendet.

Die Definition der Konstrukte wird in einer vereinfachten Form (Tabellen) präsentiert, in der nur die wichtigsten Informationen dargestellt werden. Die Tabellen entsprechen den ausgefüllten statischen Baustein-Templates. Sie stellen daher die definierten Baustein-Instanzen der CIMOSA-

Modellierungsmethode dar. Zum besseren Verständnis enthalten die Tabellen zum größten Teil statt der genauer Definition nur die Beschreibung der Informationen aus den Templates.

Die Realisierung der im Kapitel 7.1.1.1 aufgelisteten Konstrukte der CIMOSA-Funktionssicht wird in den folgenden Tabellen näher erläutert.

Als erstes muß eine Component-Instanz definiert werden, um bei der Entwicklung eines Modells das Erzeugen einer Component-Occurrence zu ermöglichen, die als Wurzel des gesamten Modells dienen soll.

Component-Instanz für *CIMOSAModel*

Attribute		Werte bzw. Beschreibung der Werte der Attribute
InstanceName		*CIMOSAModel*
GeneralViewModel		Es werden zwei Sichten vorgesehen: • *Function* - eine horizontale Sicht zur Darstellung einer Hierarchieebene • *FunktionTree* - eine vertikale Sicht zur Darstellung der gesamten Hierarchie des Funktionsmodells
ViewModel[19]	ViewOutputModel	In der *Function*-Sicht: • nicht sichtbar In der *FunctionTree*-Sicht: 
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)[20]
RulesModel	InputConnectionRules	keine (in beiden Sichten)
	OutputConnection-Rules	keine (in beiden Sichten)
	HorizontalReferences-Rules	keine (in beiden Sichten)
	VerticalReferences-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *DomainProcess* durch *SubReference* • mit *Event* durch *SubReference* • mit *Name* durch *SubReference* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
Implementation-Model	InformationModel	kein
	FunctionModel	kein

Tab. 2. Die Component-Instanz "CIMOSAModel"

Die Definition der im *RulesModel* genannten Baustein-Instanzen wird in den folgenden Tabellen dargestellt. *DomainProcess* und *Event* werden als Node-Instanzen definiert. *SubReference* wird

---

[19] Bei der Component-Instanz und bei allen Node-Instanzen wird angenommen, daß die entsprechenden Baustein-Occurrences ihre Positionen speichern sollen, wenn sie in der betroffenen Sicht sichtbar sind.

[20] Das in dieser Arbeit präsentierte Werkzeug stellt für alle Baustein-Occurrences bestimmte, vordefinierte Eingabe-Elemente (Menü, Tastenkürzel) zur Verfügung. Bei der Definition des Sicht-Modells werden nur die zusätlichen, für die Methode spezifischen Eingabe-Elemente bzw. Erweiterungen der existierenden Eingabe-Elemente angegeben.

als eine Arc-Instanz definiert. Zur Verbindung der *DomainProcesse* mit *Events* werden die Arc-Instanzen *EventConnection* und *COConnection* (für Control Output) definiert. Um die *DomainProcesse* und *Events* in dem Modell benennen zu können wird eine zusätzliche Node-Instanz *Name*[21] definiert. Zur Verbindung der Baustein-Occurrence *Name* mit den anderen Baustein-Occurrences wird eine Baustein-Occurrence der Baustein-Instanz *Reference* verwendet. *Reference* wird als eine Arc-Instanz definiert.

Node-Instanz für *DomainProcess*

Attribute		Werte bzw. Beschreibung der Werte der Attribute
InstanceName		*DomainProcess*
ViewModel	ViewOutputModel	In der *Function*-Sicht:  In der *FunctionTree*-Sicht: 
	ViewInputModel	in der *Function*-Sicht: • ein zusätzlicher Menüeintrag: "edit name" in der *FunctionTree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	mögliche Verbindungen[22] in der *Function*-Sicht: • mit *Event* durch *EventConnection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *Event* durch *COConnection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	HorizontalReferences-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *Name* durch *Reference* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	VerticalReferences-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *EnterpriseActivity* durch *SubReference* • mit *ProceduralRule* durch *SubReference* • mit *Finish* durch *SubReference* • mit *Name* durch *SubReference* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
ImplementationModel	InformationModel	kein
	FunctionModel	Funktionen für: • Editieren des Namens (für den Menüeintrag "edit name")

Tab. 3. Die Node-Instanz "DomainProcess"

[21] *Name* wird auch zur Darstellung des Namens anderer Elemente verwendet.

[22] Die im RulesModel aufgelisteten möglichen Verbindungen bestimmen, welche Verbindungen von diesem Knoten augehend in der angegebenen Sicht erzeugt werden können. Sie bestimmen nicht, welche Verbindungen angezeigt werden. Das wird durch entsprechende Definition des Sicht-Models der Arc-Instanzen bestimmt. Die Beschreibung "keine mögliche Verbindungen" schließt daher nicht aus, daß in der betroffenen Sicht eine Verbindung angezeigt werden kann.

Node-Instanz für *Event*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Event*
ViewModel	ViewOutputModel	In der *Function*-Sicht:  In der *FunctionTree*-Sicht: • nicht sichtbar
	ViewInputModel	in der *Function*-Sicht: • ein zusätzlicher Menüeintrag: "edit name" in der *FunctionTree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Function*-Sicht: • mit *DomainProcess* durch *COConnection* • mit *EnterpriseActivity* durch *COConnection* • mit *FunctionalOperation* durch *COConnection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *DomainProcess* durch *EventConnection* • mit *ProceduralRule* durch *EventConnection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	HorizontalReferences-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *Name* durch *Reference* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	VerticalReferences-Rules	keine in beiden Sichten
ImplementationModel	InformationModel	kein
	FunctionModel	Funktionen für: • Editieren des Namens (für den Menüeintrag "edit name")

Tab. 4. Die Node-Instanz "Event"

Node-Instanz für *Name*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Name*
ViewModel	ViewOutputModel	In der *Function*-Sicht: • Text abhängig vom Information Element *NameStr* In der *FunctionTree*-Sicht: • Text abhängig vom Information Element *NameStr*
	ViewInputModel	in der *Function*-Sicht: • ein zusätzlicher Menüeintrag: "edit name" in der *FunctionTree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	keine (in beiden Sichten)
	OutputConnection-Rules	keine (in beiden Sichten)
	HorizontalReferences-Rules	keine[23] (in beiden Sichten)
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	Information Element *NameStr* vom Typ STRING
	FunctionModel	Funktionen für: • Editieren des Namens (für den Menüeintrag "edit name")

Tab. 5. Die Node-Instanz "Name"

Arc-Instanz für *EventConnection*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*EventConnection*
ViewModel	ViewOutputModel	In der *Function*-Sicht: → In der *FunctionTree*-Sicht: • nicht sichtbar
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 6. Die Arc-Instanz "EventConnection"

Arc-Instanz für *COConnection*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*COConnection*
ViewModel	ViewOutputModel	In der *Function*-Sicht: CO → In der *FunctionTree*-Sicht: • nicht sichtbar
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 7. Die Arc-Instanz "COConnection"

[23] Es werden keine möglichen horizontalen Verbindungen angegeben, weil sie von den anderen Knoten ausgehend erzeugt werden.

Arc-Instanz für *SubReference*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*SubReference*
ViewModel	ViewOutputModel	In der *Function*-Sicht: • nicht sichtbar In der *FunctionTree*-Sicht: ———
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 8. Die Arc-Instanz "SubReference"

Arc-Instanz für *Reference*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Reference*
ViewModel	ViewOutputModel	nicht sichtbar (in beiden Sichten)
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 9. Die Arc-Instanz "Reference"

Um die Verfeinerung der *DomainProcesse* zu ermöglichen, müssen weitere Baustein-Instanzen definiert werden. Dies sind: *EnterpriseActivity*, *FunctionalOperation*, *ProceduralRule*, *Start* und *Finish*. Zur Verbindung dieser Elemente wird die Arc-Instanz *Connection* definiert

Node-Instanz für *EnterpriseActivity*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*EnterpriseActivity*
ViewModel	ViewOutputModel	In der *Function*-Sicht:  In der *FunctionTree*-Sicht: 
	ViewInputModel	in der *Function*-Sicht: • ein zusätzlicher Menüeintrag: "edit name" in der *FunctionTree*-Sicht: • keine zusätzlichen Eingabeelementе
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Function*-Sicht: • mit *ProceduralRule* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *Event* durch *COConnection* • mit *ProceduralRule* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	HorizontalReferences-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *Name* durch *Reference* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	VerticalReferences-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *FunctionalOperation* durch *SubReference* • mit *ProceduralRule* durch *SubReference* • mit *Start* durch *SubReference* • mit *Finish* durch *SubReference* • mit *Name* durch *SubReference* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
ImplementationModel	InformationModel	kein
	FunctionModel	Funktionen für: • Editieren des Namens (für den Menüeintrag "edit name")

Tab. 10. Die Node-Instanz "EnterpriseActivity"

Node-Instanz für *FunctionalOperation*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*FunctionalOperation*
ViewModel	ViewOutputModel	In der *Function*-Sicht:  In der *FunctionTree*-Sicht: 
	ViewInputModel	in der *Function*-Sicht: • ein zusätzlicher Menüeintrag: "edit name" in der *FunctionTree*-Sicht: • keine zusätzlichen Eingabeelementе
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Function*-Sicht: • mit *ProceduralRule* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *Event* durch *COConnection* • mit *ProceduralRule* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	HorizontalReferences-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *Name* durch *Reference* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	Funktionen für: • Editieren des Namens (für den Menüeintrag "edit name")

Tab. 11. Die Node-Instanz "FunctionalOperation"

Node-Instanz für *ProceduralRule*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*ProceduralRule*
ViewModel	ViewOutputModel	In der *Function*-Sicht:  In der *FunctionTree*-Sicht: • nicht sichtbar
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Function*-Sicht: • mit *EnterpriseActivity* durch *Connection* • mit *FunctionalOperation* durch *Connection* • mit *Event* durch *EventConnection* • mit *Start* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *EnterpriseActivity* durch *Connection* • mit *FunctionalOperation* durch *Connection* • mit *Finish* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	HorizontalReferences-Rules	keine (in beiden Sichten)
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 12. Die Node-Instanz "ProceduralRule"

Node-Instanz für *Start*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Start*
ViewModel	ViewOutputModel	In der *Function*-Sicht:  In der *FunctionTree*-Sicht: • nicht sichtbar
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Function*-Sicht: • mit *ProceduralRule* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *ProceduralRule* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	HorizontalReferences-Rules	keine (in beiden Sichten)
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 13. Die Node-Instanz "Start"

Node-Instanz für *Finish*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Finish*
ViewModel	ViewOutputModel	In der *Function*-Sicht:  In der *FunctionTree*-Sicht: • nicht sichtbar
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Function*-Sicht: • mit *ProceduralRule* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Function*-Sicht: • mit *ProceduralRule* durch *Connection* mögliche Verbindungen in der *FunctionTree*-Sicht: • keine
	HorizontalReferences-Rules	keine (in beiden Sichten)
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 14. Die Node-Instanz "Finish"

Arc-Instanz für *Connection*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Connection*
ViewModel	ViewOutputModel	In der *Function*-Sicht: In der *FunctionTree*-Sicht: • nicht sichtbar
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 15. Die Arc-Instanz "Connection"

7.1.3 Schritt 2: Entwicklung des Funktionsmodells

Nach der Definition der Modellierungskonstrukte der Funktionssicht kann die Entwicklung des Funktionsmodells erfolgen. Die Informationen aus den bereits definierten Baustein-Instanzen werden verwendet, um die entsprechenden dynamischen Baustein-Templates für die Baustein-Occurrences zu generieren (im Anhang B wird die Definition der dynamischen Baustein-Templates, die nach allen Erweiterungen der Methode generiert wurden, präsentiert). Die generierten dynamischen Baustein-Templates werden dann verwendet, um die Baustein-Occurrences während der Entwicklung des Modells zu erzeugen. Die tabellarische Darstellung der Informationen aus den ausgefüllten dynamischen Baustein-Templates, welche die Baustein-

Occurrences repräsentieren, ist für die Präsentation der Struktur des Modells nicht zweckmäßig. Das entwickelte Modell wird daher in graphischer Form dargestellt, so wie es auch im Werkzeug während bzw. nach der Entwicklung präsentiert wird.

Für die Präsentation der hierarchischen Struktur des Funktionsmodells wurde die *FunctionTree*-Sicht definiert. Diese Struktur (die in der *Function*-Sicht erstellt wurde) wird in der Abb. 71 dargestellt.

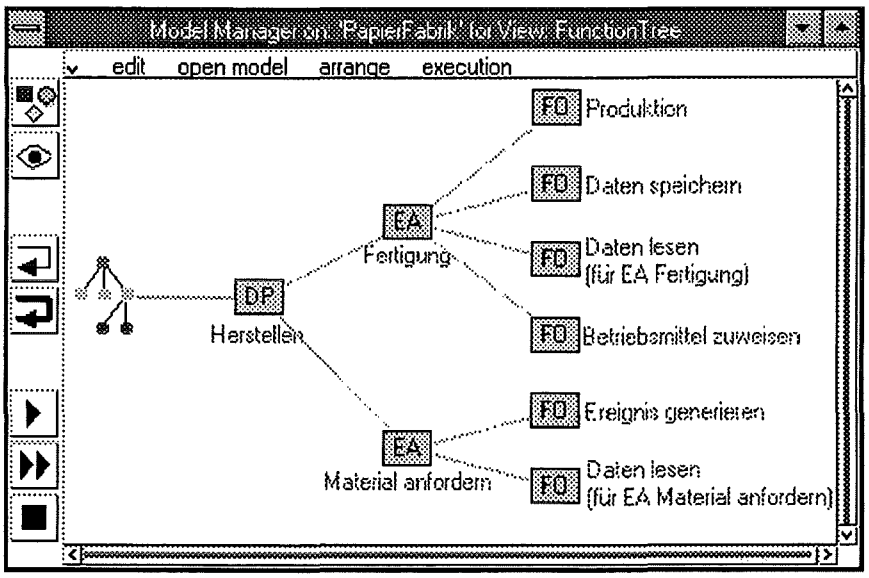


Abb. 71. Die hierarchische Struktur des Funktionsmodells

In dem Modell wird ein Domain Process *Herstellen* definiert. Dieser Prozeß wird durch zwei Enterprise Activities, *Material anfordern* und *Fertigung*, verfeinert. Die Enterprise Activity *Material anfordern* besteht aus zwei Functional Operations: *Daten lesen* und *Ereignis generieren*. Die Enterprise Activity *Fertigung* besteht aus vier Functional Operations: *Betriebsmittel zuweisen*, *Daten lesen*, *Produktion* und *Daten speichern*.

Der Domain Process *Herstellen* beschreibt die Abläufe bei der Grundpapierherstellung. Die Abläufe innerhalb dieses Prozesses werden durch zwei Events getriggert: *Trigger Fertigungsauftrag* und *Material bereit*. Bei der Ausführung dieses Prozesses werden dabei zwei weitere Events generiert: *Material wird benötigt* und *Material wegbringen*. Diese vier Events werden von anderen Domain Processes generiert bzw. in anderen Domain Processes verwendet. Die anderen Domain Processes werden hier nicht modelliert. Den Domain Process Herstellen zusammen mit den vier Events stellt die Abb. 72 (in der *Function*-Sicht) dar.

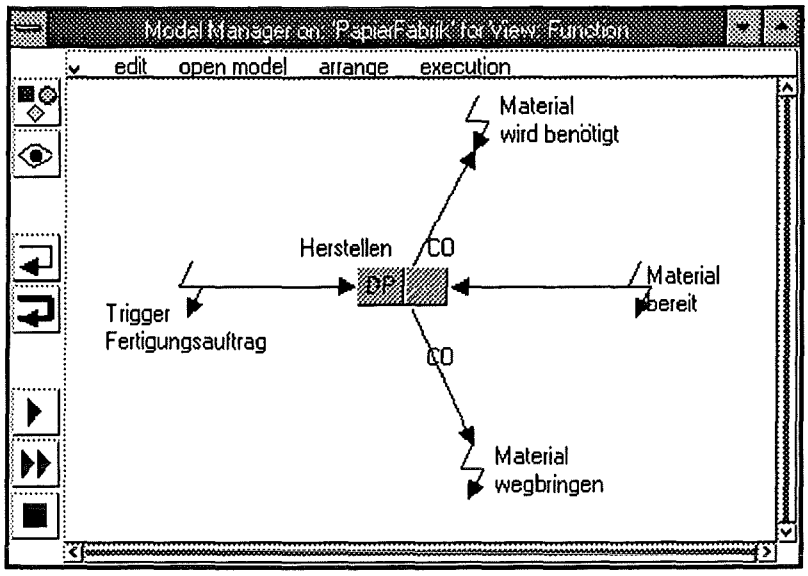


Abb. 72. Der Domain Process "Herstellen" und die Events

Die Verfeinerung des Domain Process *Herstellen* wird in der Abb. 73 dargestellt.

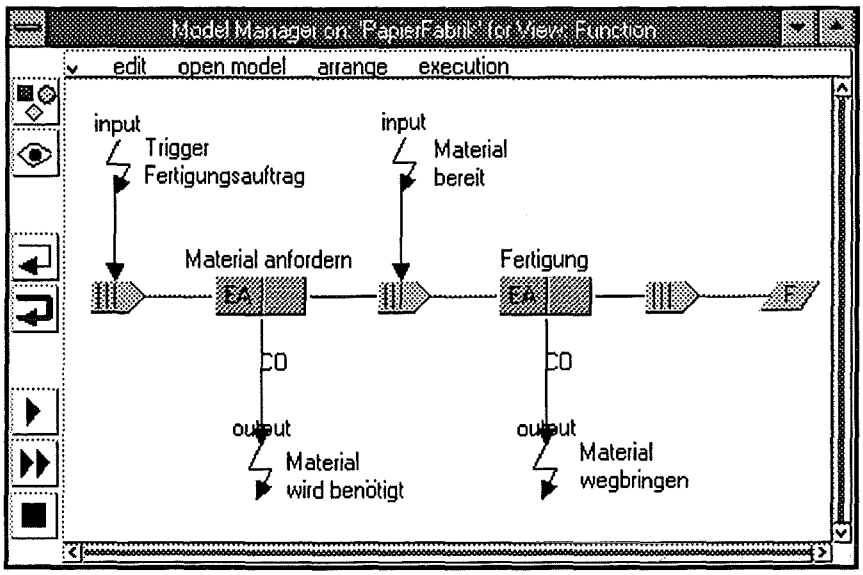


Abb. 73. Die Verfeinerung des Domain Process "Herstellen"

Die Enterprise Activities des Domain Process werden sequentiell ausgeführt. Die erste Enterprise Activity wird durch das Event *Trigger Fertigungsauftrag* gestartet. Die Ausführung der zweiten Enterprise Activity wird erst nach dem Ausführen der ersten Enterprise Activity möglich und wird durch das Event *Material bereit* initiiert. Beide Enterprise Activities generieren bei der Ausführung jeweils ein Event (*Material wird benötigt* bzw. *Material wegbringen*).

Die Funktionalität der Enterprise Activity *Material anfordern* wird durch zwei sequentielle Functional Operations beschrieben. Die zweite Functional Operation generiert bei der Ausführung das Event *Material wird benötigt* (Abb. 74).

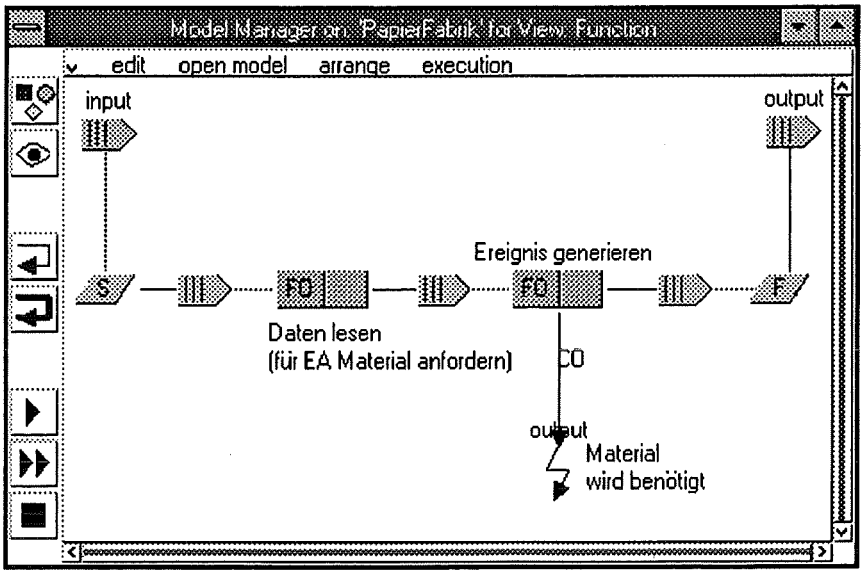


Abb. 74. Die Enterprise Activity "Material anfordern"

Die Funktionalität der Enterprise Activity *Fertigung* wird durch vier Functional Operations beschrieben. Die Functional Operations *Betriebsmittel zuweisen* und *Daten lesen* werden parallel ausgeführt. Die zwei übrigen Functional Operations (*Produktion* und *Daten speichern*) werden danach sequentiell ausgeführt, wobei die Functional Operation *Produktion* das Event *Material wegbringen* generiert (Abb. 75).

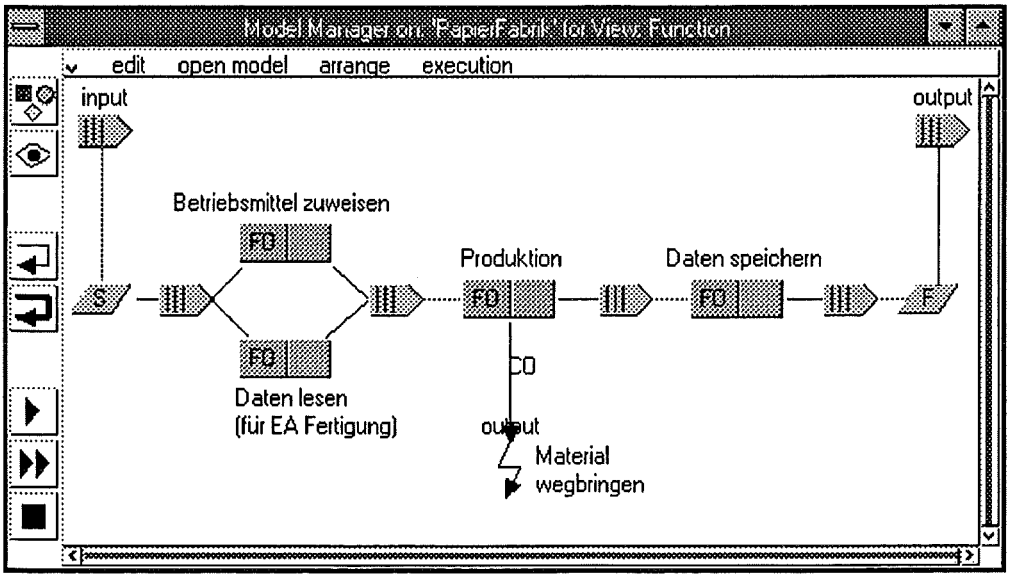


Abb. 75. Die Enterprise Activity "Fertigung"

7.1.4 Schritt 3: Definition der Informations-Sicht

Nach der Entwicklung des Funktionsmodells wird die CIMOSA-Modellierungsmethode erweitert, in dem die Informations-Sicht zusammen mit ihren Modellierungskonstrukten definiert wird. Um die neue Sicht einzufügen muß die Definition der Component-Instanz *CIMOSAModel* erweitert werden, wie es in der folgenden Tabelle dargestellt ist.

Component-Instanz für *CIMOSAModel*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*CIMOSAModel*
GeneralViewModel		**Erweiterung**: neue Sicht: • *Information* - eine horizontale Sicht zur Darstellung des Informationsmodells
ViewModel	ViewOutputModel	ohne Änderung
	ViewInputModel	ohne Änderung
RulesModel	InputConnectionRules	ohne Änderung
	OutputConnection-Rules	ohne Änderung
	HorizontalReferences-Rules	ohne Änderung
	VerticalReferences-Rules	**Erweiterung**: mögliche Verbindungen in der *Information*-Sicht: • mit *InformationElement* durch *SubReference* • mit *ObjectView* durch *SubReference* • mit *Name* durch *SubReference*
ImplementationModel	InformationModel	ohne Änderung
	FunctionModel	ohne Änderung

Tab. 16. Erweiterung der Definition der Component-Instanz "CIMOSAModel"

Anschließend werden zwei neue Node-Instanzen, *InformationElement* und *ObjectView* definiert, wie in den folgenden Tabellen dargestellt.

Node-Instanz für *InformationElement*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*InformationElement*
ViewModel	ViewOutputModel	In der *Function*-Sicht: • nicht sichtbar In der *FunctionTree*-Sicht: • nicht sichtbar In der *Information*-Sicht: 
	ViewInputModel	in der *Function*-Sicht: • keine zusätzlichen Eingabeelemente in der *FunctionTree*-Sicht: • keine zusätzlichen Eingabeelemente in der *Information*-Sicht: • ein zusätzlicher Menüeintrag: "edit name"
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Function*-Sicht: • keine mögliche Verbindungen in der *FunctionTree*-Sicht: • keine mögliche Verbindungen in der *Information*-Sicht: • mit *ObjectView* durch *Connection*
	OutputConnection-Rules	keine in allen Sichten
	HorizontalReferences-Rules	mögliche Verbindungen in der *Function*-Sicht: • keine mögliche Verbindungen in der *FunctionTree*-Sicht: • keine mögliche Verbindungen in der *Information*-Sicht: • mit *Name* durch *Reference*
	VerticalReferences-Rules	keine in allen Sichten
ImplementationModel	InformationModel	kein
	FunctionModel	Funktionen für: • Editieren des Namens (für den Menüeintrag "edit name")

Tab. 17. Die Node-Instanz "InformationElement"

Node-Instanz für ObjectView		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*ObjectView*
ViewModel	ViewOutputModel	In der *Function*-Sicht: • nicht sichtbar In der *FunctionTree*-Sicht: • nicht sichtbar In der *Information*-Sicht: 
	ViewInputModel	in der *Function*-Sicht: • keine zusätzlichen Eingabeelemente in der *FunctionTree*-Sicht: • keine zusätzlichen Eingabeelemente in der *Information*-Sicht: • ein zusätzlicher Menüeintrag: "edit name"
RulesModel	InputConnectionRules	keine in allen Sichten
	OutputConnection-Rules	mögliche Verbindungen in der *Function*-Sicht: • keine mögliche Verbindungen in der *FunctionTree*-Sicht: • keine mögliche Verbindungen in der *Information*-Sicht: • mit *InformationElement* durch *Connection*
	HorizontalReferences-Rules	mögliche Verbindungen in der *Function*-Sicht: • keine mögliche Verbindungen in der *FunctionTree*-Sicht: • keine mögliche Verbindungen in der *Information*-Sicht: • mit *Name* durch *Reference*
	VerticalReferences-Rules	keine in allen Sichten
ImplementationModel	InformationModel	kein
	FunctionModel	Funktionen für: • Editieren des Namens (für den Menüeintrag "edit name")

Tab. 18. Die Node-Instanz "ObjectView"

7.1.5 Schritt 4: Entwicklung des Informationsmodells

Nach der Definition der *Information*-Sicht wird das bis jetzt entwickelte CIMOSA-Modell (das nur ein Funktionsmodell beinhaltet) mit dem Informationsmodell erweitert. Es werden drei ObjectViews definiert: *Auftrag*, *Produkt* und *Papiermaschine*[24]. Die einzelnen *ObjectViews* werden mit Hilfe der entsprechenden *InformationElements* näher beschrieben (Abb. 76). Die *InformationElements* können dabei als Attribute der *ObjectViews* gesehen werden.

[24] Für die Beschreibung der Betriebsmittel ist bei der CIMOSA-Modellierung die Betriebsmittel-Sicht vorgesehen. Zur Vereinfachung wir hier für die Beschreibung der Papiermaschine das Konstrukt *ObjectView* aus der Information-Sicht verwendet.

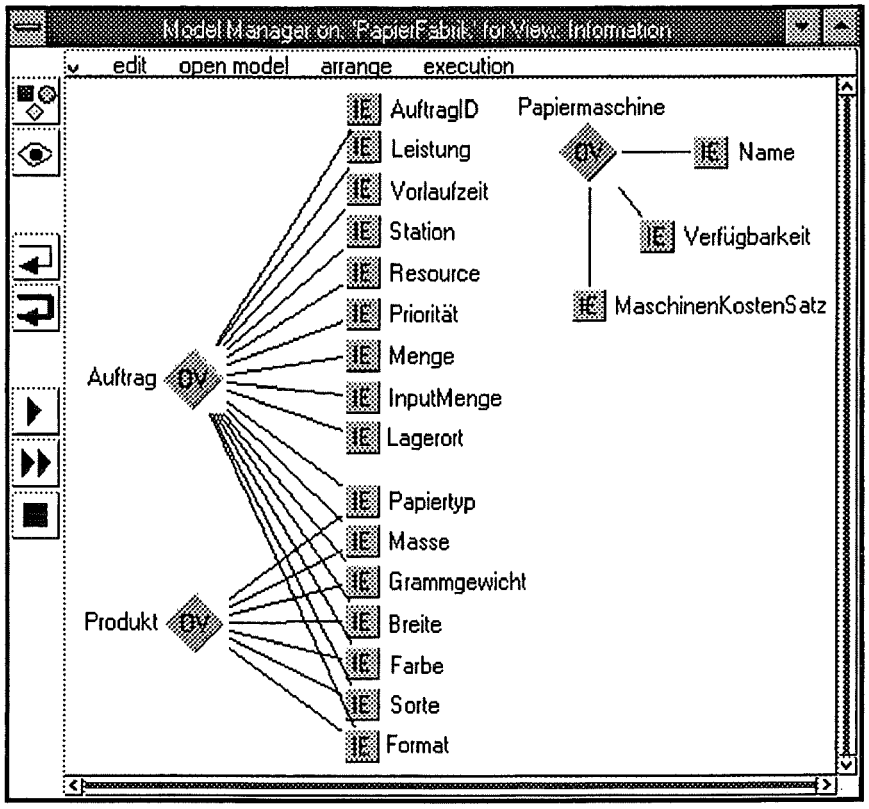


Abb. 76. Die Definition des Informationsmodells

7.1.6 Schritt 5: Definition FunktionsInformations-Sicht

Nach der Entwicklung des Informationsmodells wird die CIMOSA-Modellierungsmethode wieder erweitert, in dem eine zusätzliche Sicht definiert wird, die als *FunctionInformation*-Sicht bezeichnet wird. Diese Sicht dient nur zur Definitionen der Querverbindungen zwischen den Konstrukten der Funktions- und der Informationssicht. Sie hat keine Entsprechung in der CIMOSA-Modellierungsmethode. Um die neue Sicht einzufügen muß die Definition der Component-Instanz *CIMOSAModel* wieder erweitert werden, wie es in der folgenden Tabelle gezeigt wird.

Component-Instanz für *CIMOSAModel*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*CIMOSAModel*
GeneralViewModel		**Erweiterung**: neue Sicht: • *FunctionInformation* - eine vertikale Sicht in der alle Enterprise Functions und ObjectViews, jedoch ohne Strukturierung, präsentiert werden
ViewModel	ViewOutputModel	ohne Änderung
	ViewInputModel	ohne Änderung
RulesModel	InputConnectionRules	ohne Änderung
	OutputConnection-Rules	ohne Änderung
	HorizontalReferences-Rules	ohne Änderung
	VerticalReferences-Rules	ohne Änderung
ImplementationModel	InformationModel	ohne Änderung
	FunctionModel	ohne Änderung

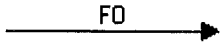
Tab. 19. Weitere Erweiterung der Definition der Component-Instanz "CIMOSAModel"

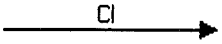
Für die Darstellung der Querverbindungen müssen neue Arc-Instanzen definiert werden. Es sind: *FIConnection* (Function Input), *FOConnection* (Function Output), *CIConnection* (Control Input), *RIConnection* (Resource Input) und *ROConnection* (Resource Output). Die Arc-Instanz *COConnection* wurde bereits bei der *Function*-Sicht definiert. Die Definition der Arc-Instanz *FIConnection* wird in der folgenden Tabelle präsentiert.

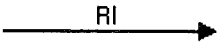
Arc-Instanz für *FIConnection*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*FIConnection*
ViewModel	ViewOutputModel	In der *Function*-Sicht: • nicht sichtbar In der *FunctionTree*-Sicht: • nicht sichtbar In der *Information*-Sicht: • nicht sichtbar In der *FunctionInformation*-Sicht: FI ⟶
	ViewInputModel	kein in allen Sichten
ImplementationModel	InformationModel	kein
	FunctionModel	kein

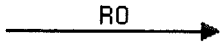
Tab. 20. Die Arc-Instanz "FIConnection"

Die Definitionen der übrigen Arc-Instanzen unterscheiden sich nur durch das Attribut *InstanceName* und die Definition des *ViewOutputModels* für die *FunctionInformation*-Sicht. Die unterschiedliche *InstanceNames* und *ViewOutputModels* sind:

- *FOConnection* - 

- *CIConnection* - 

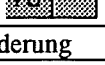
- *RIConnection* - 

- *ROConnection* - 

Um die Erstellung der Querverbindungen zu ermöglichen, müssen die Definitionen der Node-Instanzen *EnterpriseActivity*, *FunctionalOperation*, *Event* und *ObjectView* erweitert werden.

Node-Instanz für *EnterpriseActivity*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*EnterpriseActivity*
ViewModel	ViewOutputModel	**Erweiterung**: in der *FunctionInformation*-Sicht: 
	ViewInputModel	ohne Änderung
RulesModel	InputConnectionRules	**Erweiterung**: mögliche Verbindungen in der *FunctionInformation*-Sicht: • mit *ObjectView* durch *FIConnection*, *CIConnection* oder *RIConnection*
	OutputConnection-Rules	**Erweiterung**: mögliche Verbindungen in der *FunctionInformation*-Sicht: • mit *ObjectView* durch *FOConnection*, oder *ROConnection*
	HorizontalReferences-Rules	ohne Änderung
	VerticalReferences-Rules	ohne Änderung
ImplementationModel	InformationModel	ohne Änderung
	FunctionModel	ohne Änderung

Tab. 21. Die Erweiterte Definition der Node-Instanz "EnterpriseActivity"

Node-Instanz für *FunctionalOperation*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*FunctionalOperation*
ViewModel	ViewOutputModel	**Erweiterung**: in der *FunctionInformation*-Sicht: 
	ViewInputModel	ohne Änderung
RulesModel	InputConnectionRules	**Erweiterung**: mögliche Verbindungen in der *FunctionInformation*-Sicht: • mit *ObjectView* durch *FIConnection*, *CIConnection* oder *RIConnection*
	OutputConnection-Rules	**Erweiterung**: mögliche Verbindungen in der *FunctionInformation*-Sicht: • mit *ObjectView* durch *FOConnection*, oder *ROConnection* • mit *Event* durch *COConnection*
	HorizontalReferences-Rules	ohne Änderung
	VerticalReferences-Rules	ohne Änderung
ImplementationModel	InformationModel	ohne Änderung
	FunctionModel	ohne Änderung

Tab. 22. Die erweiterte Definition der Node-Instanz "FunctionalOperation"

Node-Instanz für *Event*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Event*
ViewModel	ViewOutputModel	**Erweiterung**: in der *FunctionInformation*-Sicht: 
	ViewInputModel	ohne Änderung
RulesModel	InputConnectionRules	ohne Änderung
	OutputConnection-Rules	ohne Änderung
	HorizontalReferences-Rules	ohne Änderung
	VerticalReferences-Rules	ohne Änderung
ImplementationModel	InformationModel	ohne Änderung
	FunctionModel	ohne Änderung

Tab. 23. Die erweiterte Definition der Node-Instanz "Event"

Node-Instanz für *ObjectView*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*ObjectView*
ViewModel	ViewOutputModel	**Erweiterung**: in der *FunctionInformation*-Sicht: 
	ViewInputModel	ohne Änderung
RulesModel	InputConnectionRules	**Erweiterung**: mögliche Verbindungen in der *FunctionInformation*-Sicht: • mit *EnterpriseActivity* durch *FOConnection* oder *ROConnection* • mit *FunctionalOperation* durch *FOConnection* oder *ROConnection*
	OutputConnection-Rules	**Erweiterung**: mögliche Verbindungen in der *FunctionInformation*-Sicht: • mit *EnterpriseActivity* durch *FIConnection*, *CIConnection* oder *RIConnection* • mit *FunctionalOperation* durch *FIConnection*, *CIConnection* oder *RIConnection*
	HorizontalReferences-Rules	ohne Änderung
	VerticalReferences-Rules	ohne Änderung
ImplementationModel	InformationModel	ohne Änderung
	FunctionModel	ohne Änderung

Tab. 24. Die erweiterte Definition der Node-Instanz "ObjectView"

7.1.7 Schritt 6: Definition der Querverbindungen zwischen dem Funktions- und Informationsmodell

Nach der Definition der *FunctionInformation*-Sicht wird das bis jetzt entwickelte CIMOSA-Modell (das bereits ein Funktions- und Informationsmodell beinhaltet) durch die Definition der

entsprechenden Querverbindungen im letzen Schritt vervollständigt. Die Definition der Querverbindungen in der *FunctionInformation*-Sicht stellt die Abb. 77 dar.

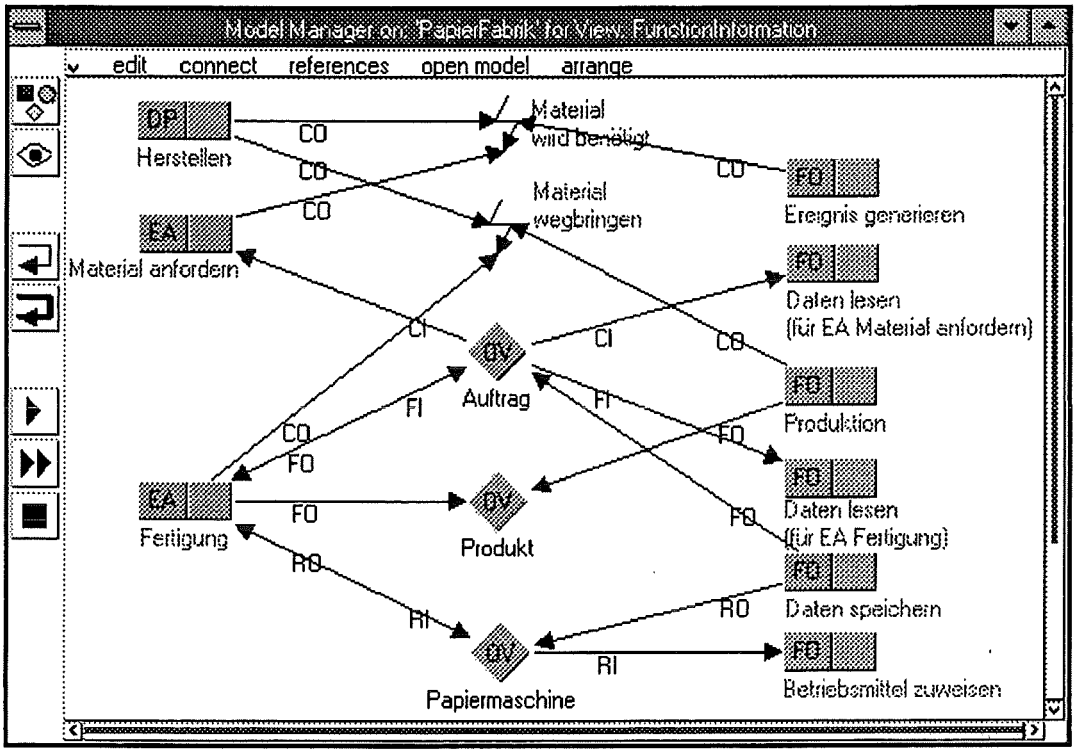


Abb. 77. Die Definition der Querverbindungen

7.2 Softwareentwicklung mit Petri-Netzen

7.2.1 Definition des Szenarios

7.2.1.1 Methode

Es wird eine Variante der Petri-Netz-Methode definiert, die ermöglicht, hierarchisch strukturierte Pr/T-Netze (Prädikat/Transitionsnetzen) zu erstellen und zu simulieren. Es werden folgende Petri-Netz-Konstrukte entwickelt:

- **Transition** - zur Darstellung der Transitionen. Jede Transition kann dabei über einen Action-Code verfügen. Der Action-Code wird von dem System-Entwickler in Smalltalk-80 definiert. Der Action-Code wird beim Schalten einer Transition ausgeführt, und kann dabei z.B. die Attribute der Token ändern.
- **Place** - zur Darstellung der Stellen.
- **Token** - zur Darstellung der Token. Jeder Token kann über verschiedene Attribute verfügen, die im Action-Code einer Transition verwendet bzw. geändert werden können.
- **Arc** - zur Darstellung der Kanten, welche die Transitionen und Stellen miteinander verbinden. Jede Kante kann über verschiedene Attribute verfügen, die bestimmen, welche Token durch diese Kante beim Feuern einer Transition fließen können.
- **Module** - zur Hierarchisierung der Petri-Netz-Modelle. Jedes Modul kann durch die Verwendung von Transitionen, Stellen, Kanten und weiteren Modulen verfeinert werden.
- **Pool** - zur Darstellung einer Stelle mit speziellen Eigenschaften. In einem Petri-Netz-Modell können mehrere Pools verwendet werden. Sie repräsentieren jedoch immer die selbe Stelle. Wenn ein Token in einem Pool abgelegt wird, wird er in allen anderen Pools ebenfalls verfügbar. Wenn ein Token von einem Pool entfernt wird, wird er auch von allen anderen Pools entfernt. Mit Hilfe der Pools können damit Cross-Verbindungen zwischen Transitionen aus verschiedenen Teilen und verschiedenen Hierarchieebenen des Petri-Netz-Modells definiert werden. Es müssen dabei keine Verbindungen zwischen allen übergeordneten Modulen definiert werden, wie es bei der Verwendung von "normalen" Stellen notwendig wäre.

Es werden zwei Sichten definiert: *Standard*-Sicht zur Darstellung der Pr/T-Netze auf einer Hierarchieebene und *Tree*-Sicht zur Darstellung der hierarchischen Struktur der Pr/T-Netze. In der Standard-Sicht werden alle oben aufgelistete Konstrukte aus einer Hierachieebene angezeigt. In der *Tree*-Sicht werden nur die Module des entwickelten Petri-Netzes angezeigt. In der *Standard*-Sicht wird das Erstellen von verschiedenen Ikonen ermöglicht. Die Ikonen können in dem entwickelten Petri-Netz-Modell verwendet werden, um die Darstellung der einzelnen Module, Transitionen und Stellen individuell zu gestalten.

Für die Simulation wird ein einfacher Algorithmus implementiert. Die Implementation des Algorithmus wird dabei auf die Funktionsmodelle der verschiedenen Baustein-Instanzen verteilt. In einem Simulationsschritt wird eine Liste der Transitionen erstellt, bei denen die

Feuerbedingungen erfüllt sind. Aus dieser Liste wird zufällig eine Transition ausgewählt und gefeuert. Die Simulation kann in einzelnen Schritten erfolgen, die jedesmal vom Benutzer initiiert werden, oder sie kann kontinuierlich durch Wiederholung dieser Simulationsschritte ablaufen. Die Zusammenstellung der Liste der feuerbaren Transitionen in jedem Simulationsschritt hat eine geringe Effizienz des Simulationsalgorithmus als Folge. Die Implementation eines effizientes Simulationsalgorithmus ist jedoch nicht das Ziel dieser Arbeit. Mit diesem Beispiel soll lediglich gezeigt werden, daß die Definition einer Methode, welche die Simulation unterstützt, möglich ist.

Bei der Definition der Methode werden folgende Regeln berücksichtigt (hierbei wird ein Pool auch als Stelle bezeichnet):

⇒ Auf jeder Hierarchieebene (repräsentiert durch ein Modul) können Transitionen, Stellen und Module erzeugt werden.

⇒ Durch Verwendung der Kanten können Stellen mit Transitionen und Modulen verbunden werden.

⇒ In jeder Stelle kann eine beliebige Zahl von Token abgelegt werden.

⇒ Jedes Modul kann benannt werden.

⇒ Für jede Transition kann ein Action-Code definiert werden.

⇒ Für jeden Token kann eine Liste von Attributen definiert werden. Jedes Attribut verfügt über einen Namen und über einen Wert. Die Attribute eines Tokens sind Platzhalter für Informationen, die in diesem Token gespeichert werden.

⇒ Für jede Kante, kann eine Liste von Attributen definiert werden. Jedes Attribut verfügt über einen Namen und optional über einen Wert. Die Liste der Attribute einer Kante bestimmt, welche Token durch diese Kante beim Feuern einer Transition, die mit einer Stelle durch diese Kante verbunden ist, fließen können. Ein Token kann durch die Kante fließen, wenn:

- ◊ die Zahl der Attribute und die Namen der Attribute des Tokens und der Kante gleich sind, und
- ◊ die definierten Werte der Kanten-Attribute mit den Werten den gleichnamigen Token-Attribute übereinstimmen. Wenn für ein Kanten-Attribut kein Wert definiert wurde, können die Werte der gleichnamigen Token-Attribute beliebig sein.

7.2.1.2 System-Modell

Mit der oben definierten Methode wird ein System-Modell entwickelt, das die vereinfachte Funktionalität des Business-Services der CIMOSA-Integrationsplattform implementiert. Die Aufgabe des Business-Services ist die Ausführung der CIMOSA-Modelle.

Die CIMOSA-Modelle werden mit Hilfe von Petri-Netz-Konstrukten dargestellt. Enterprise Functions und Procedural Rules werden dabei als Module bzw. Transitionen definiert. Die übrigen CIMOSA-Modellierungskonstrukte werden als Stellen definiert (für die Erklärung der CIMOSA-

Modellierungskonstrukte siehe Kapitel 7.1.1.1). Stellen werden zusätzlich verwendet, um die Verbindungen zwischen den Enterprise Functions und Procedural Rules zu ermöglichen. Für eine übersichtlichere Darstellung der CIMOSA-Modelle werden verschiedene Ikonen definiert und eingesetzt, so daß die Modelle in ähnlicher Weise, wie in Kapitel 7.1.3, dargestellt werden.

Die wichtigste Aufgabe bei der Entwicklung des System-Modells ist die Realisierung der Anbindung der Petri-Netz-Struktur, die das CIMOSA-Modell repräsentiert, an die Petri-Netz-Struktur, welche die Funktionalität des Business-Services implementiert. Die Realisierung dieser Anbindung ist problematisch, weil beide Strukturen von unterschiedlicher Natur sind. Die Struktur des Business-Services ist vorgegeben und ändert sich nicht. Die Struktur der CIMOSA-Modelle ist dagegen nicht vordefiniert und kann je nach definiertem Modell variieren.

In [Bog93] wurde eine Vorgehensweise zur Lösung dieses Problems beschrieben. Bei der Ausführung einer Functional Operation (d.h. wenn ein Token durch die Petri-Netz-Struktur fließt, die diese Functional Operation repräsentiert) wird ein Ereignis generiert, das an den Business Service weiter geleitet wird. Der Business Service führt die entsprechenden Aktivitäten durch und generiert ein neues Ereignis, das an die Functional Operation weitergeleitet wird. Danach verläßt der Token die Functional Operation - die Ausführung der Functional Operation ist beendet (Abb. 78). Die Ereignisse können parallel von verschiedenen Functional Operations generiert und empfangen werden. Damit können CIMOSA-Modelle mit parallelen Abläufen ausgeführt werden.

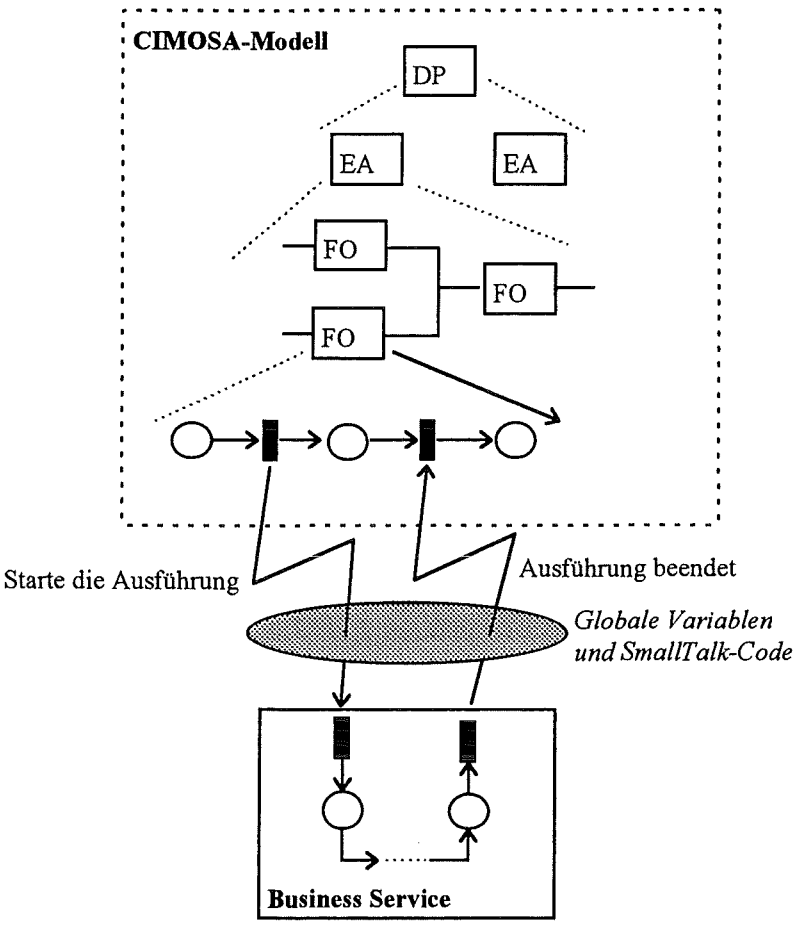


Abb. 78. Die Ereignisorientierte Ausführung der CIMOSA-Modelle

Für die Implementation des Mechanismus zum Austausch der Ereignisse standen im beschriebenen Einsatz eingesetzten Petri-Netz-Tool[25] keine entsprechende Petri-Netz-Konstrukte zur Verfügung. Dieser Mechanismus wurde daher mit Hilfe von sogenannten globalen Variablen realisiert, die zum Austausch der Ereignisse dienten. Es mußte dabei zusätzlicher Smalltalk-Code verwendet werden, um die Konflikte beim Zugriff auf diese globalen Variablen zu vermeiden.

Die Realisierung des Ereignisaustausches kann durch den Einsatz des in dem Kapitel 7.2.1.1 beschriebenen Konstruktes *Pool* wesentlich vereinfacht werden (Abb. 79).

[25] Es wurde das kommerzielle Petri-Netz-Tool PACE eingesetzt [PACE93]. Das Tool wurde in Smalltlk-80 implementiert. Die Funktionalität der Petri-Netze kann in diesem Tool durch die Definition vom zusätzlichem SmallTalk-Code erweitert werden (z.B. in einem Action-Code einer Transition)

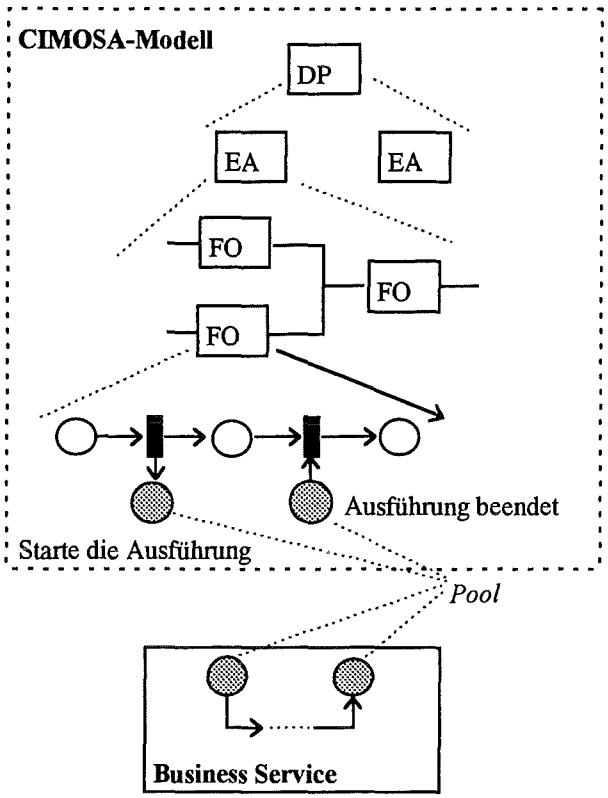


Abb. 79. Der Einsatz des Konstruktes Pool

Die Ereignisse werden dazu als Token implementiert. Der Austausch der Ereignisse erfolgt durch Ablegen und Entfernen dieser Token in bzw. aus dem Pool. Während der Simulation wird der Pool als eine gewöhnliche Stelle betrachtet. Dadurch muß kein zusätzlicher Mechanismus zur Vermeidung von Zugriffskonflikten implementiert werden, da diese Konflikte bereits im Simulationsalgorithums der Petri-Netze ausgeschlossen werden.

7.2.1.3 Entwicklung

Die Entwicklung der Petri-Netz-Methode und des Petri-Netz-Modells wird in vier Schritten (zwei Schritte der Methodendefinition und zwei Schritte der Systementwicklung) realisiert. Die einzelnen Schritte werden im weiteren näher beschrieben. Diese Schritte bilden ein Entwicklungspfad, in dem die Methode nur einmal erweitert wird, um die angestrebte Funktionalität des Systems zu erreichen. Diese Erweiterung der herkömlichen Methode erlaubt es durch die Einführung des Konstruktes *Pool*, die Funktionalität des Systems zu implementieren, die mit der Ausführung der CIMOSA-Modelle durch den Business Service verbunden ist. Es werden insgesamt zwei aufeinander folgende Prototypen entwickelt (Abb. 80).

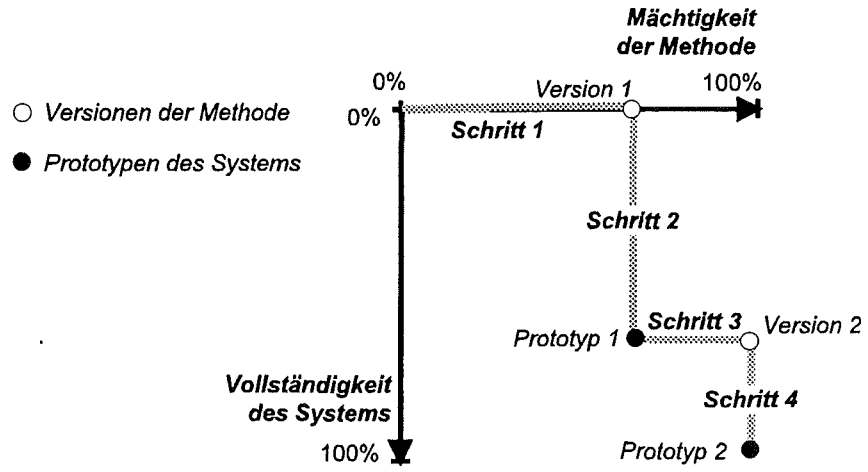


Abb. 80. Entwicklungspfad des zweiten Szenarios

7.2.2 Schritt 1: Definition der Petri-Netz-Methode

In dem ersten Schritt werden die bereits beschriebenen Petri-Netz-Konstrukte, jedoch ohne das Pool-Konstrukt, definiert. Es werden dabei die ebenfalls bereits beschrieben Sichten definiert. In den Funktions-Modellen der im weiteren dargestellten Baustein-Instanzen werden zahlreiche Funktionen definiert, die den Simulationsalgorithmus realisieren. Die ausführliche Auflistung und Beschreibung dieser Funktionen würde die Rahmen dieser Arbeit sprengen. Es werden daher nur kurze Beschreibungen der Aufgaben dieser Funktionen präsentiert.

Als erstes muß, ähnlich wie bei der Definition der CIMOSA-Modellierungsmethode, eine Component-Instanz definiert werden, um bei der Entwicklung eines Modells das Erzeugen einer Component-Occurrence zu ermöglichen, die als Wurzel des gesamten Modells dienen soll.

Component-Instanz für *PetriNetComponent*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*PetriNetComponent*
GeneralViewModel		Es werden zwei Sichten vorgesehen: • *Standard* - eine horizontale Sicht zur Darstellung einer Hierarchieebene • *Tree* - eine vertikale Sicht zur Darstellung der gesamten Hierarchie des Petri-Netz-Modells
ViewModel[26]	ViewOutputModel	In der *Standard*-Sicht: • nicht sichtbar In der *Tree*-Sicht: component
	ViewInputModel	in der *Standard*-Sicht: • ein zusätzlicher Menüeintrag: "edit icons" in der *Tree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	keine (in beiden Sichten)
	OutputConnection-Rules	keine (in beiden Sichten)
	HorizontalReferences-Rules	keine (in beiden Sichten)
	VerticalReferences-Rules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Transition* durch *SubReference* • mit *Place* durch *SubReference* • mit *Modul* durch *SubReference* • mit *Token* durch *SubReference* • mit *Name* durch *SubReference* • mit *ActionCode* durch *SubReference* mögliche Verbindungen in der *Tree*-Sicht: • keine
ImplementationModel	InformationModel	Information Element *IconList* zum Speichern der für das Petri-Netz-Modell definierten Ikonen
	FunctionModel	Funktionen für: • Erzeugen und Editieren der Ikonen (für den Menüeintrag "edit icons") • Ausführung eines Simulationsschrittes • kontinuierliche Ausführung durch Wiederholung der Simulationsschritte • Anhalten der laufenden Simulation

Tab. 25. Die Component-Instanz "PetriNetComponent"

Die Definition der im *RulesModel* genannten Baustein-Instanzen wird in den folgenden Tabellen dargestellt. *Transition*, *Place*, *Module*, und *Token* werden als Node-Instanzen definiert. *SubReference* wird als eine Arc-Instanz definiert. Zur Verbindung der Transitionen bzw. Module mit Stellen wird eine Arc-Instanz *Arc* definiert. Um die Module benennen zu können wird eine

[26] Bei der Component-Instanz und bei allen Node-Instanzen wird angenommen, daß die entsprechenden Baustein-Occurrences ihre Positionen speichern sollen, wenn sie in der betroffenen Sicht sichtbar sind.

zusätzliche Baustein-Instanz *Name* definiert. *Name* wird als eine Node-Instanz definiert. Um die Spezifikation des Action-Codes einer Transition zu ermöglichen, wird die Baustein-Instanz *ActionCode* definiert. Zur Verbindung der Baustein-Occurrences *Name* und *ActionCode* mit den anderen Baustein-Occurrences (Module bzw. Transitionen) wird eine Arc-Instanz *Reference* definiert.

Node-Instanz für *Transition*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Transition*
ViewModel	ViewOutputModel	In der *Standard*-Sicht:  In der *Tree*-Sicht: • nicht sichtbar
	ViewInputModel	in der *Standard*-Sicht: • ein zusätzlicher Menüeintrag: "edit action code" • ein zusätzlicher Menüeintrag: "change icon" in der *Tree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Place* durch *Arc* mögliche Verbindungen in der *Tree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Place* durch *Arc* mögliche Verbindungen in der *Tree*-Sicht: • keine
	HorizontalReferences-Rules	mögliche Verbindungen in der *Standard*-Sicht: • mit *ActionCode* durch *Reference* • mit *Token* durch *Reference* mögliche Verbindungen in der *Tree*-Sicht: • keine
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	Information Element *IconName* zum Speichern des Namens einer Ikone
	FunctionModel	Funktionen für: • Prüfen, ob die Transition gefeuert werden kann • Feuern der Transition • Editieren des Action-Codes (für den Menüeintrag "edit action code") • Zuweisung der Ikone (für den Menüeintrag "change icon")

Tab. 26. Die Node-Instanz "Transition"

Node-Instanz für *Place*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Place*
ViewModel	ViewOutputModel	In der *Standard*-Sicht: ⁘ In der *Tree*-Sicht: • nicht sichtbar
	ViewInputModel	in der *Standard*-Sicht: • ein zusätzlicher Menüeintrag: "change icon" in der *Tree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Transition* durch *Arc* • mit *Modul* durch *Arc* mögliche Verbindungen in der *Tree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Transition* durch *Arc* • mit *Modul* durch *Arc* mögliche Verbindungen in der *Tree*-Sicht: • keine
	HorizontalReferences-Rules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Token* durch *Reference* mögliche Verbindungen in der *Tree*-Sicht: • keine
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	Information Element *IconName* zum Speichern des Namens einer Ikone
	FunctionModel	Funktionen für: • Bereitstellung einer Liste der Token, die sich in der Stelle befinden • Zuweisung der Ikone (für den Menüeintrag "change icon")

Tab. 27. Die Node-Instanz "Place"

Node-Instanz für *Modul*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Modul*
ViewModel	ViewOutputModel	In der *Standard*-Sicht:  In der *Tree*-Sicht: 
	ViewInputModel	in der *Standard*-Sicht: • ein zusätzlicher Menüeintrag: "edit name" • ein zusätzlicher Menüeintrag: "change icon" in der *Tree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Place* durch *Arc* mögliche Verbindungen in der *Tree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Place* durch *Arc* mögliche Verbindungen in der *Tree*-Sicht: • keine
	HorizontalReferences-Rules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Name* durch *Reference* mögliche Verbindungen in der *Tree*-Sicht: • keine
	VerticalReferences-Rules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Transition* durch *SubReference* • mit *Place* durch *SubReference* • mit *Modul* durch *SubReference* • mit *Token* durch *SubReference* • mit *Name* durch *SubReference* • mit *ActionCode* durch *SubReference* mögliche Verbindungen in der *Tree*-Sicht: • keine
ImplementationModel	InformationModel	Information Element *IconName* zum Speichern des Namens einer Ikone
	FunctionModel	Funktionen für: • Editieren des Namens (für den Menüeintrag "edit name") • Zuweisung der Ikone (für den Menüeintrag "change icon")

Tab. 28. Die Node-Instanz "Modul"

Node-Instanz für *Token*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Name*
ViewModel	ViewOutputModel	In der *Standard*-Sicht: • Text abhängig von den definierten Attributen in der *Tree*-Sicht: • nicht sichtbar
	ViewInputModel	in der *Standard*-Sicht: • ein zusätzlicher Menüeintrag: "edit attributes" in der *Tree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	keine (in beiden Sichten)
	OutputConnection-Rules	keine (in beiden Sichten)
	HorizontalReferences-Rules	keine (in beiden Sichten)
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	Information Element *AttributeList* zum Speichern der Liste der Attribute des Tokens
	FunctionModel	Funktionen für: • Editieren der Attributes (für den Menüeintrag "edit attributes") • Erstellen des Textes für die Darstellung der Attribute

Tab. 29. Die Node-Instanz "Token"

Node-Instanz für *Name*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Name*
ViewModel	ViewOutputModel	In der *Standard*-Sicht: • Text abhängig vom Information Element *NameStr* In der *Tree*-Sicht: • Text abhängig vom Information Element *NameStr*
	ViewInputModel	in der *Standard*-Sicht: • ein zusätzlicher Menüeintrag: "edit name" in der *Tree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	keine (in beiden Sichten)
	OutputConnection-Rules	keine (in beiden Sichten)
	HorizontalReferences-Rules	keine (in beiden Sichten)
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	Information Element *NameStr* vom Typ STRING
	FunctionModel	Funktionen für: • Editieren des Namens (für den Menüeintrag "edit name")

Tab. 30. Die Node-Instanz "Name"

Node-Instanz für *ActionCode*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*ActionCode*
ViewModel	ViewOutputModel	In der *Standard*-Sicht: • Text abhängig vom Information Element *CodeStr* In der *Tree*-Sicht: • kein
	ViewInputModel	in der *Standard*-Sicht: • ein zusätzlicher Menüeintrag: "edit action code" in der *Tree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	keine (in beiden Sichten)
	OutputConnection-Rules	keine (in beiden Sichten)
	HorizontalReferences-Rules	keine (in beiden Sichten)
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	Information Element *CodeStr* vom Typ STRING
	FunctionModel	Funktionen für: • Editieren des Codes (für den Menüeintrag "edit action code")

Tab. 31. Die Node-Instanz "ActionCode"

Arc-Instanz für *Arc*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Arc*
ViewModel	ViewOutputModel	In der *Standard*-Sicht: {} • der angezeigte Text hängt von den definierten Attributen ab In der *FunctionTree*-Sicht: • nicht sichtbar
	ViewInputModel	in der *Standard*-Sicht: • ein zusätzlicher Menüeintrag: "edit attributes" in der *Tree*-Sicht: • keine zusätzlichen Eingabeelemente
ImplementationModel	InformationModel	Information Element *AttributeList* zum Speichern der Liste der Attribute der Kante
	FunctionModel	Funktionen für: • Prüfen, ob ein Token durch die Kante fließen kann (Übereinstimmung der Attribute der Kante und des Tokens) • Editieren der Attribute (für den Menüeintrag "edit attributes") • Erstellen des Textes für die Darstellung der Attribute

Tab. 32. Die Arc-Instanz "Arc"

Arc-Instanz für *SubReference*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*SubReference*
ViewModel	ViewOutputModel	In der *Standard*-Sicht: • nicht sichtbar In der *Tree*-Sicht:
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 33. Die Arc-Instanz "SubReference"

Arc-Instanz für *Reference*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Reference*
ViewModel	ViewOutputModel	nicht sichtbar (in beiden Sichten)
	ViewInputModel	keine zusätzlichen Eingabeelemente (in beiden Sichten)
ImplementationModel	InformationModel	kein
	FunctionModel	kein

Tab. 34. Die Arc-Instanz "Reference"

7.2.3 Schritt 2: Entwicklung der Petri-Netz-Modelle

Nach der Definition der ersten Version der Petri-Netz-Methode kann die Entwicklung des Petri-Netz-Modells erfolgen (im Anhang C wird die Definition der dynamischen Baustein-Templates, die nach allen Erweiterungen der Methode generiert wurden, präsentiert). Ähnlich, wie bei den CIMOSA-Modellen (Kapiteln 7.1.3, 7.1.5 und 7.1.7), werden die entwickelten Petri-Netz-Modelle in graphischer Form dargestellt, so wie es auch im Werkzeug während bzw. nach der Entwicklung präsentiert wird.

Als erstes werden zwei unabhängige Module definiert, *CIMOSA Model* und *CIMOSA Services*. Im Modul *CIMOSA Model* werden die Petri-Netze definiert, welche die Struktur eines CIMOSA-Modells darstellen. Im Modul *CIMOSA Services* werden die Petri-Netze erstellt, welche die Funktionalität des Business Service und des zusätzlichen, für die Ausführung der CIMOSA-Modelle notwändigen Presentation Service[27] implementieren.

[27] Presentation Service ist für den Dialog mit Betriebsmitteln während der Ausführung der CIMOSA-Modelle zuständig

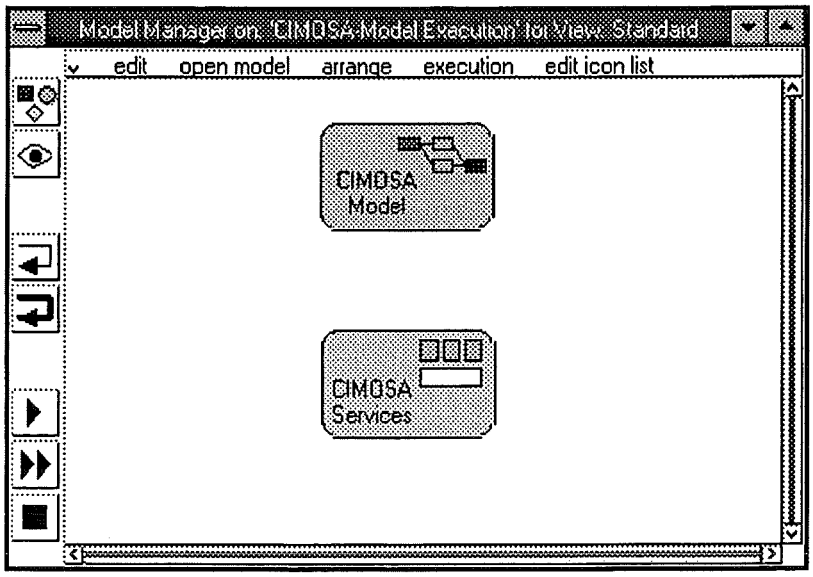


Abb. 81. Die Module "CIMOSA Model" und "CIMOSA Services"

Als Beispiel für das CIMOSA-Modell wird ein einfaches, abstraktes Funktionsmodell definiert. Das CIMOSA-Modell (also das Modul *CIMOSA Model*) besteht aus einem Domain Process *DP*. Der Domain Process *DP* beinhaltet eine Enterprise Activity *EA*. Die Enterprise Activity *EA* besteht aus drei Functional Operations *FO1*, *FO2* und *FO3*, die parallel ausgeführt werden (Abb. 82).

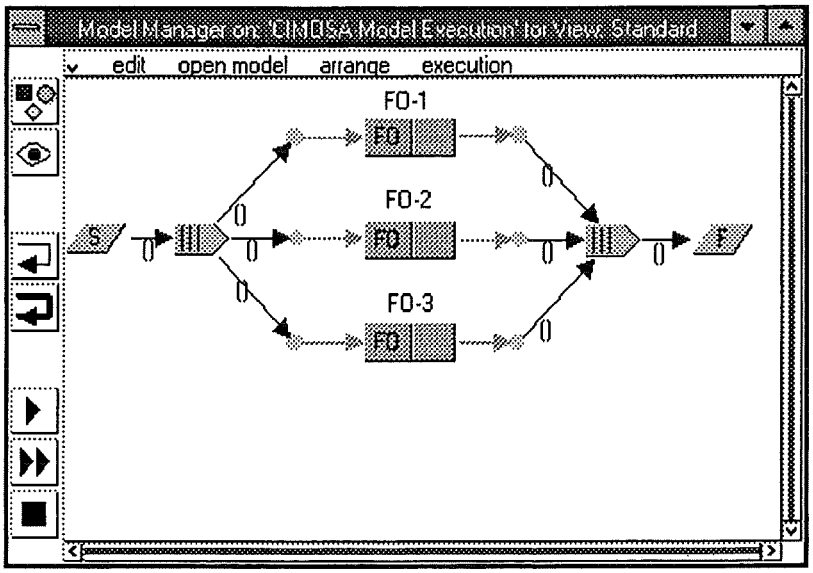


Abb. 82. Die Struktur der Enterprise Activity "EA"

Alle Enterprise Functions (*DP*, *EA*, *FO1*, *FO2* und *FO3*) werden als Module definiert. Die Procedural Rules, welche die Enterprise Functions verbinden, werden als Transitionen definiert. Um die Verständlichkeit der Darstellung des CIMOSA-Modells zu verbessern, werden die definierten Petri-Netz-Konstrukte mit entsprechenden Ikonen versehen.

Das Modul *CIMOSA Services* wird auf zwei weitere Module, *Business Service* und *Presentation Service*, unterteilt. Das Modul *Business Service* wird weiter in zwei Module, *Resource Management* und *Activity Control*, unterteilt (Abb. 83).

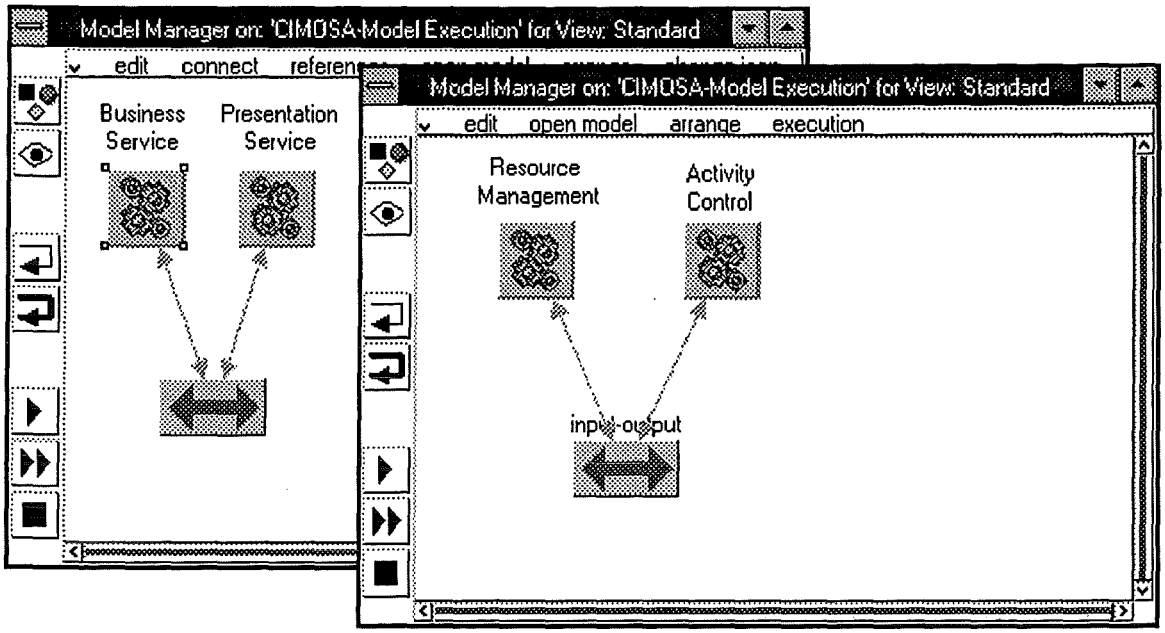


Abb. 83. Die Struktur der Module "CIMOSA Services" und "Business Service"

Das Modul *Resource Management* dient zur Verwaltung der Betriebsmittel, die bei der Ausführung der CIMOSA-Modelle benötigt werden. Das Modul *Activity Control* dient zur Ausführung der CIMOSA-Modelle. Das Modul *Presentation Service* simuliert den Dialog mit den Betriebsmitteln, die bei der Ausführung des CIMOSA-Modells benötigt werden. Hier wird nur die Struktur des Moduls *Activity Control* (Abb. 84) gezeigt.

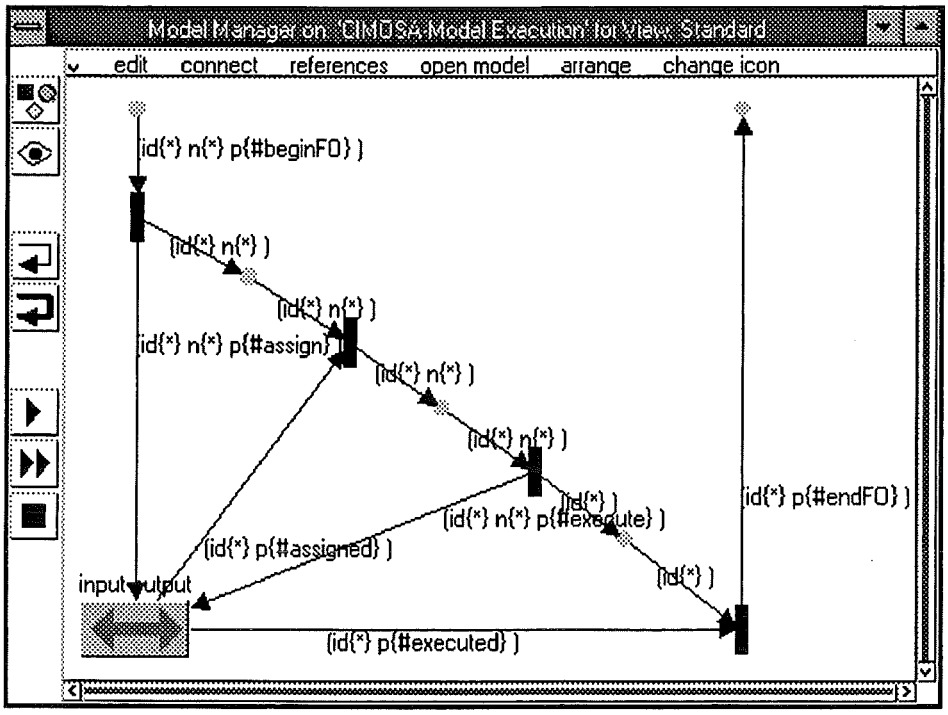


Abb. 84. Die Struktur des Moduls "Activity Control".

Das Element, das durch die Ikone mit dem Doppelpfeil dargestellt ist, ist eine Stelle, die zur Kommunikation (durch den Austausch von Token) mit anderen Diensten (z.B. Presentation Service) und innerhalb des Business Services dient. Durch das Schalten der ersten Transition wird ein Token[28] in der Kommunikationsstelle abgelegt, um dem Ressource Management die Zuweisung eines Betriebsmittels für eine Functional Operation zu befehlen. Nach der Zuweisung legt das Resource Management einen Token in die Kommunikationsstelle zurück, so daß die zweite Transition in der Activity Control schalten kann. Danach wird die dritte Transition geschaltet, die einen Token in die Kommunikationsstelle ablegt, um dem Presentation Service zu befehlen, den entsprechenden Dialog mit den Maschinen zur Ausführung der Functional Operation durchzuführen. Nachdem die Functional Operation ausgeführt wurde, legt der Presentation Service einen Token in die Kommunikationsstelle zurück, so daß die vierte Transition in der Activity Control schalten kann. Damit ist die Ausführung der Functional Operation beendet.

[28] Die Attribute der Token und der Kanten haben folgende Bedutung: id - Identifikator einer Functional Operation, n - Name einer Functional Operation, p - ein zusätzlicher Parameter. Das Symbol * in einem Kanten-Attribut beduetet, daß das entsprechende Token-Attribut einen beliebiegen Wert haben kann.

Die Eingangsstelle der ersten Transition und die Ausgangsstelle der vierten Transition werden in weiteren Verlauf der Entwicklung des Petri-Netz-Modells durch das Konstrukt *Pool* ersetzt, so daß die Anbindung des Business Services an das CIMOSA-Modell möglich wird. In den nächsten zwei Entwicklungsschritten wird dieses Konstrukt definiert und das Petri-Netz-Modell vervollständigt.

In der Abb. 85 wird die hierarchische Struktur des bis jetzt erstellten Petri-Netz-Modells (in der *Tree*-Sicht) dargestellt.

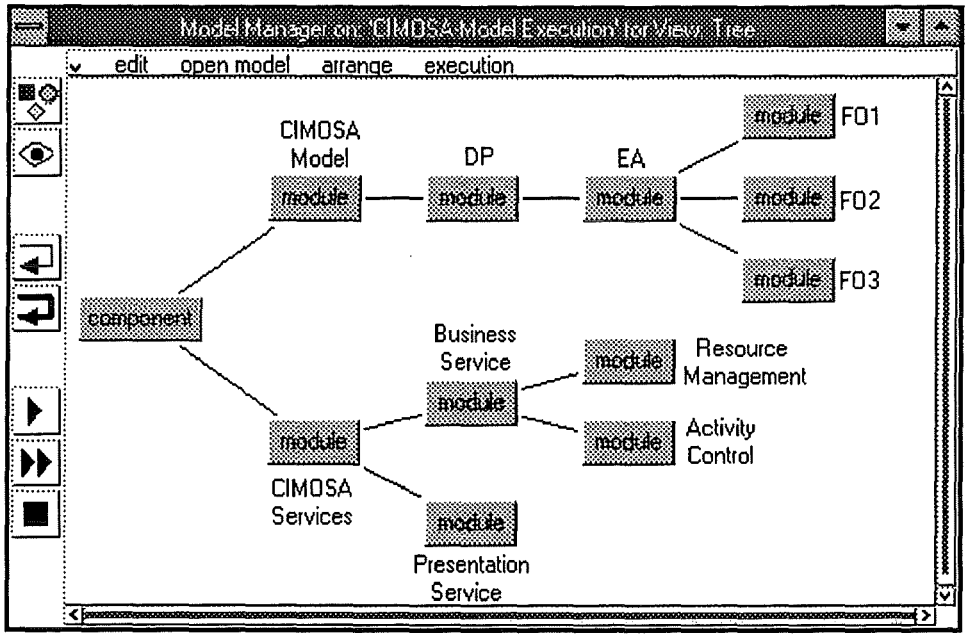


Abb. 85. Die hierarchische Struktur des Petri-Netz-Modells

7.2.4 Schritt 3: Erweiterung der Petri-Netz-Methode

In diesem Schritt wird die bereits definierte Petri-Netz-Methode erweitert, indem das zusätzliche Konstrukt *Pool* eingeführt wird. Da die Eigenschaften dieses Konstruktes ähnlich wie die Eigenschaften des Konstruktes *Place* sind, unterscheiden sich die Definitionen der beiden Konstrukte kaum. Für das Konstrukt *Pool* wir lediglich ein anderes Ausgabe-Sicht-Modell definiert (siehe Tab. 35)

Node-Instanz für *Pool*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Pool*
ViewModel	ViewOutputModel	In der *Standard*-Sicht:  In der *Tree*-Sicht: • nicht sichtbar
	ViewInputModel	in der *Standard*-Sicht: • ein zusätzlicher Menüeintrag: "change icon" in der *Tree*-Sicht: • keine zusätzlichen Eingabeelemente
RulesModel	InputConnectionRules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Transition* durch *Arc* • mit *Modul* durch *Arc* mögliche Verbindungen in der *Tree*-Sicht: • keine
	OutputConnection-Rules	mögliche Verbindungen in der *Standard*-Sicht: • mit *Transition* durch *Arc* • mit *Modul* durch *Arc* mögliche Verbindungen in der *Tree*-Sicht: • keine
	HorizontalReferences-Rules	keine (in beiden Sichten)
	VerticalReferences-Rules	keine (in beiden Sichten)
ImplementationModel	InformationModel	Information Element *IconName* zum Speichern des Namens einer Ikone
	FunctionModel	Funktionen für: • Bereitstellung einer Liste der Token, die sich in der Stelle befinden • Zuweisung der Ikone (für den Menüeintrag "change icon")

Tab. 35. Die Node-Instanz "Pool"

Die wichtigste Eigenschaft des Konstruktes ist, daß ein Pool in verschiedenen Teilen eines Petri-Netz-Modells verwendet (und auch dargestellt) werden kann, er aber immer die selbe Stelle repräsentiert. Diese Eigenschaft wird durch die entsprechende Definition des Verbindungsregel-Modells (*VerticalReferenceRules*) der Baustein-Instanzen *PetriNetModel* und *Modul* erreicht. In der Definition der Regel für eine vertikale Referenz (*SpecialConnectionRule*, siehe dazu Kapitel 5.4.1.3) erhält das Attribut *Sole* den Wert *true*. Diese Definition der Regel wird im Werkzeug (im Modul *ModelManager*) so interpretiert, daß es möglich ist, nur eine Baustein-Occurrence der Baustein-Instanz *Pool* zu erzeugen, die aber in verschiedenen Teilen des System-Modells verwendet werden kann.

Die entsprechenden Erweiterungen der Definitionen der Baustein-Instanzen *PetriNetComponent* und *Modul* werden in den folgenden Tabellen präsentiert.

Component-Instanz für *PetriNetComponent*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*PetriNetComponent*
GeneralViewModel		keine Änderung
ViewModel	ViewOutputModel	keine Änderung
	ViewInputModel	keine Änderung
RulesModel	InputConnectionRules	keine Änderung
	OutputConnection-Rules	keine Änderung
	HorizontalReferences-Rules	keine Änderung
	VerticalReferences-Rules	**Erweiterung**: mögliche Verbindungen in der *Standard*-Sicht: • mit *Pool* durch *SubReference* (mit dem Attribute *Sole=true*)
ImplementationModel	InformationModel	keine Änderung
	FunctionModel	keine Änderung

Tab. 36. Erweiterung der Definition der Component-Instanz "PetriNetComponent"

Node-Instanz für *Modul*		
Attribute		**Werte bzw. Beschreibung der Werte der Attribute**
InstanceName		*Modul*
ViewModel	ViewOutputModel	keine Änderung
	ViewInputModel	keine Änderung
RulesModel	InputConnectionRules	keine Änderung
	OutputConnection-Rules	keine Änderung
	HorizontalReferences-Rules	keine Änderung
	VerticalReferences-Rules	**Erweiterung**: mögliche Verbindungen in der *Standard*-Sicht: • mit *Pool* durch *SubReference* (mit dem Attribute *Sole=true*)
ImplementationModel	InformationModel	keine Änderung
	FunctionModel	keine Änderung

Tab. 37. Erweiterung der Definition der Node-Instanz "Modul"

7.2.5 Schritt 4: Vervollständigung der Petri-Netz-Modelle

Nach der Erweiterung der Petri-Netz-Methode wird das bereits entwickelte Petri-Netz-Modell durch den Einsatz des neuen Konstruktes vervollständigt. Die Vervollständigung dient zur Realisierung der Anbindung der CIMOSA-Modelle an das Business Service, wie im Kapitel 7.2.1.2 beschrieben.

In dem CIMOSA-Modell werden die bereits existierenden Functional Operations, die als Petri-Netz-Module definiert wurden, entsprechend verfeinert. Die verfeinerte Struktur einer Functional Operation stellt die Abb. 86 dar.

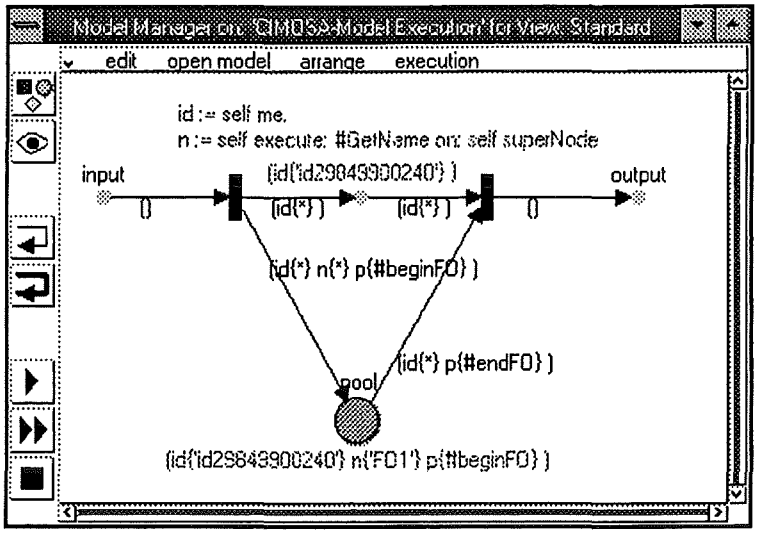


Abb. 86. Die verfeinerte Struktur einer Functional Operation

Die Abbildung stellt die Situation nach dem Feuern der ersten Transition dar. Durch das Feuern dieser Transition wurde ein Token in der mittleren Stelle und ein Token in dem Pool abgelegt. Das Token im Pool (mit dem Identifikator und Namen der Functional Operation und mit dem Parameter *#beginFO*) repräsentiert ein Ereignis, mit dem die Ausführung der Functional Operation in dem Business Service initiiert wird.

Um den Token empfangen zu können, wird die Struktur des Business Services (des Moduls *ActivityControl*) durch Verwendung des Pools entsprechend geändert (Abb. 87)

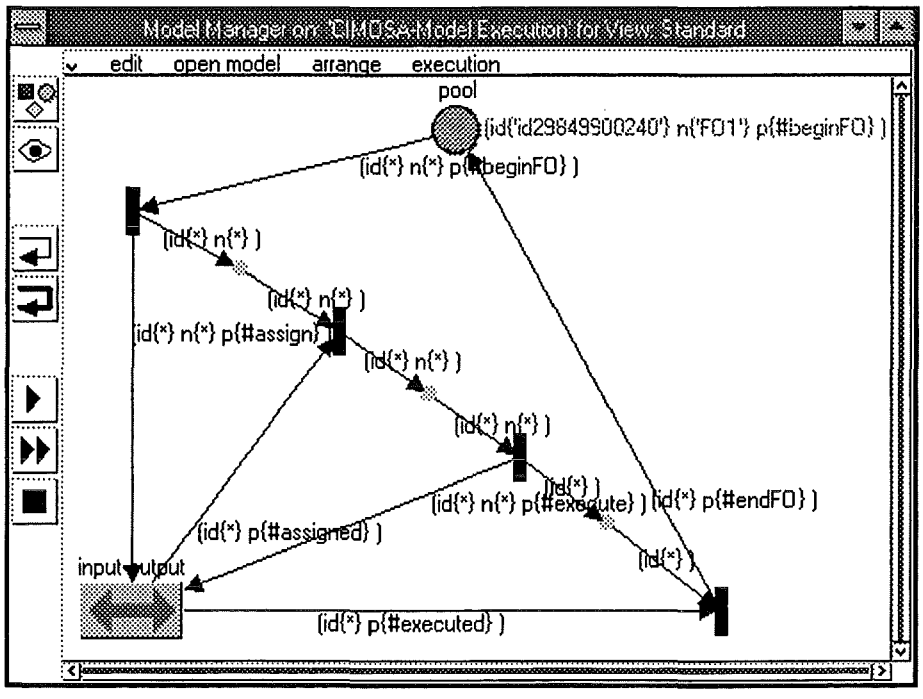


Abb. 87. Die geänderte Struktur des Moduls "ActivityControl"

Das Token erscheint nach dem Ablegen im Pool in einer Functional Operation auch im Pool des Business Service. Danach werden im Business Service die entsprechenden Aktionen ausgeführt (siehe dazu Kapitel 7.2.3). Nach der Ausführung dieser Aktionen wird ein neues Token (mit dem Identifikator der Functional Operation und dem Parameter *#endFO*) im Pool des Business Service abgelegt. Dieses Token erscheint auch im Pool in der Functional Operation. Die zweite Transition in der Functional Operation wird gefeuert, was das Ende der Ausführung dieser Functional Operation bedeutet.

Der Pool wird in allen Functional Operation verwendet. Dies bedeutet, daß der Token, der vom Business Service im Pool abgelegt wird, in allen Functional Operations erscheint. Die Verwendung von Identifikatoren der Functional Operations gewährleistet jedoch, daß nur die betroffene Functional Operation diesen Token vom Pool entfernt. Der zusätzliche Parameter (mit dem Wert *#beginFO* oder *#endFO*) garantiert dabei die korrekte Richtung der Kommunikation zwischen den Functional Operations und dem Business Service. Bei dem Parameter mit dem Wert *#beginFO* kann der Token aus dem Pool nur vom Business Service entfernt werden. Bei dem Parameter mit dem Wert *#endFO* kann der Token nur von einer (durch den Identifikator bestimmten) Functional Operation entfernt werden. Der in dem Token gespeicherte Name der Functional Operation wird vom Business Service nicht verwendet. Der Name wird an den Presentation Service weitergeleitet, damit die für die mit dem Namen gekennzeichnete Functional Operation vorgesehenen Aktionen von den Betriebsmitteln ausgeführt werden können.

8. Zusammenfassung und Ausblick

Heutzutage werden in den Unternehmen zunehmend zahlreiche computerunterstützte Anwendungen in verschiedenen Bereichen eingesetzt. Diese Anwendungen werden in der Regel in ein unternehmensweites bereichsübergreifendes System integriert. In solchen Systemen, die mit dem Begriff CIM (Computer Integrated Manufacturing) bezeichnet werden, spielt die Integration von Funktionen und Informationen eine wesentliche Rolle.

Diese Integration wird durch den Einsatz von Unternehmensmodellen und einer Integrationsplattform unterstützt. Mit Hilfe von Unternehmensmodellen können Funktionen, Daten, Organisation und Betriebsmittel eines Unternehmens beschrieben werden, um damit eine reibungslose Kooperation zwischen einzelnen CIM-Komponenten (z.B. PPS, CAD/CAM) zu gewährleisten. Die Integrationsplattform definiert die Schnittstellen, Datenformate, Kommunikationsprotokolle und Dienste zur Unterstützung des Informationsaustausches zwischen diesen Komponenten.

Es existieren bereits zahlreiche Ansätze zur Unternehmensmodellierung. Bei diesen Ansätzen wird meistens eine Modellierungsmethode eingesetzt, die verschiedene Basismethoden, wie z.B. SADT, Petri-Netze, Entity Relationship, kombiniert. Die Verwendung dieser unterschiedlichen Beschreibungsmethoden erlaubt, ein Unternehmen unter verschiedenen Aspekten (sogenannten Sichten) zu analysieren und zu beschreiben. Die meist betrachteten Sichten sind Funktions- und Daten-Sicht (wie z.B. im Information-Engineering-Konzept). Es werden oft auch weitere Aspekte betrachtet, wie z.B. organisatorische Aspekte (INCOME) oder Steuerungsaspekte (ARIS). In den meist umfangreichen Konzepten zur Unternehmensmodellierung (wie z.B. CIMOSA) werden bei der Beschreibung auch Betriebsmittel und Wirtschaftlichkeitsaspekte berücksichtigt.

Das CIMOSA-Konzept umfaßt außer der Modellierungsmethode auch die Definition der Integrationsplattform. Es werden dabei verschiedene Dienste und Kommunikationsprotokolle definiert, die den Informationsaustausch zwischen den CIM-Komponenten unterstützen und die Ausführung der mit der CIMOSA-Modellierungsmethode entwickelten Unternehmensmodelle ermöglichen.

Obwohl das CIMOSA-Konzept nicht vollständig definiert war, wurden parallel zu der Entwicklung des Konzeptes Arbeiten im Rahmen des ESPRIT-Projektes VOICE geführt, welche die Validierung des Konzeptes (also der Modellierungsmethode und der Integrationsplattform) als Ziel haben. Da die endgültige und vollständige Definition des Konzeptes nicht vorlag, mußte die

Validierung durch Entwicklung von Prototypen erfolgen, mit denen nur Teile des Konzeptes schrittweise validiert wurden. Hier wurden zwei Typen der Prototypenentwicklung durchgeführt:

- Entwicklung von beispielhaften Unternehmensmodellen (die als Prototypen der Unternehmensmodelle gesehen werden können), um die dafür eingesetzte Modellierungsmethode zu validieren
- Entwicklung von Prototypen der Softwarekomponenten der Integrationsplattform, um die Spezifikation dieser Softwarekomponenten zu validieren.

Bei dieser Prototypenentwicklung können folgende Situationen auftreten:

- die Modellierungsmethode (als Bestandteil des Konzeptes) ändert sich
- für die Validierung bestimmter Teile der Spezifikation der Softwarekomponenten wurde ein Prototyp mit einer Softwareentwicklungsmethode realisiert. Für die Validierung weiterer Teile der Spezifikation wird die nächste Version des Prototyps realisiert. Es kann vorkommen, daß dies nicht mehr mit der bis dahin eingesetzten Softwareentwicklungsmethode zu realisieren ist, da diese Methode nicht ausreichend "mächtig" ist.

Dies bedeutet, daß die Entwicklung der Prototypen (Unternehmensmodelle oder Softwarekomponenten) mit einer neuen Version der bis jetzt eingesetzten Methode (Modellierungs- oder Softwareentwicklungsmethode) fortgeführt werden muß. Die Erfahrungen, die bei der Validierung des CIMOSA-Konzeptes erworben wurden, zeigen, daß dies mit verfügbaren Werkzeugen nicht möglich bzw. nur sehr schwierig zu realisieren ist. Es existieren keine Ansätze (also auch keine Werkzeuge), welche die Änderungen (bzw. Erweiterungen) einer Methode während der mit dieser Methode durchgeführten Entwicklung unterstützen.

Zur Unterstützung der Änderungen der Methode während der Prototypenentwicklung wurde ein Konzept des zweidimensionalen Lebenszyklus eines Systems ausgearbeitet. Der Begriff System wird hier dabei als Sammelbegriff für die Bezeichnung entweder eines Unternehmensmodells oder einer Softwarekomponente verwendet. Die Entwicklung eines Systems (bzw. der Prototypen eines Systems), die auf diesem Lebenszyklus basiert, erfolgt in einem zweidimensionalem Raum. Dieser "Entwicklungsraum" wird durch zwei orthogonale Achsen bestimmt:

- die Achse der sog. Vollständigkeit des Systems - die Vollständigkeit beschreibt den Erfüllungsgrad der an das System gestellten Anforderungen, den die nacheinander folgenden Prototypen des Systems aufweisen,
- die Achse der sog. Mächtigkeit der Methode - die Mächtigkeit beschreibt die Angemessenheit der Methode bezüglich der Anforderungen, die an das mit der Methode zu entwickelndem System gestellt werden.

Der Verlauf der Entwicklung (als Entwicklungspfad bezeichnet) wird durch die Positionen der nacheinander folgenden Prototypen und der Versionen der Methode in dem zweidimensionalem Entwicklungsraum bestimmt.

Zur Unterstützung des zweidimensionalen Lebenszyklus wurde ein Baustein-Konzept entwickelt. Ähnlich wie bei den Meta-Modellierungs-Konzept werden hier drei Konkretisierungsebenen unterschieden:

- Meta-Methode-Ebene - auf der Ebene werden Konstrukte (sog. statische Baustein-Templates) einer allgemeinen Methode definiert, mit der eine spezifische Methode entwickelt wird
- Methode-Ebene - auf der Ebene stehen die Konstrukte (sog. Baustein-Instanzen und dynamische Baustein-Templates) der entwickelten spezifischen Methode zur Verfügung. Diese Konstrukte werden für die Entwicklung eines Systems (bzw. der Prototypen eines Systems) eingesetzt
- System-Ebene - auf der Ebene befinden sich die mit der spezifischen Methode entwickelten Prototypen des Systems (das durch eine Menge der sog. Baustein-Occurrences beschrieben wird). Hier kann die Validierung der Spezifikation des System z.B. durch Simulation erfolgen.

Die existierenden Meta-Modellierungs-Ansätze (Meta-CASE Tools) haben den Nachteil, daß die Änderungen der spezifischen Methode einen Bruch in den Verlauf des Entwicklungsprozesses einführen. Der Grund dafür ist eine ausgeprägte Abhängigkeit der Form der Beschreibung eines Systems von der Methode, die für die Erstellung dieser Beschreibung eingesetzt wird. Nach der Änderung der Methode muß daher eine Konvertierung der bis dahin erstellten Beschreibung des Systems erfolgen. In dem Baustein-Konzept ist diese Abhängigkeit zu einem notwendigem Minimum reduziert. Damit haben die Änderungen der Methode keine bzw. eine geringe Auswirkung auf die Beschreibung des Systems, so daß die Konvertierung der Beschreibung nach jeder Methodenänderung wegfällt. Die Prototypenentwicklung wird dadurch beschleunigt und der Zeitaufwand bei der Validierung wesentlich reduziert.

Für die Definition der in dem Baustein-Konzept verwendeten Konstrukte wurde die formale, standardisierte Beschreibungssprache EXPRESS eingesetzt. Dies erlaubt u.a. eine standardisierte Definition und Dokumentation der Konstrukte des Baustein-Konzeptes und der durch den Einsatz dieses Konzeptes entwickelten spezifischen Methoden und Beschreibungen der Systeme. Im Hinblick auf die möglichst große Flexibilität bei der Entwicklung der Methoden könnte die Definition der in dem Baustein-Konzept spezifizierten Konstrukte erweitert werden. Hier könnte z.B. die Möglichkeit geschaffen werden, abstrakte Methodenkonstrukte zu definieren, von denen entsprechende spezialisierte Konstrukte abgeleitet würden. Durch die Vererbung der

Eigenschaften der abstrakten Konstrukte, könnte die Entwicklung der spezialisierten Konstrukte beschleunigt werden, da unnötige wiederholte Definitionen der gleichen Eigenschaften vermieden werden könnten.

Zur Unterstützung des Baustein-Konzeptes und damit des zweidimensionalen Lebenszyklus wurde der Prototyp eines Werkzeugs entwickelt. Die Implementation des Werkzeugs erfolgte mit Hilfe der objektorientierten Sprache Smalltalk-80. Es wurden mehrere Module mit entsprechenden Benutzerschnittstellen implementiert, welche die Definition der Methoden und die Entwicklung der Systeme in einer komfortablen Weise ermöglichen. Es wurden dabei nur die wichtigsten Module entwickelt, mit denen die Realisierbarkeit des Baustein-Konzeptes nachgewiesen werden konnte. Hier könnte die Implementation weiter geführt werden, um weitere nützliche Module des Werkzeugs zu entwickeln und damit die Benutzerfreundlichkeit des Werkzeugs weiter zu steigern. Da die Funktionalität der vorhandenen Module beachtlich komplex und nicht weiter reduzierbar ist, mußte bei den Implementation die "Quick-And-Dirty"-Vorgehensweise eingesetzt werden. Dadurch konnte nicht die optimale Effizienz des Werkzeugs erreicht werden. Die weiteren Implementationsarbeiten könnten daher auch die Optimierung der vorhandenen Implementation als Ziel haben.

Die prototypische Implementation des Werkzeuges wurde mit Hilfe zwei Testszenarios validiert. Im ersten Szenario wurde eine vereinfachte Version der CIMOSA-Modellierungsmethode definiert. Mit dieser Methode wurde ein beispielhaftes Modell entwickelt, das die Papierherstellung in einem mittelständigen deutschen Unternehmen darstellt. Im zweiten Szenario wurde eine Petri-Netz-Methode definiert, die das Erstellen und die Simulation der Pr/T-Netze (Prädikat/Transitionsnetze) ermöglicht. Mit dieser Methode wurde eine Softwarekomponente entwickelt, welche die vereinfachte Funktionalität eines Dienstes der Integrationsplattform des CIMOSA-Konzeptes implementiert. In beiden Testszenarios wurden die entsprechenden Methoden während der Entwicklung der Prototypen schrittweise erweitert. Die Erweiterungen umfaßten die Definition von neuen Methodenkonstrukten und die Definition von neuen Sichten, mit denen zusätzliche Aspekte bei der Beschreibung eines Systems berücksichtigt werden konnten. Mit Hilfe dieser Szenarios konnte die Anwendbarkeit des Konzeptes des zweidimensionalen Lebenszyklus gezeigt werden, indem die Möglichkeit der Methodenänderung während des Entwicklungsprozesses eines Systems nachgewiesen werden konnte.

Der im Rahmen dieser Arbeit entwickelte Ansatz stellt einen Beitrag zur Effizienzsteigerung bei der Entwicklung von Prototypen im Bereich der Unternehmensintegration. Der Ansatz könnte

jedoch auch in anderen Bereichen eingesetzt werden, wo die schnelle Prototypenentwicklung von Softwarekomponenten bzw. Beschreibungsmodellen notwendig ist. Bei diesen Einsatzbereichen wäre die Auswahl der Beschreibungsmethode weniger kritisch, da diese Methode dann zu jedem Zeitpunkt erweitert und an die neuen Anforderungen angepaßt werden könnte. Ein anderer denkbarer Einsatzbereich wäre die prototypische Entwicklung der CASE-Tools. In diesem Fall würde das im Rahmen dieser Arbeit entwickelte Werkzeug als herkömmliches Meta-CASE-Tools verwendet. Die Verwendung des Werkzeugs als ein flexibles CASE-Tool mit zahlreichen vordefinierten Methoden könnte ein weiterer Einsatzbereich sein.

9. Literatur

[Agr86] Agresti, W.W. (Ed.), *New Paradigms for Software Development*, IEEE Computer Society Press, Washington D.C., 1986.

[Agr86a] Agresti, W.W., *The Conventional Software Life-Cycle Model: Its Evolution and Assumptions*, in *New Paradigms for Software Development*, Agresti, W.W. (Ed.), IEEE Computer Society Press, Washington D.C., 1986.

[Agr86b] Agresti, W.W., *What Are the New Paradigms?*, in *New Paradigms for Software Development*, Agresti, W.W. (Ed.), IEEE Computer Society Press, Washington D.C., 1986.

[Ald91] Alderson, A., *Meta-CASE Technology*, Proceedings of European Symposium on Software Development Environments and CASE Technology, Königswinter, June 1991, Springer-Verlag, Berlin, Heidelberg, 1991.

[AMICE93] ESPRIT Consortium AMICE (Eds.), *CIMOSA - Open System Architecture for CIM*, Springer-Verlag, Berlin, Heidelberg, 1993.

[Asl91] Aslett, M.J. (Ed.), *A Knowledge Based Approach to Software Development: ESPRIT Project ASPIS*, Elsevier Science Publishers B.V., Amsterdam, 1991.

[AWF85] *Integrierter EDV-Einsatz in der Produktion - Begriffe, Definitionen, Funktionszuordnungen*, AWF - Ausschuß für Wirtschaftliche Fertigung e.V., Eschborn, 1985.

[Bac60] Backus, J.W., *The Syntax and Semantics of the Proposed International Algebraic Language of the Zürich ACM-GAMM Conference*, Proceedings of International Conference on Information Processing, Paris, France, June 1960, R. Oldenburg Verlag, München, 1960.

[Bal93] Balzert, H. (Hrsg.), *CASE. Systeme und Werkzeuge*. BI Wissenschaftsverlag, Mannheim, 1993.

[BCG83] Balzer, R., Cheatham, T.E., Green, C., *Software Technology in 1990's: Using a New Paradigm*, Computer, Nov. 1983, 39-45.

[Boe76] Boehm, B.W., *Software Engineering*, IEEE Transactions on Computers 25, 12 (Dec. 1976), 1226-1241.

[Bog92] Bogdanowicz, L., *MMS Applications under MS-Windows for CIMOSA Demonstrator*, Proceedings of VDI/EMUG Workshop, München, October 1992.

[Bog93] Bogdanowicz, L., *Unveröffentlichter Bericht*, Kernforschungszentrum Karlsruhe GmbH, 1993.

[Boo86] Booch, G., *Object Oriented Development*, IEEE Transactions on Software Engineering 12, 2 (Feb. 1986), 211-221.

[Bro87] Brooks, F.P., *No Silver Bullet, Essence and Accidents of Software Engineering*, Computer 20, 4 (Apr. 1987), 10-19

[Bro93] Brouwer, J., *CIMOSA Tool Integration*, ESPRIT-VOICE Deliverable, 1993

[Che90] Chen, P., *Entity-Relationship Approach to Data Modelling*, in *System and Software Requirements Engineering*, Thayer, R.H., Dorfman, M. (Eds.), IEEE Computer Society Press, Los Alamitos, USA, 1990.

[CoY90] Coad P., Yourdon E.: *Object-Oriented Analysis*, in *System and Software Requirements Engineering*, Thayer, R.H., Dorfman, M. (Eds.), IEEE Computer Society Press, Los Alamitos, USA, 1990.

[CoY91] Coad, P., Yourdon, E., *Object-Oriented Design*, Yourdon Press, Englewood Cliffs, USA, 1991.

[CPN89] *Design/CPN: A Tool Package Supporting Colored Petri Nets*, Meta Software Corporation, Cambridge, USA, 1989.

[Cro90] Cronjäger, L., *Bausteine für die Fabrik der Zukunft. Eine Einführung in die rechnerintegrierte Produktion (CIM)*, Springer-Verlag, Berlin, Heidelberg, 1990.

[Dav88] Davis, A.M., *A Comparison of Techniques for the Specification of External System Behaviour*, Communication of the ACM 31, 9 (Sept. 1988), 1098-1115.

[DeM79] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, Englewood Cliffs, USA, 1979.

[Did92] Didic, M., *Rapid Prototyping for MAP/MMS based CIM-OSA Environments*, Proceedings of 3rd Int. Workshop on Rapid System Prototyping, Research Triangle Park, NC USA, IEEE Computer Society Press, June 1992.

[DiN91] Didic, M., Neuscheler, F., *The Use of Tools in VOICE*, ESPRIT-VOICE Deliverables, Juni 1991.

[DMR79] Dickover, M.E., McGovan, C.L., Ross, D.T., *Software Design Using SADT*, in *Tutorial: Software Design Strategies*, Bergland, G.D., Gordon, R.D. (Eds.), IEEE Computer Society, Long Beach, USA, 1979.

[DNB93] Didic, M. Neuscheler, F., Bogdanowicz, L., Klittich, L., *McCIM: Execution of CIMOSA Models*, CIM Europe Annual Conference Proceeding, Amsterdam, 1993.

[Fle94] Fleischmann, A., *Distributed Systems. Software Design & Implementation*, Springer-Verlag, Berlin, Heidelberg, 1994.

[Gen86] Genrich, H.J., *Predicate/Transition Nets*, in *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I*, Brauer, W., Reisig, W., Rosenberg, G., Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, Berlin, Heidelberg, 1987.

[GoR89] Goldberg, A., Robson, D., *Smalltalk-80: The Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1989.

[GKR92] Grobel, T., Kilger, C., Rude, S., *Objektorientierte Modellierung der Produktionsorganisation*, in *Information als Produktionsfaktor*, Görke, W., Rininsland, H., Syrbe, M. (Hrsg.), SpringerVerlag, Berlin, Heidelberg, 1992.

[GT91] *GraphTalk: Metamodeling Reference Guide*, Rank Xerox France, Dez. 1991.

[Hig78] Higgins, D.A., *Structured Programming with Warnier-Orr Diagrams*, in *Tutorial: Software Design Strategies*, Bergland, G.D., Gordon, R.D. (Eds.), IEEE Computer Society, Long Beach, USA, 1979.

[ICAM81] *Integrated Computer Aided Manufacturing (ICAM)*, Materials Laboratory, Air Force Wright Aeronautical Laboratories, Ohio, 1981.

[IDEF89] *Design/IDEF: Manual Reference*, Meta Software Corporation, Cambridge, USA, 1989.

[ISO92a] *ISO/DIS 10303-11 Industrial Automation Systems - Product Data Representation and Exchange. Part 11: Description methods: The EXPRESS Language Reference Manual*, ISO/IEC, 1992.

[ISO92b] *ISO/DIS 10303-21 Industrial Automation Systems - Product Data Representation and Exchange. Part 21: Implementation methods: Clear Text Encoding of the Exchange Structure*, ISO/IEC, 1992.

[Jac75] Jackson, M.A., *Priciples of Program Design*, Academic Press, London, UK, 1975.

[Jen86] Jensen, K., *Coloured Petri Nets*, in *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I*, Brauer, W., Reisig, W., Rosenberg, G., Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, Berlin, Heidelberg, 1987.

[JOS94] Jaeschke, P., Oberweis, A., Stucky, W., *Deriving Complex Structured Object Types for Business Process Modelling*, Proceedings of the 13th International Conference on Entity Relationship Approach, Manchester, Dec. 1994.

[Loy89] Loy, P.H., *A Comparision of Object-Oriented and Structured Development Methods*, Proceedings of Pacific Northwest Software Quality Conference, Sept. 1989.

[Mar89] Martin, J., *Information Engineering: Book 2: Planning and Analysis*, Prentice Hall, Englewood Cliffs, USA, 1989.

[Neu94] Neuscheler, F., Spath, D., *The Economic View: A Concept Using Benchmarks to Analyse, Evaluate and Optimize Business Processes*, IMSE'94 Conference Proceedings, Grenoble, 12-14 Dezember 1994.

[Neu95] Neuscheler, F., *Ein integrierter Ansatz zur Analyse und Bewertung von Geschäftsprozessen*, Dissertation, Universität Karlsruhe, 1995.

[OSS94] Oberweis, A., Scherrer, G., Stucky, W., *INCOME/STAR: Methodology and Tools for the Development of Distributed Information Systems*, Information Systems, Vol. 19, No. 8, 1994, 643-660.

[PACE93] *PACE, Tool Reference Manual, Version 2.1.1.*, Grossenbacher Elektronik AG, St. Gallen, Schweiz, 1993.

[Rei86] Reisig, W., *Petrinetze: Eine Einführung*, Springer-Verlag, Berlin, Heidelberg, 1986.

[Roy70] Royce, W.W., *Managing the Development of Large Software Systems: Concepts and Techniques*, Proceedings, WESCON, Aug. 1970.

[Sch89] Schönthaler, F., *Rapid Prototyping zur Unterstützung des konzeptuellen Entwurfs von Informationssystemen*, Dissertation, Universität Karlsruhe, 1989.

[Sch91] Schneider, H.-J., *Lexikon der Informatik und Datenverarbeitung*, R. Oldenburg Verlag, München, Wien, 1991.

[Sch92] Scheer, A.-W., *Architektur integrierter Informationssysteme*, Springer-Verlag, Berlin, Heidelberg, 1992.

[ScN90] Schönthaler, F., Németh, T., *Software-Entwicklungswerkzeuge: Methodische Grundlagen*, B.G. Teubner, Stuttgart, 1990.

[ScS83] Schalgeter, G., Stucky, W., *Datenbanksysteme: Konzepte und Modelle*, B.G. Teubner, Stuttgart, 1983.

[SNS89] Stucky, W., Németh, T., Schönthaler, F., *INCOME - Methoden und Werkzeuge zur betrieblichen Anwendungsentwicklung*, in *Interaktive betriebswirtschaftliche Informations- und Steuerungssysteme*, Kurbel, K., Mertens, P., Scheer, A.-W., (Hrsg.), Walter de Gruyter, Berlin, New York, 1989.

[Svo90] Svoboda, C.P., *Structured Analysis*, in *System and Software Requirements Engineering*, Thayer, R.H., Dorfman, M. (Eds.), IEEE Computer Society Press, Los Alamitos, USA, 1990.

[ThD90] Thayer, R.H., Dorfman, M. (Eds.), *System and Software Requirements Engineering*, IEEE Computer Society Press, Los Alamitos, USA, 1990.

[VaG91] Van Gigh, J.P., *System Design Modeling and Metamodeling*, Plenum Press, New York, 1991.

[WaM85] Ward, P., Mellor, S., *Structured Development for Real-Time Systems*, Vol. 1-3, Yourdon Press, Englewood Cliffs, USA, 1985.

[War81] Warnier, J.D., *Logical Construction of Systems*, Van Nostrand Reinhold Company, New York, 1981.

[Wir77] Wirth, N., *What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?*, Communications of the ACM, Nov. 1977, 822-823.

[Zav84] Zave, P., *The Operational Versus the Conventional Approach to Software Development*, Communications of the ACM, Feb. 1984, 104-118.

ANHANG A: Die vollständige Definition der statischen Baustein-Templates

SCHEMA BuildingBlockInstances;

(* Allgemeine Definitionen *)

```
TYPE CapStyle = ENUMERATION OF (capButt, capProjecting, capRound);
END_TYPE;

TYPE JoinStyle = ENUMERATION OF (joinBevel, joinMiter, joinRound);
END_TYPE;

ENTITY ColorValue;
        red     : REAL;
        green   : REAL;
        blue    : REAL;
END_ENTITY;

ENTITY GraphicsStyle;
        lineWidth : INTEGER;
        capStyle        : CapStyle;
        joinStyle       : JoinStyle;
        paintColor       ColorValue;
        fillColor       : OPTIONAL ColorValue;
END_ENTITY;

ENTITY Point;
        x       : REAL;
        y       : REAL;
END_ENTITY;

ENTITY Rectangle;
        origin  : Point;
        corner  : Point;
END_ENTITY;

TYPE Locus = LIST OF Point;
END_TYPE;

TYPE FlagsList = LIST OF BOOLEAN;
END_TYPE;

TYPE RunList = ARRAY [1 : 1] OF INTEGER;
END_TYPE;

TYPE RunValues = ARRAY [1 : 1] OF OPTIONAL STRING;
END_TYPE;
```

(* Definitionen der Ausgabeelemente für das Ausgabe-Sicht-Modell*)

```
ENTITY DisplayRectangle;
        style     : GraphicsStyle;
        rectangle : Rectangle;
END_ENTITY;

ENTITY DisplayDiamond;
        style     : GraphicsStyle;
        rectangle : Rectangle;
END_ENTITY;

ENTITY DisplayRoundedRectangle;
        style     : GraphicsStyle;
        rectangle : Rectangle;
END_ENTITY;

ENTITY DisplayCircle;
        style     : GraphicsStyle;
        center    : Point;
        radius    : REAL;
END_ENTITY;

ENTITY DisplayOval;
        style     : GraphicsStyle;
        center    : Point;
        radiusX   : REAL;
        radiusY   : REAL;
END_ENTITY;

ENTITY DisplayArc;
        style          : GraphicsStyle;
        boundingBox    : Rectangle;
        startAngle             : REAL;
        sweepAngle     : REAL;
END_ENTITY;

ENTITY DisplayLocus;
        style     : GraphicsStyle;
        locus     : Locus;
        flags     : FlagsList;
END_ENTITY;

ENTITY DisplayLine;
        style     : GraphicsStyle;
        locus     : Locus;
        flags     : FlagsList;
END_ENTITY;

ENTITY DisplayPolyline;
        style     : GraphicsStyle;
        locus     : Locus;
        flags     : FlagsList;
END_ENTITY;
```

```
ENTITY DisplaySpline;
        style       : GraphicsStyle;
        locus       : Locus;
        flags       : FlagsList;
END_ENTITY;

ENTITY DisplayBezier;
        style       : GraphicsStyle;
        locus       : Locus;
        flags       : FlagsList;
END_ENTITY;

ENTITY Text;
        string      : STRING;
        runs        : RunList;
        values      : RunValues;
END_ENTITY;

ENTITY DisplayText;
        style                   : GraphicsStyle;
        text                    : Text;
        firstIndent                     : INTEGER;
        restIndent              : INTEGER;
        rightIndent             : INTEGER;
        aligment                : INTEGER;
        lineGrid                : INTEGER;
        compositionWidth        : INTEGER;
        location                : Point;
END_ENTITY;

ENTITY Image;
        width           : INTEGER;
        height          : INTEGER;
        depth           : INTEGER;
        packedBits      : STRING;
        pallete         : STRING;
END_ENTITY;

ENTITY DisplayImage;
        style       : GraphicsStyle;
        image       : Image;
        location    : Point;
END_ENTITY;

ENTITY ObjectPositionPair;
        object      : DisplayElement;
        position    : Point;
END_ENTITY;

TYPE DisplayObjectList = LIST OF ObjectPositionPair;
END_TYPE;

ENTITY DisplayComposition;
        objects     : DisplayObjectList;
END_ENTITY;
```

```
TYPE DisplayElement = SELECT(
                DisplayRectangle, DisplayRoundedRectangle, DisplayDiamond, DisplayCircle,
                DisplayOval, DisplayArc, DisplayLine, DisplayPolyline, DisplaySpline,
                DisplayBezier, DisplayLocus, DisplayText, DisplayImage, DisplayComposition);
END_TYPE;
```

(* Definition des allgemeinen Sicht-Modells *)

```
TYPE ViewType = ENUMERATION OF (none, tree);
END_TYPE;

TYPE ViewProfile = ENUMERATION OF (horizontal, vertical);
END_TYPE;

ENTITY View;
        ViewName        : STRING;
        ViewType        : ViewType;
        ViewProfile     : ViewProfile;
        ViewDescription : STRING;
END_ENTITY;

TYPE GeneralViewModel = LIST OF View;
END_TYPE;
```

(* Definition des Sicht-Modells *)

```
ENTITY ViewModel;
        ViewOutputModel     : ViewOutputModel;
        ViewInputModel      : ViewInputModel;
END_ENTITY;
```

(* Definition des Ausgabe-Sicht-Modells *)

```
TYPE LayerArrangement = ENUMERATION OF (top, bottom);
END_TYPE;

TYPE RelativeArrangement = ENUMERATION OF (inside, bottom, top, right, left, ring, invisible);
END_TYPE;

ENTITY RelativeToArrangement;
        RelativeTo      : STRING;
        Arrangement     : RelativeArrangement;
END_ENTITY;

TYPE RelativeArrangementList = LIST OF RelativeToArrangement;
END_TYPE;

TYPE DisplayElementList = LIST OF DisplayElement;
END_TYPE;
```

```
ENTITY ViewOutput;
        ViewName              : STRING;
        IndividualPosition    : BOOLEAN;
        LayerArrangement      : OPTIONAL LayerArrangement;
        RelativeArrangement   : OPTIONAL RelativeArrangementList;
        OutputElements        : DisplayElementList;
END_ENTITY;

TYPE ViewOutputModel = LIST OF ViewOutput;
END_TYPE;
```

(* Definition des Eingabe-Sicht-Modells *)

```
TYPE MenuName = STRING;
END_TYPE;

TYPE MenuType = ENUMERATION OF (context, background, submenu);
END_TYPE;

TYPE MenuLabel = STRING;
END_TYPE;

TYPE MenuLabels = LIST OF MenuLabel;
END_TYPE;

TYPE ActionId = STRING;
END_TYPE;

TYPE MenuActions = LIST OF ActionId;
END_TYPE;

TYPE SeparatorPosition = INTEGER;
END_TYPE;

TYPE MenuSeparators = LIST OF SeparatorPosition;
END_TYPE;

TYPE StandardMenuAction = ENUMERATION OF (
                        select_all, select_nodes, select_arcs, clear, copy, cut, paste, duplicate, edit, refresh,
                         input_elements, output_elements, connect_input, connect_output, connect_ref,
                        open_subnet, open_supernet, fit_on_arcs, align_grid, front_nodes, front_arcs,
                        auto_arrange, arrange_format, grid_point);
END_TYPE;

TYPE StandardMenuActions = LIST OF StandardMenuAction;
END_TYPE;

ENTITY Menu;
        Name                  : MenuName;
        TypeOfMenu            : MenuType;
        Labels                : MenuLabels;
        Actions               : MenuActions;
        Separators            : MenuSeparators;
        BlockedStandardActions : StandardMenuActions;
END_ENTITY;
```

```
ENTITY ShortCut;
        ShortCutKey     : STRING;
        Action          : ActionId;
END_ENTITY;

TYPE MouseEvent = ENUMERATION OF (click, double_click, hold_down, release)
END_TYPE;

ENTITY MouseAction;
        Event     : MouseEvent;
        Action    : ActionId;
END_ENTITY;

TYPE ControlElement = SELECT(
                Menu, ShortCut, MouseAction);
END_TYPE;

TYPE ControlElementList = LIST OF ControlElement;
END_TYPE;

ENTITY ViewInput;
        ViewName        : STRING;
        InputElements   : ControlElementList;
END_ENTITY;

TYPE ViewInputModel = LIST OF ViewInput;
END_TYPE;
```

(* Definition des Verbindungsregel-Modells *)

```
ENTITY ConnectionRule;
        ViewName        : STRING;
        With            : STRING;
        By              : STRING;
END_ENTITY;

ENTITY SpecialConnectionRule;
        ViewName        : STRING;
        With            : STRING;
        By              : STRING;
        Autonomous      : BOOLEAN;
        Sole            : BOOLEAN;
END_ENTITY;

ENTITY RulesModel;
        InputConnectionsRules           : OPTIONAL LIST OF ConnectionRule;
        OutputConnectionsRules          : OPTIONAL LIST OF ConnectionRule;
        HorizontalReferencesRules       : OPTIONAL LIST OF ConnectionRule;
        VerticalReferencesRules         : OPTIONAL LIST OF SpecialConnectionRule;
END_ENTITY;
```

(* Definition des Implementations-Modells *)

```
ENTITY ImplementationModel;
        InformationModel        : OPTIONAL InformationModel;
        FunctionModel           : OPTIONAL FunctionModel;
END_ENTITY;
```

(* Definition des Information-Modells *)

```
ENTITY InformationElement;
        AccessName      : STRING;
        DataType        : STRING;
END_ENTITY;

TYPE InformationElementList = LIST OF InformationElement;
END_TYPE;

ENTITY Substitution;
        Name    : STRING;
        Link    : STRING;
END_ENTITY;

TYPE SubstitutionList = LIST OF Substitution;
END_TYPE;

ENTITY InformationModel;
        Types           : OPTIONAL STRING;
        Elements        : OPTIONAL InformationElementList;
        Substitutions   : OPTIONAL SubstitutionList;
END_ENTITY;
```

(* Definition des Funktions-Modells *)

```
TYPE EditEvent = ENUMERATION OF (
        pre_select, post_select, pre_unselect, post_unselect, pre_display, post_display, pre_move, post_move,
        pre_clear, post_clear, pre_copy, post_copy, pre_paste, post_paste, pre_duplicate, post_duplicate,
        post_create, pre_remove, pre_edit, post_edit, pre_connect_input, post_connect_input,
        pre_connect_output, post_connect_output, pre_link_to_ref, post_link_to_ref,
        pre_link_from_ref, post_link_from_ref, pre_link_sub_ref, post_link_sub_ref,
        pre_link_super_ref, post_link_super_ref, pre_open_subnet, post_open_subnet,
        pre_open_supernet, post_open_supernet);
END_TYPE;

TYPE ModelEvent = ENUMERATION OF (
        START, STOP, STEP,
        CREATED_NODE, CREATED_ARC, CREATED_ELEMENT,
        REMOVED_NODE, REMOVED_ARC, REMOVED_ELEMENT,
        created_input_node, removed_input_node, created_output_node, removed_output_node,
        created_to_ref_node, removed_to_ref_node, created_from_ref_node, removed_from_ref_node,
        created_sub_ref_node, removed_sub_ref_node, created_super_ref_node, removed_super_ref_node,
        changed_from_ref_node, post_create, pre_remove, changed_information_element, changed_position,
        pre_connect_input, post_connect_input, pre_connect_output, post_connect_output,
        pre_link_to_ref, post_link_to_ref, pre_link_from_ref, post_link_from_ref,
        pre_link_sub_ref, post_link_sub_ref, pre_link_super_ref, post_link_super_ref,
```

```
        pre_disconnect_input, post_disconnect_input, pre_disconnect_output, post_disconnect_output,
        pre_unlink_to_ref, post_unlink_to_ref, pre_unlink_from_ref, post_unlink_from_ref,
        pre_unlink_sub_ref, post_unlink_sub_ref, pre_unlink_super_ref, post_unlink_super_ref);
END_TYPE;

TYPE EventId = STRING;
END_TYPE;

TYPE CodeText = STRING;
END_TYPE;

ENTITY EditEventHandler;
        Event     : EditEvent;
        Code      : CodeText;
END_ENTITY;

ENTITY UserActionHandler;
        Action    : STRING;
        Code      : CodeText;
END_ENTITY;

ENTITY ModelEventHandler;
        Event     : ModelEvent;
        Code      : CodeText;
END_ENTITY;

ENTITY ExecutionEventHandler;
        Event     : STRING;
        Code      : CodeText;
END_ENTITY;

TYPE EditEventHandlerList = LIST OF EditEventHandler;
END_TYPE;

TYPE UserActionHandlerList = LIST OF UserActionHandler;
END_TYPE;

TYPE ModelEventHandlerList = LIST OF ModelEventHandler;
END_TYPE;

TYPE ExecutionEventHandlerList = LIST OF ExecutionEventHandler;
END_TYPE;

ENTITY FunctionModel;
        EditEventHandlers       : OPTIONAL EditEventHandlerList;
        UserActionHandlers      : OPTIONAL UserActionHandlerList;
        ModelEventHandlers      : OPTIONAL ModelEventHandlerList;
        ExecutionEventHandlers  : OPTIONAL ExecutionEventHandlerList;
END_ENTITY;
```

(* Definitionen der statischen Baustein-Templates *)

```
ENTITY Component;
        InstanceName            : STRING;
        OccurrenceSchemaName    : STRING;
        MethodId                        : STRING;
        GeneralViewModel        : eneralViewModel;
        ViewModel               : ViewModel;
        RulesModel              : RulesModel;
        ImplementationModel     : ImplementationModel;
END_ENTITY;

ENTITY Node;
        InstanceName            : STRING;
        ViewModel               : ViewModel;
        RulesModel              : RulesModel;
        ImplementationModel     : ImplementationModel;
END_ENTITY;

ENTITY Arc;
        InstanceName            : STRING;
        ViewModel               : ViewModel;
        ImplementationModel     : ImplementationModel;
END_ENTITY;

END_SCHEMA;
```

ANHANG B: Die vollständige Definition der dynamischen Baustein-Templates für das erste Testszenario (CIMOSA-Modell)

```
SCHEMA CIMOSAModel_1_0;

(* Allgemeine Definitionen *)

TYPE ValueString = STRING;
END_TYPE;

TYPE UniqueId = STRING;
END_TYPE;

TYPE IdList = LIST OF STRING;
END_TYPE;

TYPE GhostPositions = LIST OF GhostPosition;
END_TYPE;

ENTITY Point;
        x          : REAL;
        y          : REAL;
END_ENTITY;

ENTITY GhostPosition;
   HauntedNode   : STRING;
   Position      : Point;
END_ENTITY;

ENTITY ConnectionModel;
   Inputs        : IdList;
   Outputs       : IdList;
   SubRefs       : IdList;
   SuperRefs     : IdList;
   ToRefs        : IdList;
   FromRefs      : IdList;
END_ENTITY;
```

(* Definitionen der Informations-Modelle *)

```
ENTITY CIMOSAModelInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;

ENTITY EventInformationModel;
   OccurrenceId                      : UniqueId;
   FunctionPosition                  : Point;
   FunctionGhostPositions            : GhostPositions;
   FunctionInformationPosition       : Point;
   FunctionInformationGhostPositions : GhostPositions;
END_ENTITY;

ENTITY NameInformationModel;
   OccurrenceId                      : UniqueId;
   FunctionPosition                  : Point;
   FunctionGhostPositions            : GhostPositions;
   FunctionTreePosition              : Point;
   FunctionTreeGhostPositions        : GhostPositions;
   InformationPosition               : Point;
   InformationGhostPositions         : GhostPositions;
   FunctionInformationPosition       : Point;
   FunctionInformationGhostPositions : GhostPositions;
   NameStr                           : STRING;
END_ENTITY;

ENTITY DomainProcessInformationModel;
   OccurrenceId                      : UniqueId;
   FunctionPosition                  : Point;
   FunctionGhostPositions            : GhostPositions;
   FunctionTreePosition              : Point;
   FunctionTreeGhostPositions        : GhostPositions;
   FunctionInformationPosition       : Point;
   FunctionInformationGhostPositions : GhostPositions;
END_ENTITY;

ENTITY EnterpriseActivityInformationModel;
   OccurrenceId                      : UniqueId;
   FunctionPosition                  : Point;
   FunctionGhostPositions            : GhostPositions;
   FunctionTreePosition              : Point;
   FunctionTreeGhostPositions        : GhostPositions;
   FunctionInformationPosition       : Point;
   FunctionInformationGhostPositions : GhostPositions;
END_ENTITY;

ENTITY FunctionalOperationInformationModel;
   OccurrenceId                      : UniqueId;
   FunctionPosition                  : Point;
   FunctionGhostPositions            : GhostPositions;
   FunctionTreePosition              : Point;
   FunctionTreeGhostPositions        : GhostPositions;
   FunctionInformationPosition       : Point;
   FunctionInformationGhostPositions : GhostPositions;
END_ENTITY;
```

```
ENTITY ProceduralRuleInformationModel;
   OccurrenceId                : UniqueId;
   FunctionPosition            : Point;
   FunctionGhostPositions      : GhostPositions;
END_ENTITY;

ENTITY StartInformationModel;
   OccurrenceId                : UniqueId;
   FunctionPosition            : Point;
   FunctionGhostPositions      : GhostPositions;
END_ENTITY;

ENTITY FinishInformationModel;
   OccurrenceId                : UniqueId;
   FunctionPosition            : Point;
   FunctionGhostPositions      : GhostPositions;
END_ENTITY;

ENTITY ObjectViewInformationModel;
   OccurrenceId                          : UniqueId;
   InformationPosition                   : Point;
   InformationGhostPositions             : GhostPositions;
   FunctionInformationPosition           : Point;
   FunctionInformationGhostPositions     : GhostPositions;
END_ENTITY;

ENTITY InformationElementInformationModel;
   OccurrenceId                : UniqueId;
   InformationPosition         : Point;
   InformationGhostPositions   : GhostPositions;
END_ENTITY;

ENTITY SubReferenceInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;

ENTITY ReferenceInformationModel;
   OccurrenceId : UniqueId;
END_ENTITY;

ENTITY ConnectionInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;

ENTITY EventConnectionInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;

ENTITY FIConnectionInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;

ENTITY FOConnectionInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;
```

```
ENTITY CIConnectionInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;

ENTITY COConnectionInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;

ENTITY RIConnectionInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;

ENTITY ROConnectionInformationModel;
   OccurrenceId   : UniqueId;
END_ENTITY;
```

(* Definitionen der dynamischen Baustein-Templates *)

```
ENTITY CIMOSAModel;
   ConnectionModel      : ConnectionModel;
   InformationModel     : CIMOSAModelInformationModel;
END_ENTITY;

ENTITY Event;
   ConnectionModel      : ConnectionModel;
   InformationModel     : EventInformationModel;
END_ENTITY;

ENTITY Name;
   ConnectionModel      : ConnectionModel;
   InformationModel     : NameInformationModel;
END_ENTITY;

ENTITY DomainProcess;
   ConnectionModel      : ConnectionModel;
   InformationModel     : DomainProcessInformationModel;
END_ENTITY;

ENTITY EnterpriseActivity;
   ConnectionModel      : ConnectionModel;
   InformationModel     : EnterpriseActivityInformationModel;
END_ENTITY;

ENTITY FunctionalOperation;
   ConnectionModel      : ConnectionModel;
   InformationModel     : FunctionalOperationInformationModel;
END_ENTITY;

ENTITY ProceduralRule;
   ConnectionModel      : ConnectionModel;
   InformationModel     : ProceduralRuleInformationModel;
END_ENTITY;
```

```
ENTITY Start;
   ConnectionModel      : ConnectionModel;
   InformationModel     : StartInformationModel;
END_ENTITY;

ENTITY Finish;
   ConnectionModel      : ConnectionModel;
   InformationModel     : FinishInformationModel;
END_ENTITY;

ENTITY ObjectView;
   ConnectionModel      : ConnectionModel;
   InformationModel     : ObjectViewInformationModel;
END_ENTITY;

ENTITY InformationElement;
   ConnectionModel      : ConnectionModel;
   InformationModel     : InformationElementInformationModel;
END_ENTITY;

ENTITY SubReference;
   ConnectionModel      : ConnectionModel;
   InformationModel     : SubReferenceInformationModel;
END_ENTITY;

ENTITY Reference;
   ConnectionModel      : ConnectionModel;
   InformationModel     : ReferenceInformationModel;
END_ENTITY;

ENTITY Connection;
   ConnectionModel      : ConnectionModel;
   InformationModel     : ConnectionInformationModel;
END_ENTITY;

ENTITY EventConnection;
   ConnectionModel      : ConnectionModel;
   InformationModel     : EventConnectionInformationModel;
END_ENTITY;

ENTITY FIConnection;
   ConnectionModel      : ConnectionModel;
   InformationModel     : FIConnectionInformationModel;
END_ENTITY;

ENTITY FOConnection;
   ConnectionModel      : ConnectionModel;
   InformationModel     : FOConnectionInformationModel;
END_ENTITY;

ENTITY CIConnection;
   ConnectionModel      : ConnectionModel;
   InformationModel     : CIConnectionInformationModel;
END_ENTITY;
```

```
ENTITY COConnection;
   ConnectionModel      : ConnectionModel;
   InformationModel     : COConnectionInformationModel;
END_ENTITY;

ENTITY RIConnection;
   ConnectionModel      : ConnectionModel;
   InformationModel     : RIConnectionInformationModel;
END_ENTITY;

ENTITY ROConnection;
   ConnectionModel      : ConnectionModel;
   InformationModel     : ROConnectionInformationModel;
END_ENTITY;

END_SCHEMA;
```

ANHANG C: Die vollständige Definition der dynamischen Baustein-Templates für das zweite Testszenario (Petri-Netze)

SCHEMA PetriNet_1_0;

(* Allgemeine Definitionen *)

```
TYPE ValueString = STRING;
END_TYPE;

TYPE UniqueId = STRING;
END_TYPE;

TYPE IdList = LIST OF STRING;
END_TYPE;

TYPE GhostPositions = LIST OF GhostPosition;
END_TYPE;

ENTITY Point;
        x          : REAL;
        y          : REAL;
END_ENTITY;

ENTITY GhostPosition;
   HauntedNode   : STRING;
   Position      : Point;
END_ENTITY;

ENTITY ConnectionModel;
   Inputs        : IdList;
   Outputs       : IdList;
   SubRefs       : IdList;
   SuperRefs     : IdList;
   ToRefs        : IdList;
   FromRefs      : IdList;
END_ENTITY;
```

(* Definitionen der Informations-Modelle *)

```
ENTITY Icon;
   Name              : STRING;
   DisplayElements   : DisplayElementList;   (* Die Definition dieses Typs wird von der Definition *)
END_ENTITY;                                  (* der statischen Baustein-Templates übernommen    *)

TYPE IconList = LIST OF Icon;
END_TYPE;

ENTITY PetriNetComponentInformationModel;
   OccurrenceId  : UniqueId;
   StopFlag      : BOOLEAN;
   IconList      : IconList;
END_ENTITY;

ENTITY TransitionInformationModel;
   OccurrenceId              : UniqueId;
   StandardPosition          : Point;
   StandardGhostPositions    : GhostPositions;
   FireFlag                  : BOOLEAN;
   IconName                  : STRING;
END_ENTITY;

ENTITY PlaceInformationModel;
   OccurrenceId              : UniqueId;
   StandardPosition          : Point;
   StandardGhostPositions    : GhostPositions;
   IconName                  : STRING;
END_ENTITY;

ENTITY ModuleInformationModel;
   OccurrenceId              : UniqueId;
   StandardPosition          : Point;
   StandardGhostPositions    : GhostPositions;
   TreePosition              : Point;
   TreeGhostPositions        : GhostPositions;
   IconName                  : STRING;
END_ENTITY;

ENTITY TokenAttribute;
   Name          : STRING;
   Value         : ValueString;
END_ENTITY;

TYPE TokenAttributeList = LIST OF TokenAttribute;
END_TYPE;

ENTITY TokenInformationModel;
   OccurrenceId              : UniqueId;
   StandardPosition          : Point;
   StandardGhostPositions    : GhostPositions;
   Attributes                : TokenAttributeList;
END_ENTITY;
```

```
ENTITY ActionCodeInformationModel;
   OccurrenceId              : UniqueId;
   StandardPosition          : Point;
   StandardGhostPositions    : GhostPositions;
   CodeStr                   : STRING;
END_ENTITY;

ENTITY PoolInformationModel;
   OccurrenceId              : UniqueId;
   StandardPosition          : Point;
   StandardGhostPositions    : GhostPositions;
   IconName                  : STRING;
END_ENTITY;

ENTITY NameInformationModel;
   OccurrenceId              : UniqueId;
   StandardPosition          : Point;
   StandardGhostPositions    : GhostPositions;
   TreePosition              : Point;
   TreeGhostPositions        : GhostPositions;
   NameStr : STRING;
END_ENTITY;

ENTITY SubReferenceInformationModel;
   OccurrenceId : UniqueId;
END_ENTITY;

ENTITY ReferenceInformationModel;
   OccurrenceId : UniqueId;
END_ENTITY;

ENTITY ArcAttribute;
   Name         : STRING;
   Value        : OPTIONAL ValueString;
END_ENTITY;

TYPE ArcAttributeList = LIST OF ArcAttribute;
END_TYPE;

ENTITY ArcInformationModel;
   OccurrenceId : UniqueId;
   Attributes   : ArcAttributeList;
END_ENTITY;
```

(* Definitionen der dynamischen Baustein-Templates *)

```
ENTITY PetriNetComponent;
   ConnectionModel     : ConnectionModel;
   InformationModel    : PetriNetComponentInformationModel;
END_ENTITY;

ENTITY Transition;
   ConnectionModel     : ConnectionModel;
   InformationModel    : TransitionInformationModel;
END_ENTITY;
```

```
ENTITY Place;
   ConnectionModel      : ConnectionModel;
   InformationModel     : PlaceInformationModel;
END_ENTITY;

ENTITY Module;
   ConnectionModel      : ConnectionModel;
   InformationModel     : ModuleInformationModel;
END_ENTITY;

ENTITY Token;
   ConnectionModel      : ConnectionModel;
   InformationModel     : TokenInformationModel;
END_ENTITY;

ENTITY ActionCode;
   ConnectionModel      : ConnectionModel;
   InformationModel     : ActionCodeInformationModel;
END_ENTITY;

ENTITY Pool;
   ConnectionModel      : ConnectionModel;
   InformationModel     : PoolInformationModel;
END_ENTITY;

ENTITY Name;
   ConnectionModel      : ConnectionModel;
   InformationModel     : NameInformationModel;
END_ENTITY;

ENTITY SubReference;
   ConnectionModel      : ConnectionModel;
   InformationModel     : SubReferenceInformationModel;
END_ENTITY;

ENTITY Reference;
   ConnectionModel      : ConnectionModel;
   InformationModel     : ReferenceInformationModel;
END_ENTITY;

ENTITY Arc;
   ConnectionModel      : ConnectionModel;
   InformationModel     : ArcInformationModel;
END_ENTITY;

END_SCHEMA;
```