Das ASCEND-Modell zur Unterstützung kooperativer Prozesse

Von der Fakultät Informatik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) eingereichte Abhandlung

Vorgelegt durch Aiko Frank aus Memmingen

Hauptberichter: Prof. Dr.-Ing. habil. B. Mitschang

Mitberichter: Prof. Dr.-Ing. D. Roller

Tag der mündlichen Prüfung: 2.8.2002

Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR)
der Universität Stuttgart

2002

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter für Prof. Dr. Bernhard Mitschang an der Technischen Universität München und später der Universität Stuttgart. Er ermöglichte mir eine weitgehend selbstständige Arbeit und Herangehensweise an dieses Thema und unterstützte mich in vielen Bereichen. Weiterer Dank gilt auch Herrn Prof. Dr. Roller, welcher die Zweitbegutachtung übernahm, und Herrn Prof. Dr. Claus, der die Prüfungskommission leitete.

Die Arbeit wurde unter Leitung von Prof. Dr. Mitschang am Lehrstuhl von Herrn Prof. Rudolf Bayer in München begonnen. In diesem Rahmen möchte ich mich herzlich bei den ehemaligen Kollegen und Kollegin Clara, Stephan, Giannis, Michael, Markus, Martin, Volker, Norbert, Carsten und allen anderen bedanken, da durch deren Unterstützung - auf die eine oder andere Weise - meine Arbeit entsprechend entstehen konnte. Ebenso freundlich wurde ich an der Universität Stuttgart aufgenommen und weiter bei meiner Arbeit unterstützt, wobei auch die Nachbarabteilung für Verteilte Systeme unter Leitung von Prof. Dr. Rothermel einen wesentlichen Anteil hatte.

Ganz besonders wertvoll war die Zusammenarbeit mit Dr. Jürgen Sellentin, Kerstin Schneider und Dr. Christian Becker, die dieser Arbeit entscheidende Impulse gaben, nicht nur in wissenschaftlicher Hinsicht.

Prof. Dr. Theo Härder und seinen wissenschaftlichen Mitarbeiter(inne)n sei ein besonderer Dank für die konstruktiven Gespräche und die erfreulichen Workshops (meist in schöner Umgebung). Herrn Prof. Dr. Norber Ritter gilt in diesem Rahmen als mein Vorgänger bei der Untersuchung kooperativer Designflows mein spezieller Dank, da er sich auch die Zeit nahm, wichtige Aspekte mit mir zu diskutieren. Herrn Prof. Dr. Jablonski und Herrn Dr. Bußler möchte ich dafür danken, daß ich von ihnen gelernt habe, wie exakt Forschung sein sollte und mir dies bis zum Abschluß dieser Arbeit half.

Und nicht zuletzt sei der Einsatz der Studenten und Studentinnen erwähnt, der die Umsetzung der vorliegenden Arbeit ermöglichte.

Für meine Eltern,

München im Dezember 2002, Aiko Frank

Abstract

The *ASCEND Designflow Model* (ADM) supports the execution of cooperative processes, which we refer to as designflows. The main focus of this model is the combination of activity support and sharing of common data. A prototype implementation of this model was achieved by skillfully integrating technologies from groupware, workflow management, and interaction protocols.

Processes in concurrent engineering and design rely on cooperation between the different participants. This cooperation is necessary to establish and coordinate the tasks needed to fulfill the goal of the design process. Those tasks (also called design steps) - in which the complete process can be subdivided into - rely on the results, states, and progress of concurrent, previous, and future tasks. Typically, it can not be exactly planned beforehand which tasks are necessary, how the tasks depend on each other, and how and when those tasks have to interact. Thus, it is necessary to offer mechanisms to support cooperation in a dynamic fashion and at runtime. This new class of cooperative processes is described and shown by example scenarios. They have have the need for interaction, cooperation, cooperative resource sharing, delegation of activities, structured and less-structured process parts, integration of results from delegated activities, and the coordination of actions in common.

From this we derive the requirements of an adequate user assistance, which does not prescribe a single process model but gives the users the freedom to choose from the most suitable service and adapt it to their needs. To find the suitable assistance, different technologies are examined: CSCW, workflow management, CAD frameworks, agent technologies, software engineering process models, and process net planning.

Aspects of the examined technologies are combined in the ASCEND Designflow Model. Workflow management aspects are used to offer a structured activity model and automate processes. Constraints are used to allow one to model certain process specific constraints in activities, thus allowing for more flexibility in processes. Primitive activities encapsulate the so called design steps, which encapsulate unstructured work like CAD tool applications or groupware activities.

All activities and constraints have access to a common information space. Thus, allowing for system supported common data access and exchange. A high level of cooperation on those data is achieved. In order to enable this (interactive) cooperation, protocol based conversation patterns are established. The adaptability of the conversations also realizes customization to a great variety of cooperative processes. Since the protocols and the data sharing are both supported by ASCEND, the consistent data handling can be ensured within the system.

As a conclusion this approach promotes cooperative techniques like awareness and negotiation through the activity model an the shared information space combined with the protocols. This allows cooperation to take place between the actors and entities in the ASCEND Designflow Model.

Kurzfassung

In dieser Arbeit wird das *ASCEND Designflow Model* (ADM) zur Unterstützung kooperativer Prozesse, die wir als *Entwurfprozesse* bezeichnen, vorgestellt. Der Kern dieses Modells ist die Fokussierung auf die kooperative Nutzung gemeinsamer Daten in Verbindung mit einem Aktivitätenmodell. Die Konzeption wird durch die geschickte Integration von Technologien aus den Bereichen Groupware, Workflow-Management und Interaktionsprotokollen gekennzeichnet und wurde prototypisch realisiert.

Es wird eine neue Klasse von kooperativen Prozessen bestimmt und durch Beispiele betrachtet, deren Unterstützung durch das ADM erfolgen soll. Diesen Prozessen ist der Bedarf nach Interaktion, Kooperation, kooperativer Nutzung gemeinsamer Ressourcen, Delegation von Teilarbeiten, strukturierten und weniger strukturierten Teilprozessen, Integration von Arbeitsergebnissen und Abstimmung von Aktionen gemein. Daraus wird die Forderung an eine geeignete Benutzerunterstützung abgeleitet, die anstatt einer festen Vorgehensbeschreibung den Nutzern die geeignete Unterstützung in Form entsprechend konfigurierbarer Dienste zur Verfügung stellt. Dies schließt auch, wo es geeignet ist, die Durchführung wohl-strukturierter Teilprozesse mit ein.

Es werden Technologien vorgestellt und bewertet, die Teile der aufgestellten Forderungen erfüllen können. Der Schwerpunkt dieser Untersuchung betrifft *CSCW* und *Workflow-Management*, deren explizites Ziel die allgemeine Unterstützung von Arbeit ist. Eine weitere Klasse von Systemen zur Durchführung von Arbeiten sind *CAD-Frameworks*, die spezialisierte Dienste für den technischen Entwurf anbieten. Moderne Systeme umfassen die Prozeß-Steuerung und zentrale Datenversorgung. Dies wird auch unter dem Begriff *Designflow-Management* zusammengefaßt. Für die Realisierung der von uns gewünschten flexiblen Zugriffsregelung werden außerdem einige Aspekte der *Agententechnologie* betrachtet, insbesondere Verhandlungsprotokolle. Als weitere interessante Technologien werden ein Prozeßmodell für das Software-Engineering, der TOGA-Ansatz und die Prozeßplanung mittels einer erweiterten Netzplantechnik untersucht. Aufgrund der so gewonnenen Erkenntnisse wird ein Lösungsansatz präsentiert, der auf einer geeigneten Integration dieser Technologien basiert.

Dieser Lösungsansatz wird durch das ASCEND Designflow Model umgesetzt. Dieses Modell verwendet drei wesentliche Aspekte: ein Aktivitätenmodell, einen Informationsraum und Interaktionsprotokolle. Workflow-Management stellt eine ideale Technologie für die Automatisierung der Steuerung von strukturierten Teilprozessen dar. Das Aktivitätenkonzept ist eine geeignete Basis zur Repräsentation von abhängigen Arbeitsschritten. Daher werden diese Konzepte weitgehend in das ADM integriert. Das Aktivitätenkonzept zur Modellierung und Durchführung abgegrenzter Arbeitsschritte hilft die Aufgabenverteilung und Vorgehensweise von Entwurfsprozessen, soweit möglich, zu strukturieren. Bspw. nutzt die Delegations-Beziehung des ADM Aktivitäten zur Spezifikation verschiedener Unteraufträge. Außerdem werden sogenannte Workflow-Aktivitäten eingeführt, die alle Eigenschaften eines Workflows übernehmen und innerhalb eines Entwurfsprozesses ausgeführt werden können. Dadurch wird eine geeigne-

te Unterstützung gut strukturierter Teilprozesse erreicht. Weiterhin werden primitive Aktivitäten zum Kapseln von Werkzeuganwendungen und Groupware-Aktivitäten zur Durchführung von wenig strukturierten Teilarbeiten eingeführt. Eine Besonderheit stellen die Designflow-Aktivitäten dar, die durch sogenannte *Design-Primitive* eine erweiterte Funktionalität realisieren. So können anpaßbare Constraints angewendet werden, welche die Abhängigkeiten zwischen den in einer Designflow-Aktivität enthaltenen Ressourcen und Aktivitäten beschreiben. Durch die weitgehende Definierbarkeit solcher Constraints, besteht die Möglichkeit anwendungsspezifische Abhängigkeiten einzuführen und eine flexible Ablaufunterstützung zu erreichen.

Aufgrund der Forderung nach frühem Austausch von gemeinsamen Ergebnissen, der Bearbeitung gemeinsamer Daten und der Abhängigkeiten bezüglich Daten und Ergebnissen, die in verschiedenen Teilprozessen erarbeitet werden, ist eine Abstimmung zwischen den am Prozeß teilnehmenden Personen notwendig. Dafür wird die gemeinsame Nutzung von Ressourcen im Rahmen eines gemeinsamen Informationsraums eingeführt. Dadurch können unvorherbestimmte Abläufe über die Objektzugriffe koordiniert werden.

Zur Durchführung und Abstimmung der Nutzung gemeinsamer Objekte werden Protokolle in Konversationsmustern angewendet, die zum einen eine gewisse Weise des Zugriffs vorschreiben, aber auch die Möglichkeit zur Verhandlung anbieten. Diese Verhandlung, wie sie bei konkurrierenden Zugriffen oder bei der Durchführung des sogenannten Delegationsprotokolls auftreten, stellen ein mächtiges Werkzeug zur Interaktion zwischen allen Entitäten des ADM dar, d.h. zwischen Akteuren, Objekten und Aktivitäten. Die Effekte der Interaktionen werden komplett durch das zugrunde liegende System unterstützt, womit eine konsistente Behandlung ermöglicht ist. Durch die Einbindung der Akteure wird eine direkte Kooperation erzielt. Die flexible Einsetzbarkeit, die Anpaßbarkeit und die Erweiterbarkeit der Protokolle ermöglicht einen hohen Grad der Anpassung des ADM an verschiedenste kooperative Prozesse.

Damit unterstützt das ADM zum einen Entwurfsprozesse, die teilweise gut strukturiert sind. So können bei Teilprozessen Workflows eingesetzt werden. Zum anderen erlauben die eingeführten Entwurfskonstrukte (bspw. Delegation, Objektzugriffe und Constraints), auch schwächer strukturierte Teilprozesse und damit ein wesentliches Merkmal des Entwurfs bzw. der in dieser Arbeit anvisierten kooperativen Prozesse zu unterstützen. Genauso ist es möglich, Groupware gewinnbringend für die Teile eines Prozesses einzusetzen, die keine inhärente Struktur mehr besitzen, wobei es zudem möglich ist, über den gemeinsamen Informationsraum zu kooperieren. Somit wird erreicht, daß die passendste, unterstützende Technologie für den jeweiligen Teilprozeß verwendet werden kann. Dadurch werden die verschiedenen Anforderungen bezüglich koordinativer, wie auch kooperativer Zusammenarbeit erfüllt.

1 Einleitung

Die vorliegende Arbeit untersucht eine neu bestimmte Prozeßklasse. Diese Prozesse sind durch ihren kooperativen Charakter gekennzeichnet und spannen sich von klassischen Entwurfsprozessen bis hin zu Planungsprozessen. Zur systemgestützten Durchführung dieser Prozesse wurde ein generisches Modell entwickelt und prototypisch umgesetzt.

Dieses Kapitel gibt eine Einführung in die zu unterstützenden Arbeitsabläufe und welche Aufgabenstellung damit verbunden ist. Zuletzt wird der Aufbau der vorliegenden Arbeit beschrieben.

1.1 Beschreibung des Anwendungsgebietes

Die Unterstützung der Zusammenarbeit verschiedener Teammitglieder an einem gemeinsamen Ziel wird im wesentlichen im Rahmen der Forschungsrichtung der computergestützten Gruppenarbeit [BORGHOFF UND SCHLICHTER 1995] untersucht. In der vorliegenden Arbeit wird hierbei besonders eine enge kooperative Zusammenarbeit betrachtet, bei der die einzelnen Tätigkeiten voneinander abhängig sind und gleichzeitig durchgeführt werden. Dies hat im allgemeinen zur Folge, daß ohne Kommunikation und/oder entsprechende Koordinierungsmaßnahmen leicht Konflikte und Mißverständnisse auftreten können. Wir motivieren dies zunächst anhand von Entwurfsprozessen.

Ein Entwurf von Produkten findet in fast allen Bereichen der Industrie statt. Dabei kann es sich um eine Dienstleistung handeln aber auch um etwas faßbares, wie z.B. ein Auto. Der Entwurf ist von Kreativität geprägt, welche die Schaffung eines Produkts oder die Erbringung einer Dienstleistung erst ermöglicht. Eine Organisation will einen Entwurfsprozeß beherrschbar machen, um Merkmale wie zeitliche Dauer, Qualität oder Kosten überwachen zu können. Prozesse in der Verwaltung von Unternehmen haben auch diese Anforderung bezüglich der Koordination und können bspw. geeignet mittels Workflow-Management durchgeführt werden. Damit auch Entwurfsprozesse eine gesamtheitliche Systemunterstützung erfahren, etwa in den Bereichen der Datenbehandlung und Kooperation, sind zusätzliche Technologien erforderlich.

Die Komplexität des Chip-Entwurfs erfordert die Partitionierung der Aufgabe in Teilaufgaben, die von verschiedenen Personen zu bearbeiten sind [ANTREICH 2000]. Die dadurch entstehenden Teilentwürfe sind zu integrieren und sollen letztendlich einen Entwurf für einen funktionsfähigen Chip ergeben. Jedoch kann man nicht ohne eine entsprechende Abstimmung davon ausgehen, daß die Integration der Teilergebnisse problemlos ist. Vielmehr ist zu erwarten, daß der Gesamtentwurf nicht den gewünschten Anforderungen genügt oder sogar fehlerhaft ist. Konflikte entstehen beispielsweise durch fehlende und fehlerhafte Spezifikationen für die Teilentwürfe und insbesondere durch mangelnde Abstimmung zwischen den gleichzeitig arbeitenden Ingenieuren [SELLENTIN ET AL. 1999]. Je später Diskrepanzen erkannt werden, desto teurer wird auch der ganze Entwicklungsprozeß. Daher wird hier ein Modell vorgestellt, das geeignete Mechanismen zur Verbesserung dieser Situation einsetzt, was sich insbesonders auf die Nutzung gemeinsamer Ressourcen und die gegenseitige Abstimmung bezieht. Die Motivation für dieses Modell ist aus den Anforderungen des *Concurrent Engineering* und CAD-Entwurfs entstanden, wie auch an dem Beispiel des Chip-Entwurfs gezeigt. Im Verlauf dieser Arbeit wird gezeigt, daß viele der dort auftretenden Probleme auch in anderen Arbeitsbereichen zu finden sind. So können beispielsweise auch Planungsprozesse von den hier vorgestellten Lösungen profitieren.

Ein weiterer wichtiger Aspekt in der Bereitstellung einer umfassenden Systemunterstützung solcher Prozesse ist die Integration heterogener Systeme und Werkzeuge. Zum einen ist es dringend notwendig spezialisierte Werkzeuge, wie etwa eine Simulations-Software für Schaltungen, einzubinden. Zum anderen bietet sich die Verwendung existierender und bezüglich ihrer Funktionalität bereits ausgereifter Systeme, wie etwa Datenbank- und Workflow-Management-Systeme, an. Dies sind auch die typischen Herausforderungen an eine Integration, die durch heterogene Systeme entstehen.

Derzeit besteht ein Defizit an umfassender Systemunterstützung für die zuvor genannten Arbeitsbereiche. In dieser Arbeit werden Mechanismen und Technologien untersucht, welche insbesondere die systemumfassende Kooperation in Verbindung mit einem entsprechendem Aktivitätenmodell betreffen.

1.2 Ziele

Arbeitsvorgänge wie Entwurf, Concurrent Engineering oder auch Planung können durch ihren Teamcharakter und inhärente Kreativität gekennzeichnet werden. Im allgemeinen sind solche Vorgänge nicht gänzlich im voraus planbar, da sie durch immer neu auftretende Probleme und Fragestellungen dauernder Anpassung unterworfen sind. Daher kann deren Ablauf nicht vollkommen automatisiert werden. Es existieren aber automatisierbare Teilvorgänge, die wiederkehrend auftreten, wie zum Beispiel das Testen der Logik eines Schaltungsentwurfs. Um diese Prozesse in ihrer Gesamtheit rechnergestützt durchzuführen, wird ein entsprechend flexibles Ablaufmodell benötigt, das sowohl automatisierbare und als auch weniger automatisierbare Teilvorgänge unterstützt. Zudem besteht ein hoher Bedarf an Kooperation, da die Arbeitsschritte abgestimmt werden müssen und die einzelnen Teilnehmer solcher Vorgänge auf die Ergebnisse und Arbeiten der anderen in hohem Maße angewiesen sind. Es werden damit auch Mechanismen zur Interaktion und Kooperation benötigt. Wir betrachten die Konsistenzwahrung der Daten als wichtiges Merkmal und fordern daher auch einen systemgestützten Zugriff der Daten durch die Kommunkationsprotokolle. Aufgrund dieser Eigenschaften muß ein umfassendes Systemmodell Ablaufunterstützung, gemeinsame Datenutzung und Kooperationsunterstützung umfassen.

In der vorliegenden Arbeit wird ein Modell für die Unterstützung der von uns betrachteten Prozeßklasse entwickelt. Um die beschriebenen Anforderungen zu erfüllen, liegt es nahe Techniken aus den Bereichen Workflow-Management, Groupware und Datenverwaltung zu kombinieren. Wir werden die Anforderungen aufgrund einer Untersuchung der zu unterstützenden Arbeitsvorgänge im folgenden Kapitel verfeinern.

1.3 Einordnung dieser Arbeit

Die vorliegende Arbeit präsentiert die erweiterten Ergebnisse, die ihren Ursprung im ASCEND-Projekt (Activity Support in Co-operative ENvironments for Design issues[1]) hatten. Dessen Ziel war es eine umfassende Unterstützung von Entwurfsprozessen zu erreichen, unter besonderer Berücksichtigung der kooperativen Aspekte im Entwurf.

Die bisherige Unterstützung im Bereich des Concurrent Engineering und gemeinsamen Entwurfs kennt nur wenige Ansätze für eine umfassende Systemunterstützung. Einige positive Ansätze sind im Bereich von CAD-Systemen zu finden, wo CAD-Frameworks um entsprechende Funktionalitäten erweitert wurden. Integrierte Systeme, wie Gruppeneditoren, erlauben einen

1. Dieses Projekt wurde zum Teil von der DFG unter dem Geschäftszeichen MI 311/8 “Entwurfsablaufsteuerung” finanziert.

eng gekoppelten gemeinsamen Entwurf. Der Einsatz von Workflow-Management-Systemen ist durchaus auch üblich, wobei die vorhandenen CAD-Werkzeuge in koordinierte Arbeitsschritte eingebunden werden. Diese Ansätze berücksichtigen jeweils lediglich einen Teil der in dieser Arbeit gestellten Anforderungen. Das hier vorgestellte Modell bietet einen integrierenden Ansatz, der

- die einzelnen Arbeitsschritte durch verschiedene Koordinations- und Kommunikationsmechanismen unterstützt, jedoch nicht “nur” regelt im Sinne der Ablaufsteuerung,
- die notwendigen Werkzeuge und Systeme einbindet,
- und kooperatives Arbeiten ermöglicht.

Letztendlich ist dieses Modell aufgrund des Kooperationsaspekts dem Forschungsbereich CSCW (*computer supported cooperative work*) zuzuordnen.

1.4 Leitfaden zum Lesen

Der schematische Aufbau dieser Arbeit ist in Abbildung 1 illustriert. Anhand der dargestellten Themen und deren Abhängigkeiten, kann der Leser sich einen Überblick über die bearbeiteten Themen schaffen und die für ihn interessanten Wege verfolgen.

In Kapitel 2 beschreiben wir die zu unterstützenden kooperativen Prozesse um die Schwerpunkte der Aufgabenstellung zu verdeutlichen. Dazu besprechen wir verschiedene Arbeitsformen und leiten Eigenschaften ab, die zu deren Kategorisierung genutzt werden. Diese Kategorien ermöglichen es, verschiedenen Klassen die zu unterstützenden Technologien zuzuordnen. Um diese Klassen anzuwenden, besprechen wir drei unterschiedliche Szenarien für kooperative Prozesse: Chip-Entwurf, Software-Entwicklung und Filialplanung. Aufgrund der zuvor ermittelten Kategorien ordnen wir diese neu bestimmten Klasse der kooperativen Prozesse zu. Wir deuten schließlich anhand der Folgerungen für die Aufgabenstellung an, daß eine Kombination von Workflow- und Groupware-Technologie die Lösung der eingangs motivierten Fragestellung ist.

Basierend auf der nun fixierten Aufgabenstellung und eines ersten Lösungsansatz werden in Kapitel 3 die dafür in Frage kommenden Technologien diskutiert, d.h. Groupware, Workflow-Management, Agententechnologie und CAD-Frameworks. Wir bewerten anschließend deren Einsetzbarkeit zur Lösung unserer Aufgabenstellung und lassen dies in unseren Lösungsansatz einfließen.

In Kapitel 4 wird das ASCEND Designflow Model detailliert vorgestellt, das den in den vorherigen Kapiteln motivierten Anforderungen genügt. Dieses besteht aus den drei Bereichen: Aktivitätenmodell, Informationsraum und Interaktionsmodell. Das Aktivitätenmodell basiert auf Konzepten des Workflow-Managements und wird für die zu unterstützenden kooperativen Prozesse erweitert. Die Erweiterungen sind spezielle Protokolle für den Ablauf, wie z.B. die Delegation und Zugriffs- und Verhandlungsprotokolle für die gemeinsame Datennutzung. Der Informationsraum stellt die Basis für die gemeinsame Nutzung für Daten dar und ist damit ein typisches Groupware-Konzept. Das agentenbasierte Interaktionskonzept erlaubt die Definition und flexible Verwendung von Protokollen zur Kommunikation. Dadurch werden Kooperationsverhandlungen und Informationsraumverwendung realisiert. Wir wenden dieses Modell auf das in Kapitel 2 beschriebene Planungsszenario zur Verdeutlichung des ASCEND Designflow Model an.

Ausgehend von diesem Modell wurde der CASSY-Prototyp (*Cooperative Activity Support System*) entwickelt, den wir in Kapitel 5 vorstellen. Als Grundlage für dieses Kapitel führen wir den Systementwurf ein und besprechen einige wichtige Aspekte zur Implementierung integra-

tiver Systeme. Daraufhin werden die einzelnen Komponenten des Prototyps vorgestellt: Event Engine, Workflow Facility, Groupware Facility und die Protokollmaschine. Zur Verwendung dieser Komponenten bestehen entsprechende Benutzungsschnittstellen, die ebenso vorgestellt werden. Am Ende des Kapitels folgen eine Übersicht, eine Ausführungsbeispiel und schließlich ein Fazit.

Kapitel 6 beinhaltet eine Zusammenfassung der behandelten Inhalte dieser Arbeit und deren Abgrenzung zu verwandten Themengebieten. Das Kapitel schließt mit einem Ausblick auf weiterführende Arbeiten.

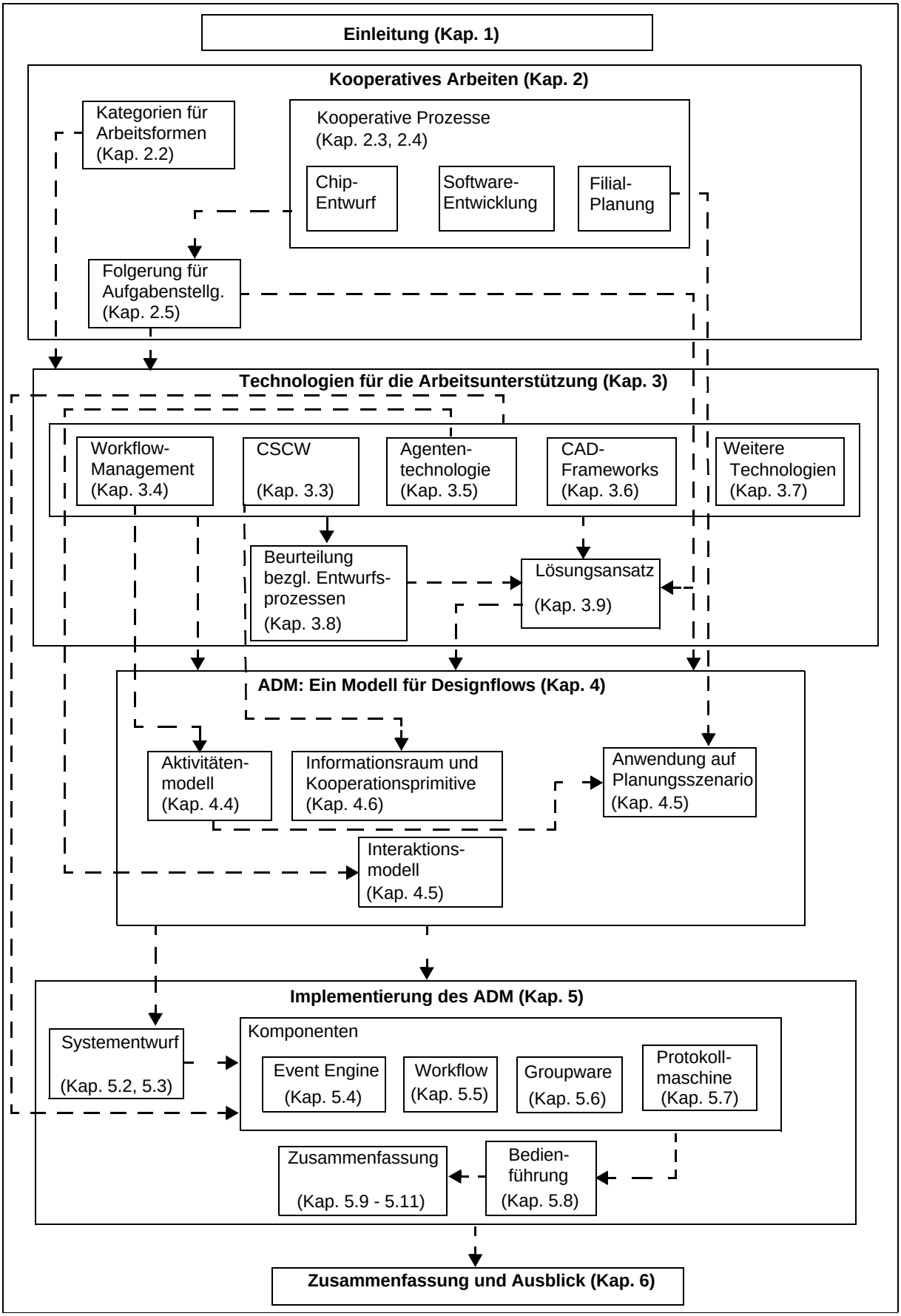


Abbildung 1: Leitfaden zum Lesen der Arbeit

2 Kooperatives Arbeiten

In diesem Kapitel werden verschiedene Arbeitsformen charakterisiert. Diese ordnen wir den zu unterstützenden Prozessen zu, um schließlich die Aufgabenstellung zu fixieren.

2.1 Überblick

Wie bereits in der Einleitung angesprochen, betrachten wir die Zusammenarbeit mehrerer Personen, die ein gemeinsames Ziel haben. Einen solchen kooperativen Arbeitsvorgang haben wir am Beispiel des IC-Entwurfs motiviert. Wie wir sehen werden, bedürfen andere kooperative Prozesse einer weitgehend gleichen Systemunterstützung, wie es das in Abschnitt 2.3.3 vorgestellte Planungsszenario belegt.

In Abschnitt 2.2 besprechen wir zuerst welche Aspekte verschiedene Arbeitsformen besitzen, um kooperative Arbeitsprozesse von anderen abzugrenzen. In Abschnitt 2.3 werden drei Szenarien besprochen, auf die das in dieser Arbeit vorgestellte Modell anwendbar sein soll. Aufgrund dieser Szenarien werden die gesammelten Anforderungen und Ergebnisse in Abschnitt 2.4 zusammengefaßt und schließlich im Rahmen der Aufgabenstellung für die vorliegende Arbeit präsentiert (Abschnitt 2.5).

2.2 Charakterisierung von Arbeitsformen

In diesem Abschnitt werden verschiedene Einordnungsmöglichkeiten für Arbeitsformen entwickelt und beispielhaft dargestellt. Den daraus resultierenden Aspekten ordnen wir unsere Problemstellung zu.

2.2.1 Grundlagen

Arbeit nennen wir einen zielgerichteten Vorgang, der zur Erreichung eines bestimmten Ziels führt. Dies kann der Entwurf eines Produkts, das Erstellen eines Dokuments oder der Verkauf eines Softwarepakets sein. Wie man bei der Nennung der Arbeitsbeispiele sieht, werden diese durch ihr Ziel beschrieben. Wir betrachten vor allem Ziele, die für Organisationen (Firmen, Vereine, Staat, Gemeinschaften, ...) bedeutend sind, in dem Sinne, daß sie zu deren eigentlichen Aufgaben gehören. Mit *Arbeitsvorgang* und *Arbeitsprozeß* bezeichnen wir den gesamten Ablauf zur Erreichung eines solchen Ziels. Es wird dabei nicht ausgeschlossen, daß das Ziel während des Ablaufs eines Arbeitsvorganges geändert bzw. angepaßt werden kann. Ein Arbeitsvorgang kann in Unterziele gegliedert werden, deren Erledigung für das Gesamtziel notwendig ist. Der Ablauf selbst kann somit in einzelne *Arbeitsschritte* aufgeteilt werden.

2.2.2 Arbeitsformen

Arbeitsvorgänge lassen sich verschiedenen Kategorien und Spektren zuordnen. Die nachfolgende Kategorisierung basiert auch auf verschiedenen Ansätzen (z.B. [TEUFEL ET AL. 1995], [GEORGAKOPOULOS ET AL. 1995], [ALONSO ET AL. 1997], [LEYMANN 1997A]) und wurde für die vorliegende Arbeit entsprechend angepaßt und erweitert. Wir wollen gleich zu Beginn einige Beispiele vorstellen und auf deren Unterschiede hinweisen. Im Rahmen der hier vorliegenden Arbeit, werden wir jedoch nicht alle möglichen Konzepte und Kategorien besprechen können, wir werden uns vielmehr auf die für die vorliegende Arbeit interessanten beschränken. Diese Kategorien schließen sich nicht gegenseitig aus. Wichtig ist festzustellen, für welche dieser Kategorien sich welche Mechanismen zu ihrer Unterstützung anbieten:

- **Wiederkehrende, gleichbleibende Arbeiten**
  Hiermit sind Vorgänge gemeint, die immer gleiche Arbeitsschritte in einer festgelegten Reihenfolge verlangen. Deswegen bieten sich vordefinierte Abläufe zur systemseitigen Unterstützung an, da diese nicht oder nur geringfügig zu ändern sind. Dadurch kann eine weitgehende Automatisierung der Ablaufsteuerung erfolgen. Dies gilt aber nur, wenn deren Abläufe beschreibbar sind. Wir nennen beschreibbare Abläufe, die in einer weitgehend formalen Weise ausgedrückt werden können, *definierbar* (z.B. durch eine Workflow-Spezifikation, siehe Abschnitt 3.4) und damit auch strukturierbar (s.u.). Bei nur schlecht definierbaren Abläufen ist natürlich in Betracht zu ziehen, die Arbeit nicht bezüglich des Ablaufs zu definieren, sondern anhand anderer Merkmale der Arbeit (bspw. ist die Aufgabe bekannt, ggf. Betriebsmittel und Hilfsmittel, wie Software, oder auch Bearbeiter bzw. deren Qualifikation).
- **Wiederkehrende aber leicht unterschiedliche Arbeiten**
  Solche Arbeiten unterscheiden sich nur geringfügig bezüglich der Durchführung und treten häufig auf. Eine Automatisierung ist in der Regel weitgehend möglich, jedoch sind Techniken zur Unterstützung der Unterschiede im Prozeß erforderlich. Dies zu erreichen verspricht man sich von sogenannten Adaptive- bzw. Ad-Hoc-Workflow-Systemen. Eine Möglichkeit besteht bzgl. der Unterstützung von nicht vollkommen definierbaren Abläufen, in dem man die leicht unterschiedliche Menge von Arbeitsschritten in einem (grobgranularen) Arbeitsschritt zusammenfaßt bzw. ungenauer beschreibt und somit eine Variabilität in deren Ausführung ermöglicht.
- **Ad hoc auftretende Arbeiten bzw. Aufgaben**
  Es gibt Ad-Hoc-Arbeiten, die wiederkehrend sind. Sind deren Abläufe zur Erledigung wohlbeschreibbar, so bietet sich hier auch die Nutzung bzw. Bereitstellung vordefinierter Abläufe an. Zur Unterstützung ist es nun interessant sich zu fragen, welcher systemseitige Aufwand gerechtfertigt ist, diese Definition auszuführen. Verschlingt eine Automatisierung, wie oben vorgeschlagen, zu viel Ressourcen, insbesondere Zeit, so muß man ggf. auf andere Unterstützungsmechanismen zurückgreifen.
  Für ad hoc auftretende Arbeiten ist es aber i.a. natürlich nicht leicht vordefinierte Abläufe vorzufinden, da wir annehmen, daß diese i.d.R. neuartige bzw. abgeänderte Abläufe benötigen. Somit ist der genaue Ablauf selbst nicht definierbar und daher nicht leicht automatisierbar. Es fehlt also an Wissen, wie bezüglich der Vorgehensweise bzw. des Ablaufs eine solche Aufgabe erledigt werden muß. Sie kann damit nicht immer vorab und vollständig definiert werden. Jedoch ist zumindest die Aufgabe und damit ein Ziel bekannt. Wir haben in dem angenommenen Fall also eine Zielorientierung, jedoch nicht das Wissen für deren genaue Durchführung, oder vielleicht scheut man auch einen unverhältnismäßigen (technischen und/oder intellektuellen und/oder zeitlichen und/oder wirtschaftlichen) Aufwand, festzulegen wie diese Aufgabe im einzelnen zu erledigen ist. Daher sind für die Unterstützung solcher Arbeiten häufig keine rein ablauforientierten Ansätze geeignet. Wie bei den wiederkehrenden Arbeiten besteht dann jedoch die Möglichkeit die Arbeit aufgrund anderer Merkmale (Betriebsmittel, gemeinsame Daten, ...) systemseitig zu unterstützen.
- **Strukturierte, semi-strukturierte und unstrukturierte Arbeiten**
  Unter der Struktur einer Arbeit verstehen wir hier, die Möglichkeit die Arbeit anhand verschiedener Merkmale beschreiben zu können. Dies betrifft vor allem den Aufbau dieser Arbeit, d.h. die Abhängigkeiten, Ressourcenbedürfnisse, Teilabschnitte, Organisationsstrukturen, Regeln usw. Häufig kann man strukturierte Arbeit gut in formalen Beschreibungen fassen. Unstrukturierte bieten hingegen weniger Möglichkeit diese umfassend zu beschreiben. Vielmehr nehmen wir an (wie oben geschrieben), daß im schlechtesten Fall lediglich das Ziel bekannt ist. Semi-strukturierte Arbeit liegt offensichtlich zwischen die-

sen beiden Extremen, d.h. sie besitzt Anteile, die gut beschreibbar sind, jedoch nicht vollständig. Manche semi-strukturierte Abläufe können auch durch spätes Modellieren (*late modeling*) unterstützt werden, durch das der Ablauf während der Durchführung verfeinert und fortgeschrieben wird. Im Nachhinein kann man semi- bzw. unstrukturierte Arbeiten in der Regel durchaus beschreiben, jedoch ist dies vorab nicht möglich. Für unstrukturiertere Prozesse bieten sich damit wiederum Systeme an, die nicht ablaufbezogen sind, wie zum Beispiel TeamWave, das die Kollaboration in gemeinsamen Umgebungen unterstützt (siehe Abschnitt 3.3).

- **Selten auftretende Arbeiten**
  Selten auftretende Arbeiten haben den Nachteil, daß sich für sie eine (relativ aufwendige) Strukturierung für eine Automatisierung ggf. nicht rentiert. Ansonsten können diese aber unstrukturiert bis strukturiert sein.

- **Arbeiten mit vagen Zielen**
  Es treten durchaus Arbeiten auf, die nur ein vage definiertes Ziel vorweisen, und damit auch generell eine unklare Ablauf- und Zielvorstellung haben. D.h. es gibt kein festes Ziel vorab, es wird jedoch ein Ergebnis erwartet (z.B. Professor: “Schauen Sie sich ’mal verteilte Systeme an!”). Die Festlegung des Ziels kann während der Durchführung der Arbeit erfolgen. Das Ergebnis kann auch sein, daß die Arbeit aufgrund des Fehlens eines konkreten Ziels aufgegeben wird. Bei diesen vage definierten Arbeiten kann auch kein Ablauf vorgegeben werden, da kein konkretes Ziel bekannt ist.

- **Kooperative Arbeiten**
  Kooperative Arbeiten, wie sie hier definiert werden, zeichnen sich durch die starke Abhängigkeit bei gemeinsamem “Problemlösen” aus. So sind sie häufig durch Ihren Team-Charakter gekennzeichnet. Im Deutschen bezeichnet *Team* eine Gruppe die zusammen, auf kooperative Weise, vorgeht (vgl. auch [TEUFEL ET AL. 1995] und [KATZENBACH UND SMITH 1993]). Dies drückt sich vor allem durch ein gemeinsames Ziel und die besondere Organisationsstruktur aus. So sind Teams i.d.R. nicht auf formale hierarchische Kommunikationswege angewiesen, wie sie typischerweise im administrativen Bereich zu finden sind (die sog. Kommunikationstransparenz [BORGHOFF UND SCHLICHTER 1995]).
  Manchmal werden aber auch alle Arbeiten, die von mehreren Personen erledigt werden, kooperativ genannt. Wir nennen dies hier *kollaborativ*. Kooperation verwenden wir für eine sozial enge Zusammenarbeit, die durch Interaktion geprägt ist. Beispielsweise sind sich Teammitglieder über das gemeinsame Ziel bewußt, im Gegensatz z.B. bei vorgeschriebenen Abläufen in denen eine Person nur eine Arbeit erledigt, die sie nicht notwendigerweise im Gesamtzusammenhang sehen muß. Zusätzlich verstehen wir hier kooperative Arbeit oder vielleicht auch besser Team-Arbeit als i.d.R. interaktiv, d.h. es treten explizite und implizite Kooperationen zwischen den Teammitgliedern auf.

- **Administrative Arbeiten**
  Administrative Arbeiten, die man auch als bürokratisch bezeichnen kann, umfassen Vorgänge in einem Umfeld, das wohlorganisiert und strukturiert ist, wie es typisch für Verwaltungsorganisationen ist. So existieren auch weitgehende Regeln für die Durchführung von Arbeiten, und die Aufgaben und Pflichten von Mitarbeitern sind bekannt (siehe zum Beispiel die Beschreibung der Vorschriften in deutschen Ministerien [PRINZ ET AL. 1998]). Daher sind viele Abläufe in diesem Bereich gut definierbar. Diese frühzeitige Definitionsmöglichkeit bietet eine Vorwegnahme der Kooperation, da die Arbeit in kleine Arbeitsschritte zerlegt werden kann. Die Abhängigkeiten zwischen diesen Arbeitsschritten werden etwa durch weiterzureichende Dokumente, Daten oder Formulare realisiert.

2.2.3 Ermittelte Charakteristika

Aus obiger Diskussion können verschiedene Kriterien für Arbeit extrahiert werden. Uns sind folgende Punkte wichtig für die Einteilung von verschiedenen Arbeiten:

- *Definierbarkeit und Struktur*
  Die Struktur beschreibt den Aufbau der Arbeit. Ist eine Arbeit wohl-strukturiert läßt sie sich ihr Ablauf besser formal beschreiben und somit leichter definieren, wie etwa die Bearbeitung einer Dienstreiseabrechnung. Je weniger sie strukturiert ist, desto schwerer fällt eine frühzeitige Definition und damit eine vollständige formale Erfassung.
- *Regelungsgrad*
  Zur Durchführung von Arbeit sind verschiedene Arbeitsschritte, die durch die Personen erledigt werden, notwendig. Diese einzelnen Teile können verschieden organisiert werden, etwa durch eine feste Regelung des Ablaufs oder aber auch durch willkürliche Ausführung. Dieses Spektrum nennen wir den *Regelungsgrad* oder auch *Koordinationsgrad* des Ablaufs. Die Regelbarkeit eines Vorgangs hängt mit der Strukturierbarkeit zusammen. Formal beschriebene Abläufe bieten sich für eine automatisierte, also geregelte, Durchführung an. Jedoch ist dies nicht zwingend der Fall. Der Regelungsgrad kann auch den Kooperationsgrad beeinflussen, da eine vorgegebene Regelung zur Durchführung die Freiheit für Kooperation einschränken kann.
- *Wiederholbarkeit und Häufigkeit*
  Es ist interessant, ob eine Arbeit in gleicher Weise öfters durchgeführt wird oder ob sie auch in leicht abgewandelten Formen auftritt. Einmalig oder sehr selten durchzuführende gleichartige Arbeiten sind möglicherweise auf andere Art zu unterstützen als häufig auftretende. Leichte Abänderungen erfordern eine flexiblere Unterstützung als starre Abläufe.
- *Bedeutung*
  Arbeiten können für eine Organisation von unterschiedlicher Bedeutung sein. So ist die Herstellung von Produkten, deren Qualität, Quantität und sichergestellte Erzeugung äußerst wichtig. Die Bestellung von Büromaterial ist hingegen weniger wichtig. Entsprechend sind die Anforderungen an die Qualität der Unterstützung zur Durchführung dieser Vorgänge.
- *Kooperationsbedarf*
  An einem Arbeitsvorgang sind häufig mehrere Personen beteiligt. Diese arbeiten an einem gemeinsamen Ziel, auch wenn dies einigen teilnehmenden Personen gar nicht bewußt sein mag. Daher nennen wir das Arbeiten an einem gemeinsamen Ziel i.A. nicht kooperativ sondern kollaborativ. Manche Vorgänge benötigen jedoch Handlungen, wie Interaktion oder Abstimmung zwischen Personen, dies nennen wir kooperativ.

2.2.4 Einordnung von Entwurfsprozessen

Der Ausgangspunkt unserer Untersuchungen sind Entwurfprozesse. Diese sind kooperative Vorgänge, da sie einen hohen Interaktionsbedarf und damit allgemein einen hohen Kooperationsbedarf haben. Das spiegelt sich vor allem in dem gemeinsamen Ziel und den gemeinsam genutzten Ressourcen wider. Da die Arbeitsschritte in einem Entwurfsprozeß von den ggf. vorläufigen Ergebnissen gleichzeitig stattfindender Arbeitsschritte abhängen, ist ein vorab definierter Datenfluß i.d.R. zu starr. Die gemeinsamen Ressourcen sollten also kooperativ genutzt werden, um eine konfliktfreie und zügige Durchführung eines Prozesses zu ermöglichen. Zudem sind direkte Kooperationen notwendig, bspw. um sich abzusprechen.

Durch diese Charakterisierung von Entwurfsprozessen wird deutlich, daß wir als Systemunterstützung kooperative und regelnde Technologien benötigen, die uns idealerweise Flexibilität, wohl-strukturierte Teilprozeß-Steuerung und Kooperationsmechanismen erlauben. Im folgen-

den werden wir zwei Beispiele für Entwurfsprozesse vorstellen und an einem weiteren Szenario darlegen, daß diese Forderungen nicht nur auf Entwurfsprozesse zutreffen, sondern auch in anderen Bereichen bestehen.

2.3 Beispielszenarien

In diesem Abschnitt werden wir zwei Szenarien für Entwurfsprozesse besprechen. Zuerst wird ein CAD-Entwurf für einen Chip vorgestellt, der unser Ausgangspunkt für die Untersuchung von Entwurfsprozessen dargestellt hat. Danach wird das Software-Engineering besprochen, das ähnliche Anforderungen besitzt.

Zuletzt präsentieren wir einen Planungsprozeß, der viele Merkmale der vorherigen Prozesse aufweist, obwohl hier ein unterschiedliches Anwendungsgebiet vorgestellt wird. Dies soll zeigen, daß die Anforderungen im Entwurf auch in anderen Bereichen bestehen. Entsprechend werden wir dies in die Aufgabenstellung einbeziehen.

2.3.1 Entwurf eines Chips

Der Entwurf eines komplexen Chips stellt ein typisches Anforderungsprofil für Entwurfsprozesse dar. Im folgenden werden wir einen Überblick der prinzipiellen Abläufe und bestimmter Charakteristika geben. Aufgrund der Komplexität von ICs für heutige Chips wird der VLSI-Entwurf (Very Large Scale Integration) typischerweise von Teams in zweistelliger Größe durchgeführt. Beispiele für bestimmte Phasen im Entwurf sind (vgl. auch [ANTREICH 2000]):

- Systementwurf
  Die grundlegenden Eigenschaften müssen definiert werden. Dazu gehören physische Merkmale wie Maße (z.B. die *Module Dimensions* in Abbildung 2), Produktionsprozeß (z.B. 13 µm), oder Taktung. Die funktionalen Forderungen sind besonders wichtig (z.B. Befehlsatz ist Intel-Pentium-kompatibel). Diese Spezifikation ist zu verfeinern, bis die Untergruppen eines Chips an einzelne Bearbeiter delegiert werden können, z.B. der Cache und die FPU (siehe *Block Diagram* in Abbildung 2). Dies führt zu einem hierarchischen Grobentwurf der Schaltung und Logik.

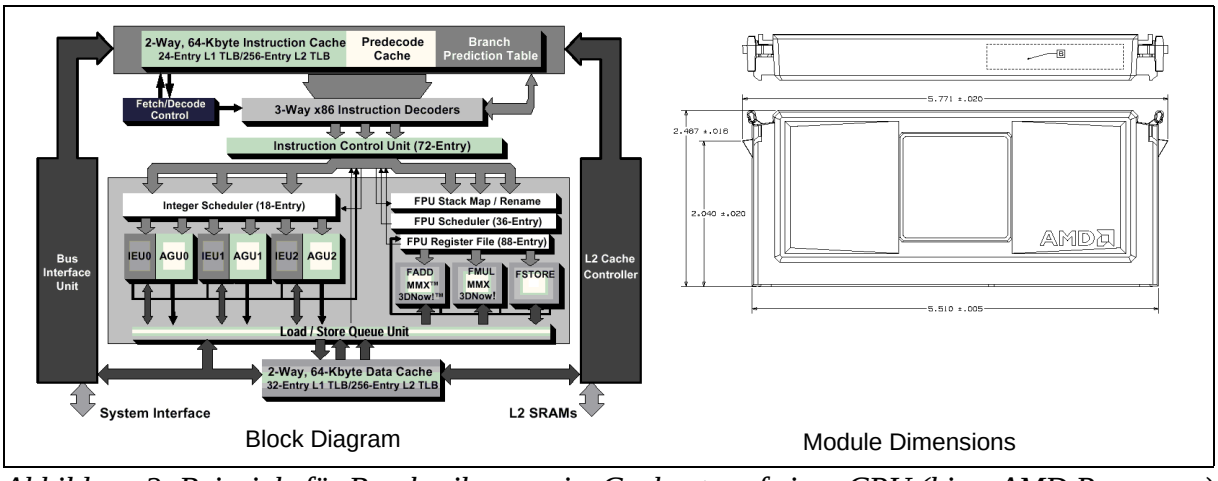


Abbildung 2: Beispiele für Beschreibungen im Grobentwurf einer CPU (hier: AMD Prozessor)

- Logik- und Schaltungsentwurf
  Um die funktionalen Anforderungen zu erfüllen, muß die Gatterlogik des Chips erstellt werden. Die Untergruppen müssen geeignete Schnittstellen zwischen den Einzelentwürfen realisieren um die Gesamtlogik auch zu erreichen. Diese Entwürfe können durch Simulations-Software (z.B. SPICE [VLADIMIRESCU UND LIU 1980]) getestet werden.

- Physisches Layout
  Die Logik wird den physischen Anforderungen angepaßt. Dazu findet eine Abbildung der Schaltung auf die physischen Strukturen statt, um letztendlich eine sog. Maske für die Produktion zu erhalten. Dieser letzte Schritt kann weitgehend automatisch erfolgen, jedoch sind Eingriffe durch Spezialisten bei diesen komplexen Entwürfen erforderlich.

Die drei vereinfachten Teilbereiche des Entwurfs besitzen jeweils verschiedene Merkmale bezüglich der Arbeitsweise. Die Spezifikation kann als eine Diskussion zur Festlegung der Anforderungen an einen Entwurf aufgefaßt werden. Die Verfeinerung und Änderung dieser Spezifikation findet aber im Rahmen aller Phasen statt, da dort Teilentwürfe im Rahmen einer Delegation entstehen oder auftretende Probleme eine Änderung der Spezifikation zur Folge haben können.

Der größte Anteil interaktiver und kooperativer Arbeit fällt im Bereich des Logik- und Schaltungsentwurfs an, da die verschiedenen Entwerfer Abhängigkeiten beachten müssen und sich mit ihren Teilentwürfen abstimmen müssen.

Das physische Layout zu erzeugen, ist ein Teilprozeß zur Vorbereitung auf die Produktion. Für die Abbildung von Schaltungen auf 'Siliziumstrukturen' gibt es automatisierte Werkzeuge. Die Ergebnisse müssen jedoch überprüft und ggf. überarbeitet werden, z.B. auf elektromagnetische Verträglichkeit und Layout-Fehler. Dies kann sogar eine Überarbeitung der Logik zur Folge haben, bspw. wenn der derzeitige Logikentwurf es nicht erlaubt die physikalischen Spezifikationen einzuhalten (z.B. zuviele Transistoren). Heutzutage ist auch die effiziente Prüfung produzierter Chips besonders wichtig, da der Produktionsprozeß sehr fehleranfällig ist. Daher gehört auch die Erzeugung von Testmustern zur Ausgabe [FRANK ET AL. 1992]. Die Testmuster erlauben es, hergestellte Chips auf ihre Funktionsfähigkeit zu prüfen.

Damit ist der Ablauf dieser Phasen dem aus dem Software-Engineering bekannten Wasserfall-Modell ähnlich, wie in Abbildung 3 dargestellt. Dort werden auch die entsprechenden Repräsentationsmodelle im Entwurf aufgeführt. Diese gehen vom abstrakten Niveau bis hin zum konkreten physikalischen Modell, das die Erzeugung einer Maske für die Produktion ermöglicht.

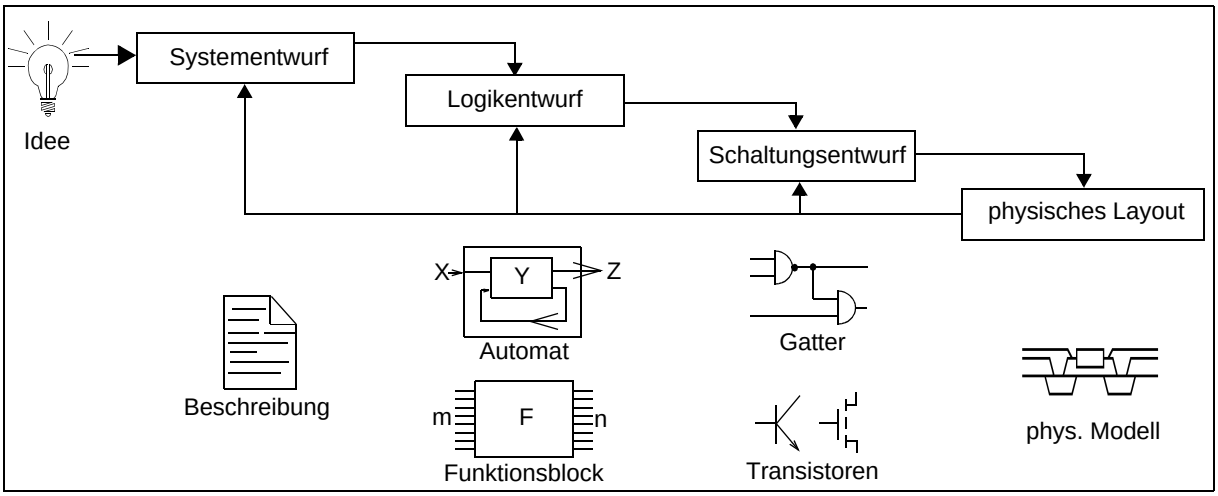


Abbildung 3: Vereinfachtes Phasenmodell mit den entsprechenden Modellen für VLSI-Entwurf

Aufgrund der großen Komplexität von Chip-Entwürfen (>10 Mio. Transistoren) sind diese von einer Gruppe von Personen durchzuführen. Um einen Entwurf zu realisieren wird der Gesamtentwurf immer weiter hierarchisch in Teilentwürfe zerlegt ("divide et impera"), bis die Komplexität die Bearbeitung eines solchen Teilentwurfs durch eine einzelne Person möglich ist. Den Vorgang der Zerlegung mit nachfolgender Unterauftragsvergabe und abschließender Integration nennen wir *Delegation*. Um den Entwurf zu beenden ist die Zusammenführung der Teilent-

würfe notwendig. Dieser Vorgang ist schematisch in Abbildung 4 skizziert, wobei nicht alle Ebenen der Zerlegung und Integration aufgeführt wurden. Wie bereits im Phasenmodell gezeigt bestehen zwischen den einzelnen Teilentwürfen und Vorgängen Iterationen, die im Bild rechts angedeutet sind.

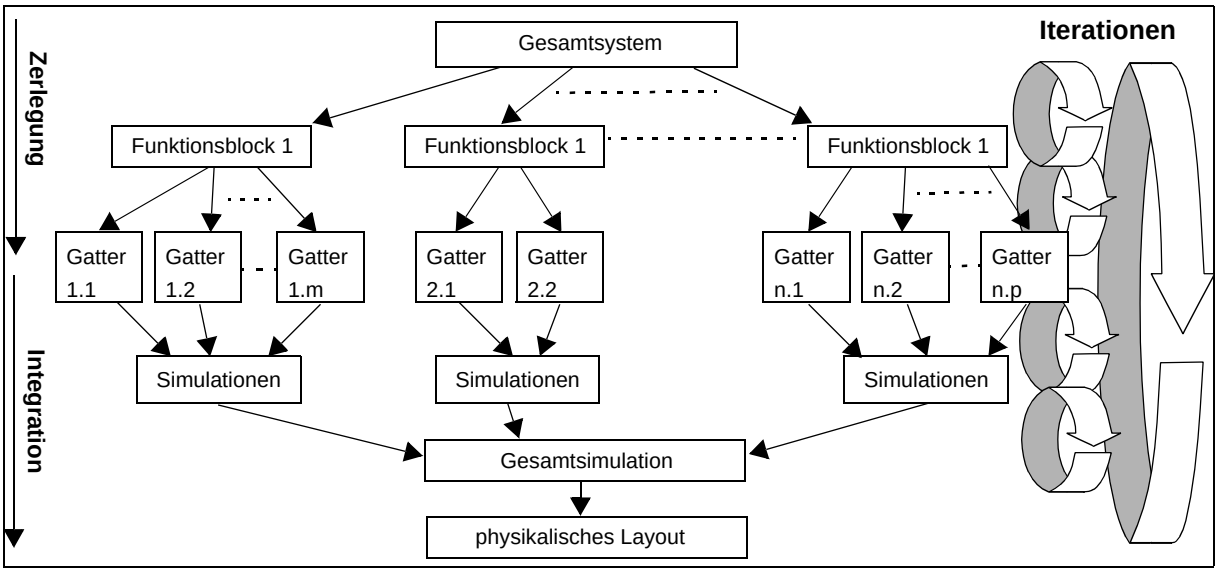


Abbildung 4: Zerlegung und Integration eines Entwurfs

Bei der Zerlegung des Entwurfs bestehen aber auch Probleme, wie etwa durch fehlerhafte Teilspezifikationen oder ungenügende Abstimmung. Diese werden im schlechtesten Fall erst nach der Integration erkannt und bedeuten einen fehlerhaften Entwurf[1]. Wenn dies am Ende des Entwurfs geschieht ist eine Fehlersuche teuer und zeitaufwendig. Daher wird versucht auftretende Fehler möglichst frühzeitig zu erkennen. Maßnahmen zur Unterstützung einer fehlerfreieren Arbeit können sein:

- Perfekte Spezifikation und Zerlegung
  In diesem Fall ist die Spezifikation so exakt und auf die Zerlegung zugeschnitten, daß eine Erfüllung der Spezifikation auch einen funktionierenden Gesamtentwurf bedingt. Dies ist offensichtlich kaum möglich, jedoch erstrebenswert.
- Automatisierung
  Durch eine weitgehende Automatisierung und Ablaufsteuerung können viele Koordinationsprobleme und Werkzeuganwendungen korrekt durchgeführt werden. Jedoch bergen Prozesse dieser Art (wie oben bereits beschrieben) aber kreative Anteile, die nicht vollständig automatisierbar sind.
- Design-Guidelines und -Regeln
  Ein Regelwerk, welches das Vorgehen beim Entwurf be- und vorschreibt, vereinfacht es, Konflikte und Probleme zu lösen, da entsprechende Vorgehensweisen vorgeschrieben sind. Diese sind an sich aber nur selten durch ein System unterstützt (vgl. dazu [SELLENTIN ET AL. 1999]).
- Besprechungen
  Zur Abstimmung der Teilentwürfe sind Besprechungen, bspw. via E-Mail, Telefon oder in Sitzungen sinnvoll, da sich hier die einzelnen Personen schnell abstimmen können.
- Propagation von Arbeitsergebnissen
  Da viele Probleme daraus resultieren, daß einem einzelnen Entwerfer nicht alle Arbeiten seiner Kollegen bekannt sind, bietet es sich an diese Arbeiten zu propagieren. Ein Beispiel

1. Siehe bspw. die zahlreichen Pressemeldungen bzgl. der Fehler in den Intel Pentium Chips.

hierfür ist etwa eine Technologie, wie ein Gruppeneditor (siehe Abschnitt 3.3.3). Dadurch werden Einflüsse von Arbeiten anderer und Konflikte schneller offensichtlich und können entsprechend früher gelöst oder vermieden werden.

- Frühzeitige Integration und Tests
  Es bietet sich an frühzeitig Teilergebnisse auszutauschen, zu integrieren und zu testen. Damit wird die Wahrscheinlichkeit verringert, Fehler erst spät zu erkennen.

Die beschriebenen Probleme können anhand eines Beispiels verdeutlicht werden. In Abbildung 5 wird ein häufig zu wiederholender Teilprozess des Schaltungsentwurfs dargestellt. Er beginnt mit der Erstellung eines ersten Schaltplans und wird mit der Extraktion und Konvertierung der CAD-Daten in das Format eines Schaltungssimulators (bspw. Netzlisten für SPICE) fortgeführt. Anhand der vorliegenden Daten kann nun eine Schaltungssimulation durchgeführt werden. Das Ergebnis muß anschließend bewertet werden, um zu entscheiden, ob eine Überarbeitung des Schaltplans erforderlich ist, oder dieser zufriedenstellend erstellt wurde.

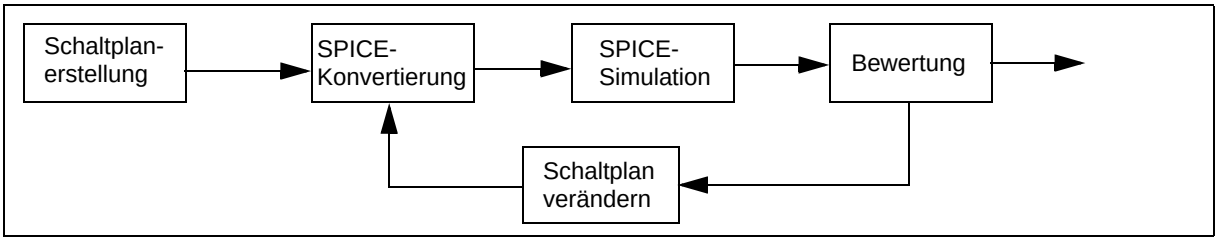


Abbildung 5: Prozeß für eine Schaltplanerstellung (siehe auch [RYBA 1996]*)*

In dem skizzierten Teilprozess fehlt beispielsweise die Abbildung von Kooperationen in der Schaltplanerstellung oder auch eine Rückfrage bezüglich des Ergebnisses an andere Entwerfer, die vielleicht bei auftretenden Problemen helfen könnten. Es werden auch nicht die Konsequenzen aus diesem Teilentwurf bezüglich der anderen Entwürfe berücksichtigt, dies wird der Erfahrung und dem Wissen des Bearbeiters überlassen. Wünschenswert wäre hingegen ein frühes Erkennen von Auswirkungen des eigenen Entwurfs auf andere Teilentwürfe. Der prinzipielle Vorgang ist aber ansonsten regelbar, wie Abbildung 5 suggeriert.

Wie wir an diesem Beispiel für Entwurfsprozesse gesehen haben, zeichnet sich der Chip-Entwurf durch seine Entwurfsartefakt-Zerlegung, Team-Arbeit und den semi-strukturierten Ablauf aus. Ein Entwerfer ist auf die Ergebnisse anderer angewiesen. Um frühzeitig Konflikte zu erkennen ist es aber vorteilhaft auch Zwischenergebnisse bekannt zu machen. Dies führt zu einer Interaktion zwischen den beteiligten Entwerfern, also kooperativem Vorgehen. Die Spezifikation als Teil der Vergabe von Unteraufträgen spielt eine wichtige Rolle für eine hierarchische Struktur der Aufgabenabarbeitung. Da diese Unteraufträge nicht unabhängig voneinander sind, ist es sinnvoll, Mechanismen wie das zuvor beschriebene Bekanntmachen von Zwischenergebnissen zu realisieren und auch Mechanismen zur Änderung von Spezifikation und zur Verhandlung einzuführen um Fehlerfälle aufzulösen.

2.3.2 Das Vorgehen in der Software-Entwicklung

Zu den Entwurfsprozessen zählt man auch das Software-Engineering. Es existieren bereits viele Untersuchungen bezüglich des Teamcharakters und Koordinationsmechanismen in der Software-Entwicklung, weswegen wir darauf auch detaillierter eingehen können. Wie im obigen Entwurfsprozeß stehen auch Referenz- bzw. Phasenmodelle zur Beschreibung der Vorgehensweise zur Verfügung. Als Beispiel für solche Modelle finden sich in Abbildung 6 das sog. Wasserfall- und Spiralmodell. Das Wasserfallmodell zeigt die einzelnen Phasen im Software-Engineering, die in einer festen Reihenfolge wiederholt angewendet werden können. Das Wasserfallmodell wird durch vier Hauptphasen: *Festlegung von Zielen*, *Beurteilung*, *Entwicklung*

und Validierung, *Planung*, und einen inkrementellen Fortschritt von Prototyp zu Prototyp gekennzeichnet. Der interessierte Leser kann z.B. in [POMBERGER UND BLASCHEK 1996] eine genaue Beschreibung dieser Modelle finden.

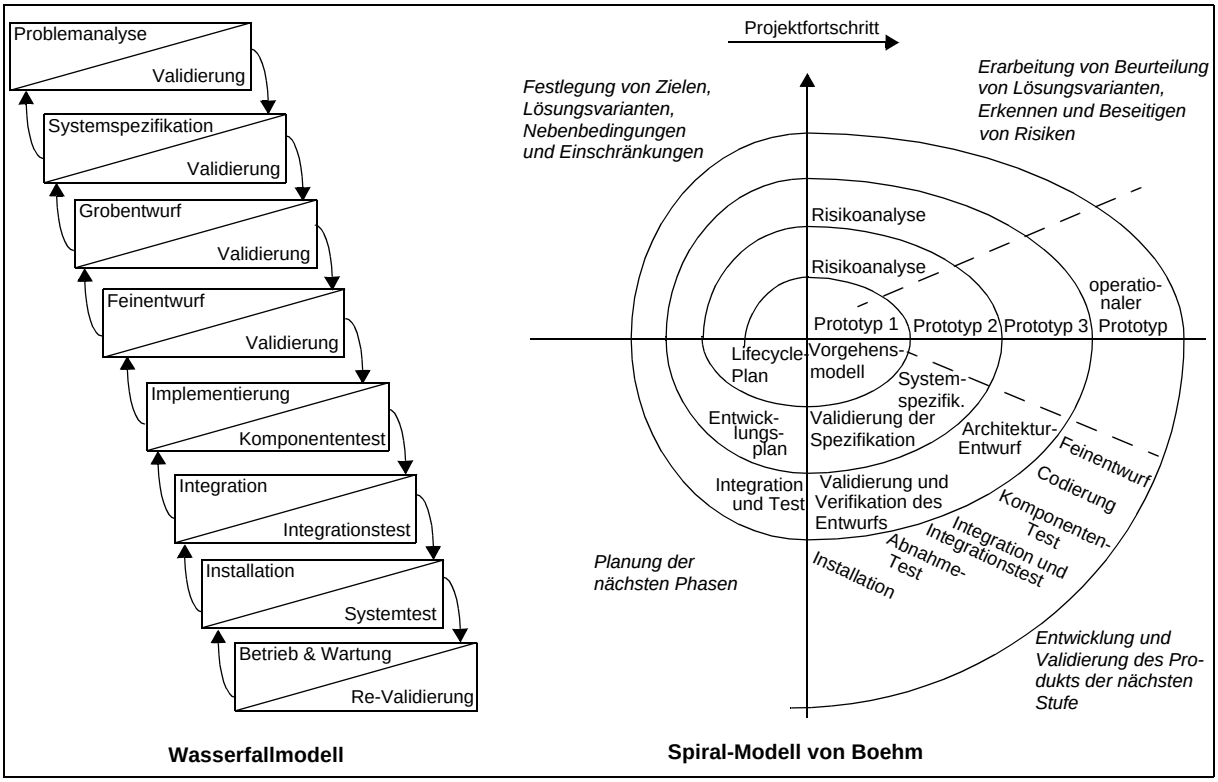


Abbildung 6: Zwei Phasenmodelle für das Software-Engineering

Die Referenzmodelle zeigen auch hier, wie schon beim CAD-Entwurf, eine iterative Vorgehensweise zwischen den Phasen und die Zerlegung des Entwurfs (z.B. in Komponenten) zur Reduktion der Komplexität. Anstatt der Erzeugung eines physischen Modells für einen Chip, wird im Software-Engineering ein fertiges Softwareprodukt entwickelt. Den verschiedenen Phasen kann man - ähnlich dem Chip-Design - auch Dokumente, wie Spezifikation, Entwurf, Modul-Diagramm und Quelltext, zuordnen.

Die Werkzeuganwendungen im Falle der Implementierung umfassen Zyklen, wie das Schreiben von Quelltext, Kompilieren, Testen und Versionieren. Im Gegensatz zum CAD-Entwurf, scheint aber das Software-Engineering besonders stark an konfliktträchtiger Entwicklung zu leiden, wie die vielen Untersuchungen bezüglich des Teamcharakters (s. u.), geeigneter Vorgehensmodelle (V-Modell, Prototyping, objektorientierte Analyse und Design, ...) und Implementierungsmodellen belegt (objektorientiertes Programmieren, Komponentenmodelle, Frameworks, ...). Ein wesentlicher Unterschied zum CAD-Entwurf, ist die komplette Entwicklung eines Produkts. Wir betrachten beim Software-Engineering nicht nur den Entwurf sondern auch dessen Implementierung und ggf. Anwendung und Wartung. Jedoch unterscheiden sich die kreativen Anteile in der Implementierung relativ wenig von bspw. dem Schaltplanentwurf. In Abschnitt 3.7.1 stellen wir ein Beschreibungsmodell für Software-Prozesse vor, das auch Interaktion zwischen Arbeitsschritten realisiert.

2.3.2.1 Teams im Computer Aided Software Engineering (CASE)

Wie oben angedeutet sind Teams ein typisches Merkmal der Software-Entwicklung. In diesem Abschnitt werden wir die Bedeutung von Teamarbeit darstellen, die prinzipiell in allen Ent-

wurfsprozessen gelten. [BALZERT 1998] nennt die nachfolgenden Charakteristika von Teamarbeit für den Software-Prozeß:

- Regelmäßige und kontinuierliche Kommunikation untereinander
- von Fall zu Fall gegenseitige Abstimmung
- Gleichberechtigte Mitbestimmung aller Teammitglieder bei der Diskussion von Methoden, Inhalten und Zielen der Arbeit und ihrer Durchführung
- Alle Teammitglieder sind gleichrangig und agieren auch gleichrangig
- Verschiedene Teammitglieder übernehmen zeitweise die Führungsrolle, jeweils auf einem Gebiet, auf dem sie ihre Stärken haben.

Die Struktur eines Teams ist ein Netzwerk und keine Hierarchie. Manager sind normalerweise nicht Teil des Team, das sie leiten. Besonders bewährt sich Teamarbeit zu Beginn der Systemanalyse und des Software-Entwurfs, da dort der Bedarf besteht gemeinsam wichtige Entscheidungen zu treffen. Wichtig für Teams sind konkrete und auch als solche akzeptierte Ziele ihrer Arbeit. Ein vielfältiges Team hat höhere Erfolgschancen (z.B. bestehend aus Mitarbeitern, Endnutzern, Entwicklern). Zudem müssen Teams möglichst autonom arbeiten können.

Kreativität ist notwendig für die Software-Entwicklung. Sie kann sich in Gruppen besonders gut entfalten, vorausgesetzt es ist eine offene Kommunikation über das Wissen der verschiedenen Teilnehmer möglich. Methoden sind bspw. Brainstorming und Brainwriting, die etwa durch die Metaplan-Technik (Pinnwände, Flip-Charts, etc.) unterstützt werden können. In Abschnitt 3.3 werden CSCW-Technologien beschrieben, welche die hier beschriebenen Anforderungen realisieren.

2.3.2.2 Koordinationsmechanismen in der Software-Entwicklung

Für die Durchführung eines Entwicklungsprozesses ist neben der Ablauforganisation (Prozeß-Modell) auch die Aufbau-Organisation (z.B. hierarchisch) notwendig, da mehrere Personen an dem Prozeß beteiligt sind. Nach [MINTZBERG 1992] kann man folgende Interdependenzen und Koordinationsmöglichkeiten für Arbeitsabläufe extrahieren:

- *gegenseitige Abstimmung*
  Dies erfolgt i.a. durch informelle Kommunikation, die Arbeitskontrolle liegt bei den Mitarbeitern selbst. Daher gibt es auch keinen Umweg über Hierarchie, was nach [BORGHOFF UND SCHLICHTER 1995] typisch für Gruppenarbeit ist.
- *persönliche Weisung*
  Diese Art der Weisung ist häufig mit hierarchischen Strukturen einer Organisation verbunden, in der eine “Führungskraft” Weisungsbefugnisse gegenüber ausführenden Mitarbeitern besitzt.
- *Standardisierung der Arbeitsprozesse*
  Die Koordinierung erfolgt mit Hilfe von festgelegten und vorausgeplanten Arbeitsgängen.
- *Standardisierung der Arbeitsprodukte*
  Die Ergebnisse der Arbeit (z.B. Funktionsumfang, Qualität) sind etwa durch eine Spezifikation festgelegt und können somit überprüft werden.
- *Standardisierung der bei den Mitarbeitern vorauszusetzenden Qualifikationen*
  Die Ausbildung, die für die Ausführung bestimmter Arbeiten notwendig ist, ist festgelegt. Aufgrund der Kenntnis der Ausbildungsinhalte ist bekannt, was man von den Mitarbeitern erwarten kann (bspw. objektorientiertes Programmieren).

Diese fünf Mechanismen werden in den meisten Organisationen kombiniert eingesetzt. Sowohl die persönliche Weisung, als auch die gegenseitige Abstimmung sind generell erforderlich. Nach [BALZERT 1998] sind die gegenseitige Abstimmung, standardisierte Qualifikationen und Arbeitsprozesse die vorrangigen Koordinationsmechanismen in der Software-Entwicklung. Die Verhaltensformalisierung wird als 'organisch' und bürokratisch angesehen. Dies bedeutet, daß die Mitarbeiter gewisse Freiheitsgrade in ihrer Arbeit geniessen. Die Fertigung von Software kann als teilweise automatisiert angesehenen werden, da es wie die Phasenmodelle bereits andeuten, gewisse Beschreibungen der Arbeitsabläufe festgelegt werden können.

[HUMPHREY 1995] propagiert eine Art der Unterstützung für die Durchführung der Software-Entwicklung, die sich als einen stark dokumentenorientierten Ansatz bezeichnen läßt. So werden vielen Arbeitschritten und Ergebnissen Dokumente zugeordnet, die es erlauben den Projektfortschritt zu kontrollieren und die Qualität der Entwicklung anzuheben. Dies ist besonders für größere Teams interessant, die nicht mehr einfach überschaubar sind.

In [BALZERT 1998] werden Untersuchungen vorgestellt (u.a. [DEMARCO UND LISTER 1987] und [GRADY 1992]), die belegen, daß für Software-Entwickler eigene Arbeitsumgebungen in denen sie 'abtauchen' können sehr bedeutsam für deren Leistungssteigerung ist. So sind ca. 15 Minuten ungestörter Konzentration notwendig um eine Aufgabe effizient beginnen zu können. Dies bedeutet jedoch nicht, daß Software-Entwickler am besten isoliert von anderen Mitarbeitern arbeiten sollten. Vielmehr hat es sich gezeigt, daß dies nur für etwa 30% ihrer Zeit zutrifft. Sehr wichtig sind auch die Phasen der Zusammenarbeit zu zweit (~50%) oder in größeren Teams (~20%). Eine solche Aufteilung in synchrone und asynchrone Phasen ist typisch für kooperative Arbeiten und wird in Abschnitt 3.3 noch einmal genauer dargestellt.

Wir sehen, daß es sich offensichtlich bei der Software-Entwicklung um Prozesse handelt, die sowohl automatisierbare Teile enthält, als auch die persönliche Abstimmung zwischen Teilnehmern benötigt, also nicht vollkommen strukturiert ist, sondern auch flexible Handlungsweisen unterstützen muß. Teamarbeit ist ein wichtiger Faktor für das Software-Engineering und muß entsprechend unterstützt werden. Vor allem die direkte Kommunikation und der Austausch zwischen Personen ist wichtig.

2.3.2.3 Einordnung und Anforderungen an die Unterstützung von Software-Prozessen

Unsere Erfahrungen und Untersuchungen [MARIUCCI 1998] legen die Vermutung nahe, daß Software-Prozesse i.a. nicht sehr feingranular zu definieren sind, sondern daß im wesentlichen auf Phasen-Modelle zurückgegriffen wird. Zwar lassen sich bestimmte Aspekte gut beschreiben, jedoch erschweren unserer Meinung nach die iterativen Phasen und Aktivitätsdurchläufe zum einen den effizienten Einsatz von automatisierenden Systemen (bspw. Workflow-Management-Systeme) und zum anderen entsteht durch die typischerweise häufig auftretenden Änderungen in der Spezifikation bzw. Änderungen des Entwurfs ein hoher Bedarf an Flexibilität für den Prozeßverlauf und Kooperationsmöglichkeiten der beteiligten Personen.

Kleine Softwareprojekte besitzen grundsätzlich die gleiche Struktur wie größere Projekte, jedoch ist die Abstimmung der Teams (und in den Teams) aufgrund der geringeren Anzahl von Projektmitarbeiter prinzipiell einfacher. Beispielsweise werden dann die Spezifaktionen nicht unbedingt so ausführlich erstellt (etwa Schnittstellenbeschreibungen), da die Team-Mitglieder vielleicht im gleichen Raum arbeiten und somit Nachfragen wenig aufwendig sind.

In den Implementierungsphasen ist eine umfassende Prozeß-Unterstützung schwer, da hier konzeptionelle Arbeit, Quelltext-Schreiben, Integrieren und Testen wiederholt angewendet werden. Stattdessen werden Synchronisationsmittel wie ein gemeinsames Dateisystem oder eine entsprechende CASE-Umgebung (wie etwa die integrierte Entwicklungsumgebung SNiFF+

[WINDRIVER 2001]), ggf. mit Versionierung, eingesetzt. Eine Implementierungsphase kann über Monate hinweg andauern. Dabei werden Entwürfe verfeinert und geändert und gemeinsame Entwicklungen abgeglichen. Da Änderungen eines Programmfragmentes häufig Änderungen in anderen nach sich ziehen, kann auch nicht von unabhängigen Subaktivitäten gesprochen werden Wir halten es daher für sehr schwer, feingranulare Arbeitsschritte für diese Phase anzugeben, die asynchron voneinander ausgeführt werden können. So sind Besprechungen zwischen Entwicklern während der Implementierung eines Moduls oder sogar nur einer Programmzeile ad hoc notwendig. Dies ist nur mit Einschränkungen mit Automatisierungsunterstützung möglich, da die späte Definition wesentlich zu zeitaufwendig gegenüber ihrem Nutzen wäre. Außerdem würde der Entwickler zu sehr in seinem Arbeitsfluß gestört, wenn er erst Eingaben im Automatisierungssystem machen müßte, um kurzfristig eine Besprechung zu starten.

2.3.2.4 Die Software-Entwicklung als Entwurfsprozeß

Die Software-Entwicklung weicht inhaltlich von den typischen Entwurfsprozessen ab. Allgemeine Entwurfsprozesse (z.B. Automobilentwurf, Chipdesign) bedeuten meist die Fertigstellung eines Bauplans für ein Produkt. Das Produkt selbst wird in einem gesonderten Fertigungsprozeß industriell erstellt. Die Software-Entwicklung umfaßt die Phasen Analyse, Entwurf, Implementierung und Wartung, also ist der Entwurf eigentlich nur ein Teilprozeß. Jedoch sind diese Phasen iterativ und somit ist der Entwurf stark mit den anderen Phasen verzahnt.

Da Software kein Produkt in dem Sinne ist, daß sie in größeren Mengen mit relativ hohem Aufwand gefertigt wird, sondern ein Unikat mit beliebig vielen billigen Kopien, entfällt der industrielle Fertigungsprozeß. Auch bei der Implementierung sind häufig noch (kleine) Entwurfsentscheidungen zu treffen bzw. zu redigieren. Zudem werden in dieser Phase auch Werkzeuge in einer ingenieursähnlichen Weise verwendet, welche die Abgrenzung zu einem CAD-Entwurf natürlich erschweren. Unserer Meinung nach kann man daher die Software-Entwicklung den Entwurfsprozessen zuordnen, besonders wegen der flexiblen ingenieursmäßigen Vorgehensweise. Außerdem basiert die Software-Entwicklung auch auf der typischen Zerlegung des Produkts in Teile und deren Bearbeitung durch Teams. Letztendlich glauben wir, daß auch hier der Einsatz von Workflow-Technologie keine allumfassende befriedigende Lösung zur Unterstützung von Software-Entwicklungsprozessen ist. Techniken, die hilfreicher sind, umfassen etwa: Versionierung, Datenaustausch, flexible Auswahl von Werkzeugen, Ad-Hoc Entscheidungen über Vorgehensweisen, gemeinsame Datenbestände, Verwendung von Zielspezifikationen, Gruppenkommunikation. Die Anforderungen sind damit sehr ähnlich den vorgestellten Entwurfsprozessen und verlangen eine entsprechend gleiche Unterstützung (siehe auch [DART 1992] und [RYBA 1996]).

2.3.3 Planungsszenario: Filialen

Wir werden nun ein konkretes Szenario als Beispiel für die Durchführung einer Planung ausführlicher besprechen, nämlich die Errichtung neuer Filialen. Dieses Beispiel soll zum einem verdeutlichen, daß auch in Szenarien wie der Planung ähnliche Anforderungen bei typischen Entwurfsanwendungen bestehen. Zum anderen soll auch dargestellt werden, welche arbeitsbedingten Situationen (bspw. Kooperationen, Delegation etc.) in der Beschreibung eines Arbeitsvorgangs zu berücksichtigen sind.

2.3.3.1 Überblick

Will eine Handelskette expandieren, insbesondere durch die Eröffnung neuer Filialen in neuen Regionen, so beauftragt die Geschäftsleitung ihre Abteilungen mit der Standortplanung zu beginnen. Dazu ist eine detaillierte Marktanalyse, eine Standortsuche und eine Kostenabschätzung

durchzuführen. An eine erfolgreiche Standortplanung schließt sich das eigentliche Bauvorhaben und die spätere Inbetriebnahme an. Alle diese Aktivitäten sind sehr komplex. Im folgenden wollen wir auf den Teilvorgang der Standortplanung detailliert eingehen.

Die Standortplanung wird aktiviert und kontrolliert von der Geschäftsführung. Es werden drei Teilaktivitäten delegiert:

- *Marktanalyse*
  Hier wird der Markt analysiert und entschieden, in welcher Region eine Filiale profitabel wäre.
- *Standortsuche*
  In der ausgewählten Region werden nun geeignete Standorte für eine neue Filiale gesucht und deren Merkmale festgehalten (Größe, Zustand, Grundstückspreis, Gewerbesteuer, Miete, Auflagen usw.).
- *Kostenabschätzung*
  Im Rahmen der Kostenabschätzung wird für die vielversprechendsten Kandidaten der Standortsuche eine Berechnung der zu erwartenden Gesamtkosten für diese Filialen durchgeführt.

Das Ziel der Standortplanung ist die Erstellung eines Planungsberichtes, der die Entscheidungen und Konsequenzen für einen Standort dokumentiert. Die Marktanalyse ist eine gut strukturierte Tätigkeit (z.B. als Workflow modellierbar). Wie wir noch sehen werden, handelt es sich bei der Standortsuche und der Kostenabschätzung um Tätigkeiten, die umfangreiche Interaktionen und Kooperationen erfordern. Die simultane Delegation der genannten Tätigkeiten (siehe auch Abbildung 7) innerhalb der Standortplanung hilft uns hier, Resultate der Aktivitäten frühzeitig auszutauschen und möglicherweise verfeinern zu lassen, wie auch durch gleichzeitige Bearbeitung eine deutliche Reduktion der Gesamtdauer zu erzielen. Die Standortsuche und Kostenabschätzung sind eng miteinander verwoben. So hat etwa die Änderung bei dem Verhandlungsstand in der Standortsuche (z.B. Grundstückspreis heruntergehandelt) Auswirkungen auf die zuvor berechneten Gesamtkosten. Andererseits kann bei einem Vergleich der Gesamtkosten eine neue Verhandlungsvorlage für einen bereits besichtigten Standort geschaffen werden (“In der Nachbargemeinde müssen wir aber nur die Hälfte der Gewerbesteuer zahlen!”).

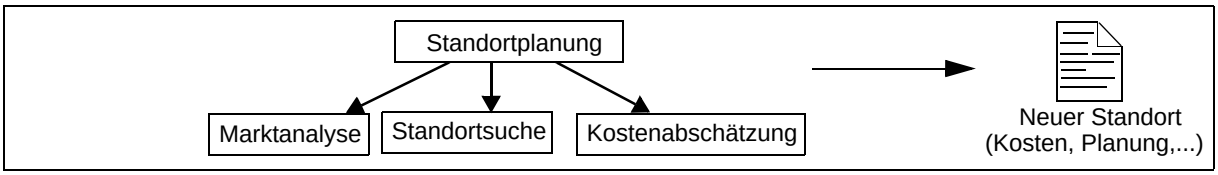


Abbildung 7: Delegation der Hauptaktivitäten einer Standortplanung

2.3.3.2 Detaillierte Aktivitätenbeschreibung

Zu Beginn der Standortplanung beschließt die Geschäftsführung eine Marktanalyse durchführen zu lassen, um eine Region zu finden, die vielversprechend für eine neue Filiale ist. Die Arbeit des Analystenteams ist weitgehend bekannt und strukturiert. So wird die Erhebung von geographischen Marktdaten in Auftrag gegeben und die Verteilung von Filialen der Konkurrenz untersucht. Die ermittelten Daten werden in einem Dokument zusammengefaßt. Daraus resultiert der Arbeitsschritt *Zusammenfassen* (siehe Abbildung 8). Letztendlich ist zu entscheiden, welche Lage auf Basis der vorliegenden Daten strategisch und marktpolitisch günstig ist (z.B. in Form einer Gruppensitzung).

Um nun einen Standort in einer ermittelten Region zu finden, gibt die Geschäftsführung den Auftrag, dort eine Standortsuche durchzuführen (Abbildung 9). Diese umfaßt beispielsweise die Nachfrage bei den betroffenen Gemeinden, das Ermitteln von Gewerbezonen, das Aufgeben

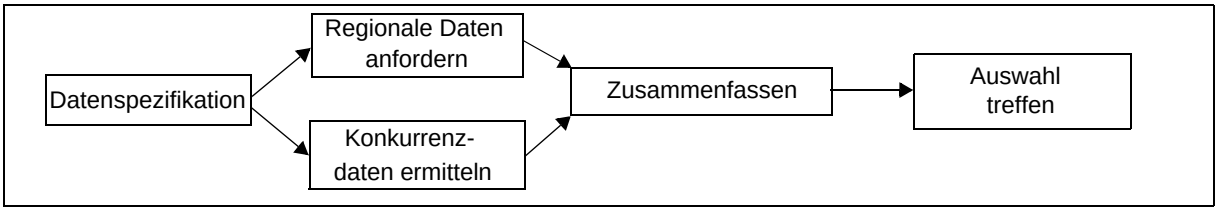


Abbildung 8: Aktivität für Marktanalyse

von Immobilienanzeigen, das Einschalten von Maklern usw. Auf diese Weise gefundene Areale müssen preislich bewertet werden (abgebildet als *Bewertung 1* bis *Bewertung n*). Dazu können aufwendige und lange Verhandlungen notwendig sein. Es soll eine Liste erstellt werden, die bestimmte Kriterien (Größe, Preis, Lage, etc.) der Standorte zusammenfaßt. Die interessantesten Kandidaten sollen zur Kostenabschätzung weitergereicht werden. Aufgrund der möglicherweise langwierigen Suche behält es sich die Geschäftsführung vor, vielversprechende Kandidaten dieser Liste frühzeitig in die Kostenabschätzung zu übernehmen (die Realisierung dieser Beziehung wird weiter unten besprochen).

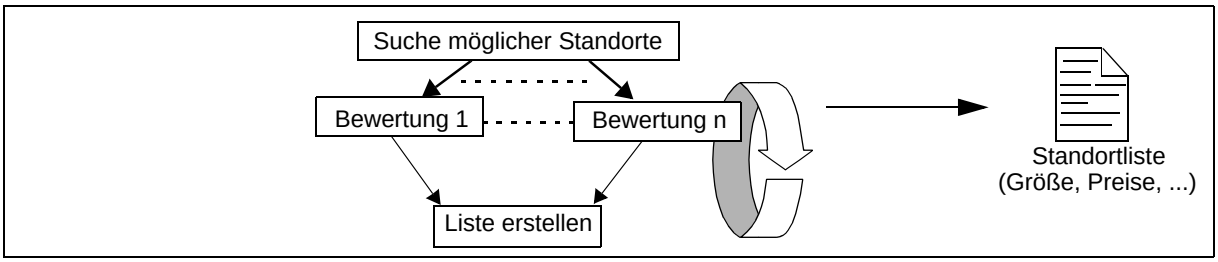


Abbildung 9: Aktivität für Standortsuche

Die Aktivität *Suche möglicher Standorte* delegiert die Bewertungen 1 bis n. Diese Delegation muß sicherstellen, daß bei einem Fehlschlag aller Verhandlungen über die Areale die Suchkriterien erweitert werden können, um eine breitere Anzahl an Standorten zu finden. Zudem können die Verhandlungen besser koordiniert werden, wenn die delegierende Aktivität vorläufige Ergebnisse der Bewertungen erhält und so frühzeitig eingreifende Maßnahmen treffen kann. Die Arealbewertungen können ihrerseits auf die ermittelten Daten in der Liste (z.B. Preise für Grundstücke oder Miete) zugreifen, um diese für eine bessere Verhandlungsposition nutzen zu können. Die Liste kann im wesentlichen automatisch erstellt werden. Sie erfaßt lediglich die Daten, sortiert diese und stellt sie in einer Standortdatenliste zur Verfügung.

Bei der Kostenabschätzung werden die Grundstückspreise aus der Standortsuche und die vermutlichen Bau- und Bestückungskosten der Filiale errechnet (Abbildung 10). Sehr wichtig für die Handelskette ist auch die Versorgung durch Zentrallager. Daher ist zu entscheiden, welchem Lager eine Filiale zugeordnet ist. Ein Endresultat dieser Bewertung könnte sein, daß ein in Frage kommendes Lager erweitert werden muß, da dessen Kapazität nicht ausreicht, um auch die neue Filiale zu beliefern. Dies würde natürlich die Gesamtkosten erhöhen. Es kann durchaus der Fall sein, daß für verschiedene Standorte in einer Region auch verschiedene Lager als Versorger dienen, dies ist in der Kostenabschätzung ebenfalls zu berücksichtigen. Zum Schluß wird ein Bericht erstellt. Dieser enthält die ermittelten Ergebnisse, die der Geschäftsführung mitgeteilt werden, um sich nun für einen Standort zu entscheiden.

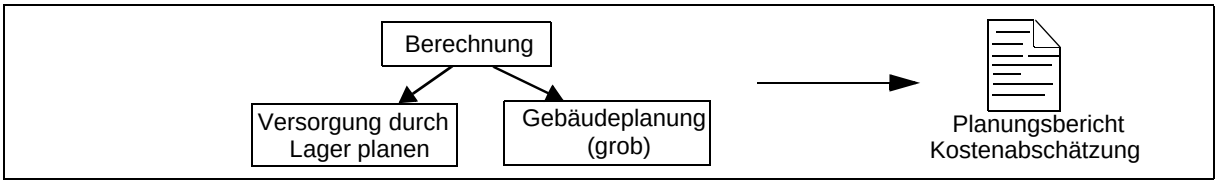


Abbildung 10: Aktivitäten für die Kostenabschätzung

Wie bereits angedeutet wurde, bestehen weitere Abhängigkeiten zwischen diesen umfassenden Aktivitäten. Eine solche Abhängigkeit ist die frühzeitige Nutzung der aktuellen bzw. vorläufigen Ergebnisse der Standortsuche durch die Geschäftsführung. Dadurch ist es möglich, schnell auf sich ändernde Bewertungen der Standorte einzugehen. Außerdem kann die Geschäftsführung eine Änderung der Liste direkt durchführen. Die Delegation allein hätte nur zur Folge, daß die Liste (d.h. das Ergebnis der delegierten Aktivität) an die Geschäftsführung erst weitergereicht wird, wenn die Standortsuche vorerst abgeschlossen ist. Durch die zusätzliche Verwendung der Nutzungsbeziehung entsteht jedoch eine wesentlich engere Kopplung. Dies ist wünschenswert, da die Standortsuche sehr langwierig sein kann und somit die Geschäftsführung frühzeitig eine Kostenabschätzung veranlassen kann.

Eine weitere Beziehung in der Standortplanung ist die Registrierung der Kostenabschätzung bei der Standortsuche, um möglichst früh neue Ergebnisse übernehmen zu können. Die Motivation ist ähnlich wie im obigen Fall, wodurch hier zusätzlich inkonsistente Daten weitgehend vermieden werden können. So sollte die Liste auch zur gemeinsamen Nutzung übernommen werden. Die entsprechende Aktivität ist in Abbildung 11 dargestellt.

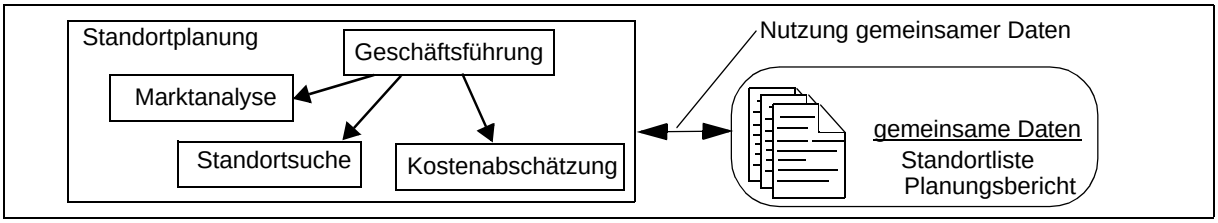


Abbildung 11: Erweiterte Beschreibung der Standortplanung

2.3.3.3 Bewertung

Anhand obigen Beispiels wurde verdeutlicht, wie wichtig der frühzeitige Austausch von Daten und abhängige Arbeitsschritte auch in der Planung sind. Zudem ist die Delegation von Arbeiten eine wichtige Modellierungstechnik, um abhängige Aktivitäten darzustellen, falls diese nicht sequentialisiert werden sollen. Es wurde zudem detailliert beschrieben, welche Überlegungen bei der Beschreibung eines solchen Arbeitsvorgangs zu beachten sind. Somit kann man sehen, daß hier ähnliche Anforderungen wie für Entwurfsprozesse bestehen.

2.4 Zusammenfassung der Szenarien

Für den Entwurf existieren Werkzeuge, die in den einzelnen Arbeitsschritten auch angewandt werden. Daher ist ein Ansatz für die Unterstützung von Entwurfsprozessen, diese Werkzeuganwendungen zu koordinieren. Im Bereich CAD existieren sog. CAD-Frameworks (etwa in [JCF 1991], Abschnitt 3.6), die hier auch die Verwaltung der Arbeitsergebnisse und Teilentwürfe unterstützen. Dort werden Technologien angewendet, wie z.B. Repositories und Versionierung, die auch in dem kommerziellen Produkt FPGA Advantage [MENTOR 2000] zu finden sind. Eine Erweiterung stellt die Kombination mit der (Workflow-ähnlichen) Regelung der Werkzeuganwendungen dar (vgl. [BOSCH 1995], [RYBA 1996]). Im CONCORD-Modell [RITTER 1997B] wurde diese Technologie um geregelte Kooperationen erweitert.

Wie am Beispiel der CAD-Unterstützung zu sehen ist, wurden bereits einige Aspekte einer weitgehenden Unterstützung von Entwurfsprozessen untersucht. Ähnliche Systeme existieren auch für andere CAx-Bereichen (bspw. CASE). Diese bieten jedoch nur Unterstützung für Teile der Anforderungen, es finden sich bspw. keine Systeme, die sowohl Gruppendiskussionen, Versionierung, Werkzeuganwendungseinbindung und Ablaufkontrolle durchgehend und einheitlich unterstützen.

Die Beschreibung der Szenarien hat unsere Vorstellung von Entwurfsprozessen und Prozessen mit ähnlichen Eigenschaften verdeutlicht. So können diese kooperativen Prozesse in Phasenmodellen beschrieben werden, wie z.B. im Falle des Software-Engineerings (siehe Abschnitt 2.3.2). Versucht man jedoch die einzelnen Phasen genauer zu beschreiben, so erreicht man schnell einen Punkt, ab welchem eine detaillierte Beschreibung einzelner Arbeitsschritte (z.B. Werkzeuganwendungen) und deren Abfolge nicht mehr möglich, vorhersehbar oder auch gar nicht erwünscht ist. Charakteristisch im Entwurf ist auch das 'trial and error'-Prinzip, d.h. ein Entwerfer stößt auf Probleme deren Lösung ihm nicht bekannt ist und probiert deswegen verschiedene Möglichkeiten zur Erreichung seines Ziels aus. Daher kann man einen Entwurfsprozeß als einmalig durchzuführenden Prozeß kennzeichnen, auch wenn Fragmente gleichbleibend sind. Aufgrund dieser Einmaligkeit, sind Entwurfsprozesse auch nicht den häufig wiederkehrenden Arbeiten zuzuordnen. Jedoch besitzen Entwurfsprozesse Teilprozesse, die durchaus eine gewisse Strukturiertheit innehaben, wie etwa die Phasen an sich oder bestimmte Folgen von Arbeitsschritten (z.B. Speichern des Quellcodes, Kompilation, Start der Kompilats in der Software-Entwicklung). Jedoch ist eine feste Strukturierung aller Arbeitsschritte kaum zu erreichen, da viele Entscheidungen ad hoc getroffen werden. Dies ist u.a. wegen der kreativen Abläufe notwendig, die den Entwerfern nicht fest vorgeschrieben werden können. Die Delegation ist ein weiteres Mittel eine gewisse Struktur des Vorganges aufzubauen. Wie in den Szenarien zu sehen ist, entspricht dies auch einem intuitiven Vorgehen, da die Zerlegung eines Problems (Entwurfsobjekt, Teilauftrag etc.), dessen Verteilung und schließlich dessen Integration dargestellt werden kann.

Ein weiterer wichtiger Aspekt wird durch die Kooperationsbedürfnisse beschrieben. Da in Entwurfsprozessen prinzipiell Abhängigkeiten zwischen Arbeitsschritten auftreten, stellen diese Konflikt- und Fehlerpotentiale dar. Da es nicht möglich scheint, eine genaue Definition der Abläufe vorzuplanen, und somit Konflikte im vorhinein ausschließen zu können, müssen Mechanismen für die Regelung solcher Konflikte und Probleme zur Laufzeit angeboten werden. Diese Konflikte basieren einerseits auf der gleichzeitigen Verwendung von gemeinsamen Betriebsmitteln. Andererseits sind Ergebnisse, die in anderen Arbeitsschritten gerade erarbeitet werden oder auch wurden, zunächst nicht in anderen Arbeitsschritten bekannt. Dies führt zu Problemen, wenn etwa Ergebnisse integriert werden müssen. Der Konflikt tritt also erst spät auf. Um dies zu vermeiden bzw. einzuschränken ist es notwendig, frühzeitig Daten auszutauschen und/oder sich über die Arbeit anderer bewußt zu sein.

Aufgrund der Semistrukturierung der von uns untersuchten kooperativen Prozesse, ist eine vollständige Regelung nicht möglich. Um beim Beispiel eines Software-Projekts zu bleiben, kann man die Häufigkeit der Erstellung eines spezifischen Softwareprodukts als einmalig betrachten. Teilprozesse mögen, wie oben schon angedeutet, jedoch häufiger ausgeführt werden. Zusätzlich sind kooperative Mechanismen notwendig um Konflikte frühzeitig zu erkennen und zu vermeiden. Dadurch ergibt sich eine ungefähre Einteilung von kooperativen Prozessen in die fünf Dimensionen Häufigkeit, Regelung, Strukturierung, Bedeutung und Kooperationsbedarf, wie durch die schraffierten Bereiche in Abbildung 12 dargestellt[1]. In dieser Abbildung wird auch eine Charakterisierung administrativer Vorgänge, wie zum Beispiel die Bearbeitung einer Dienstreiseabrechnung, den Entwurfsprozessen gegenübergestellt. Auffallend sind besonders die Diskrepanzen bezüglich Kooperation, Häufigkeit und Strukturierung. Da diese Merkmale besonders voneinander abweichen, bestätigt sich die Auffassung, daß hier auch andere Technologien einzusetzen sind, als für administrative Abläufe.

1. Die Grafik wurde ohne die Anwendung von Metriken zur Messung der einzelnen Dimensionen erstellt und orientiert sich an [TEUFEL ET AL. 1995] und [SHETH ET AL. 1996].

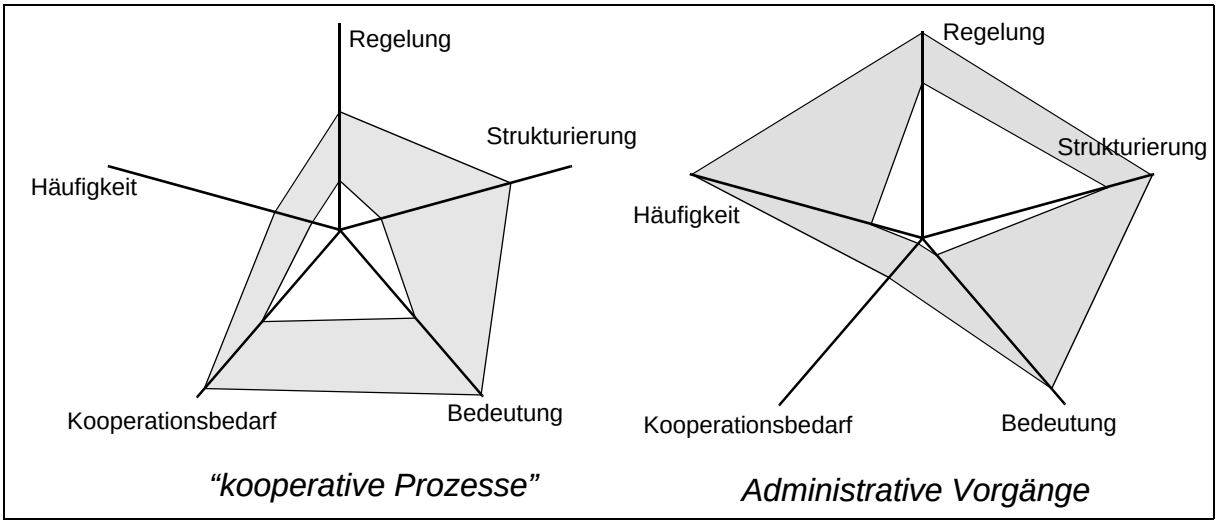


Abbildung 12: Charakterisierung von Entwurfsprozessen und administrativen Vorgängen

2.5 Folgerungen für die Aufgabenstellung

Ziel dieser Arbeit ist es, eine umfassende Systemunterstützung für Prozesse zu erreichen, wie sie in den vorherigen Abschnitten beschrieben wurden. Wir beschränken uns hier nicht auf Entwurfsprozesse, sondern beziehen kooperative Prozesse mit ähnlichen Eigenschaften in die Aufgabenstellung ein. Durch die im vorangegangen Abschnitt beschriebene Charakterisierung dieser betrachteten Prozesse wird deutlich, daß wir als Systemunterstützung kooperative und regelnde Technologien benötigen, die uns idealerweise Flexibilität, wohl-strukturierte Teilprozess-Steuerung und Kooperationsmechanismen erlauben.

Hier wird somit nicht das alleinige 'Vorantreiben' von Aktivitäten gewünscht, sondern zusätzliche Aufmerksamkeit der direkten Unterstützung der Tätigkeiten eines Mitarbeiters gewidmet. Anders formuliert: nicht die korrekte Abfolge von Arbeitsschritten ist das Ziel unseres Systems, sondern die geeignete Unterstützung zur Lösung einer Aufgabe zu bieten. D.h. wir zielen auf eine sog. *User Assistance* ab, anstatt einer Vorgehensbeschreibung. Somit soll der Nutzer des Systems in der Lage bleiben, die Arbeit auch nach seinen Entscheidungen voranzutreiben. Wir benötigen also:

- Ablaufunterstützung, welche die einzelnen Arbeitsschritte vorantreibt und damit den Gesamtablauf beherrschbar macht,
- Delegationsunterstützung, die sich auf das Aktivitätsmodell und das Entwurfsobjektmodell auswirkt, da auf der einen Seite die Abläufe organisiert werden müssen und auf der anderen Seite ein entsprechendes Datenmodell zur Modellierung der Zerlegung eines Entwurfsobjekts bestehen muß,
- Regelung gleichzeitiger Arbeiten, wie zum Beispiel gemeinsame Datennutzung und der frühzeitige Austausch von Arbeitsergebnisse,
- direkte Kommunikations- und Kooperationsunterstützung für die Abstimmung der beteiligten Personen und
- Integrative Bestandteile zur Einbindung von existierenden Systemen und Werkzeugen.

Im folgenden Kapitel werden die Technologien, die zur Realisierung unserer Anforderungen von Bedeutung sind, eingehend besprochen. Diese sind die Grundlage eines umfassenden Systemmodells, welches grundlegend für diese Arbeit sein wird.

3 Technologien für die Arbeitsunterstützung

Dieses Kapitel stellt Technologien für die Unterstützung verschiedener Arbeitsformen vor. Am Ende werden wir skizzieren, wie eine geeignete Kombination dieser Technologien für die Realisierung der Aufgabenstellung herangezogen werden kann.

3.1 Überblick

Im vorhergehenden Kapitel wurden verschiedene Arbeitsformen charakterisiert und anhand einiger Beispielprozesse die Anforderungen für derartige kooperative Prozesse herausgearbeitet. In diesem Kapitel werden grundlegende Technologien und Mechanismen beschrieben, die für eine Realisierung der Aufgabenstellung von Bedeutung sind. Als Grundlage für die folgenden Abschnitte führen wir zuerst einige Grundlagen der verteilten Systeme ein (Abschnitt 3.2). Wir streben eine Kombination von Groupware- und Workflow-Technologie an. Diese Technologien besprechen wir in den Abschnitten 3.3 und 3.4. Für die Kooperationsprotokolle bieten Multi-Agentensysteme eine mächtige Protokollunterstützung (siehe Abschnitt 3.5). CAD-Frameworks (Abschnitt 3.6) sind für Entwurfsprozesse aufgrund ihres flexiblen und integrativen Charakters besonders geeignet, auch wenn die kooperativen Aspekte ungenügend berücksichtigt werden. In Abschnitt 3.7 beschreiben wir kurz weitere hilfreiche Ansätze, die der Umsetzung unserer Anforderungen nützlich sind. Zuletzt stellen wir den Lösungsansatz der vorliegenden Arbeit in Abschnitt 3.9 vor.

3.2 Verteilte Systeme

Da wir Middleware für die Umsetzung unserer Anforderung zur Integration verschiedener Systeme benötigen, wollen wir hier einige grundlegende Aspekte besprechen. CSCW- und Workflow-Systeme sind Untergruppen der verteilten Systeme, da sie ihre Dienste in verteilten Umgebungen anbieten. Daher beschreiben wir hier kurz wesentliche Aspekte und Dienste für die Entwicklung verteilter Systeme.

3.2.1 Send/Receive

Für die Kommunikation von Programmen in verteilten Umgebungen existiert der Nachrichtenaustausch mit *send* und *receive*. Dabei dient *send* dem Senden einer Nachricht an einen (oder mehrere) Empfänger. *Receive* ist das Gegenstück, das den Empfang einer gesendeten Nachricht ermöglicht.

Man kann mehrere Arten der send/receive-Funktionalität unterscheiden (Abbildung 13):

- synchrone und asynchrone Kommunikation
- Ausführungssemantik: exactly once, at least once etc.
- unidirektional und bidirektional.

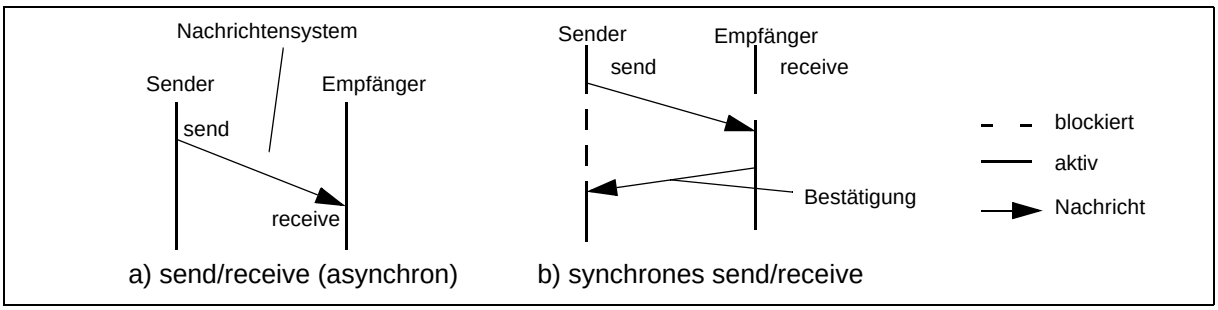


Abbildung 13: send/receive Kommunikation zwischen Betriebssystemprozessen

Mit diesen Primitiven lassen sich alle Kommunikationsstrukturen in Netzumgebungen aufbauen, wie z. B. Client-Server-Kommunikation.

3.2.2 RPC und CORBA

Ein Mechanismus zur Interprozeßkommunikation ist der *Remote Procedure Call* (RPC), der in etwa einem synchronen send/receive entspricht. Der Aufruf einer anderen Funktion via RPC erfolgt analog einem gewöhnlichen Funktionsaufruf, wobei dessen Ausführung auf einem anderen Rechner geschehen kann. Dabei werden plattformabhängige Techniken zur Parameterübergabe und zur Semantik entsprechend umgesetzt.

Vereinfacht entspricht CORBA (*Common Request Broker Architecture* [OMG 1996]) einem objektorientierten RPC. In einem CORBA-System werden Methoden aufgerufen, die bestimmte Objekte als Service anbieten. Die Lokalisierung von Objekten und die Durchführung von Aufrufen übernimmt dabei der ORB (*Object Request Broker*, siehe Abbildung 14), der als zentraler Dienst über die beteiligten Rechner hinweg verfügbar sein muß.

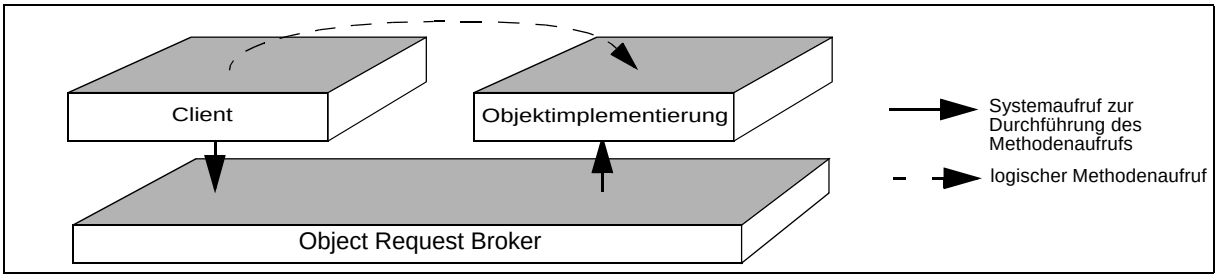


Abbildung 14: Methodenaufruf mit CORBA

3.2.3 Verteilte Dateisysteme

In verteilten Systemen, speziell in lokalen Netzwerken, wird häufig ein verteiltes Dateisystem eingesetzt, wie zum Beispiel das *Network File System*. Die Vorteile eines verteilten Dateisystems sind unter anderem die Ortstransparenz, d.h. der Zugriff auf eine Datei von verschiedenen Rechnern aus wird wie ein lokaler Zugriff behandelt. Zusätzlich werden benutzerspezifische Sperren verwaltet, um den konkurrierenden Zugriff zu koordinieren.

3.2.4 Zusammenfassung

In den vorgestellten Techniken, die den Aufbau von verteilten System ermöglichen, sehen wir bereits einige Anwendungsmöglichkeiten. CORBA erlaubt es vorhandene System zu kapseln und anderen CORBA-Objekten zur Verfügung zu stellen, ist also für die Intergration verschiedener Systeme in Betracht zu ziehen. Die send/receive-Semantik verdeutlicht, welche Arten der Basiskommunikation zwischen verschiedenen Entitäten eines verteilten System möglich sind und lassen uns die Interaktion der geplanten Systembestandteile definieren. Verteilte Dateisysteme bieten die Möglichkeit auf einfache Art bereits gewisse Kooperationsformen auf gemeinsamen Daten zu erreichen. Somit geben uns diese Techniken erste Mittel an die Hand, die von uns gewünschte Integration und erste Kooperationsunterstützung zu erreichen.

3.3 CSCW

Die zunehmende Vernetzung von Arbeitsplätzen und Firmen über die ganze Welt hinweg eröffnet die Möglichkeit über die Rechnergrenzen hinaus mit anderen Personen zusammenzuarbeiten. Der Forschungsbereich CSCW (*Computer Supported Co-operative Work*) beschäftigt sich

mit der Untersuchung kooperativer Arbeit. Für solche Arbeiten werden Gruppen von Personen gebildet, die vor allem eine gemeinsame Aufgabe teilen. Mittels sogenannter Groupware soll es diesen Gruppen ermöglicht werden, kooperativ an einer Aufgabe zu arbeiten.

Der Begriff CSCW wurde in der Mitte der 80er-Jahre im Rahmen des Workshops im Endicott House, Massachusetts, von Greif und Cashman eingeführt [BORGHOFF UND SCHLICHTER 1995]. Applikationen, die CSCW-Techniken verwenden, werden als *Groupware* bezeichnet. Kommerzielle Beispiele hierfür sind Lotus Notes oder Microsoft Exchange. Aber unter CSCW-Applikationen sind allgemein wesentlich vielfältigere Anwendungen zu verstehen.

Im folgenden werden die zentralen Begriffe aus dem Forschungsbereich CSCW definiert und einige Beispielanwendung zur Verdeutlichung vorgestellt. Danach werden grundlegende Techniken aus den verteilten Systemen erläutert, welche die Basis von CSCW-Anwendungen bilden, ein Überblick über die Applikationen gegeben und die wichtigsten Mechanismen in CSCW untersucht. Schließlich werden die Merkmale von CSCW-Anwendungen zusammengefaßt.

3.3.1 Grundlagen im Bereich CSCW

In diesem Abschnitt werden die im folgenden verwendeten Grundbegriffe besprochen. Eine umfassende Übersicht der in CSCW verwendeten Begriffe und Funktionalitäten findet sich in [BORGHOFF UND SCHLICHTER 1995].

CSCW

Als generelle und kurze Definition von CSCW wählen wir folgende:

Definition *CSCW*
"CSCW is a generic term which combines the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and techniques." [WILSON 1991]

Wie in dieser Definition zu sehen ist, spielt das Gruppenkonzept eine wesentliche Rolle. Durch eine Gruppe wird eine Menge von Leuten gebildet, die aus irgendeiner Motivation heraus zusammenarbeiten wollen. Dies soll letztendlich durch eine entsprechende Infrastruktur ermöglicht werden. Deutschsprachige Publikationen verwenden häufig auch den Ausdruck Team anstatt Gruppe, um den kollaborativen Aspekt hervorzuheben.

Groupware

Groupware ist zur Zeit ein häufig gebrauchtes und ein ebenso viele Facetten umfassendes Schlagwort. Dies geht besonders aus nachstehender Begriffsdefinition hervor:

Definition *Groupware*
"Groupware is a generic term for specialized computer aids that are designed for the use of collaborative work groups. Typically, these groups are small project-oriented teams that have important tasks and tight deadlines. Groupware can involve software, hardware, services and/or group process support." [JOHANSEN 1991]

Basierend auf dieser Definition können wir Groupware als die technische Realisierung von CSCW-Aspekten ansehen.

Gruppe und Group Awareness

Wie schon zu sehen war ist der Begriff Gruppe zentral für CSCW. Für das Anliegen kooperatives Arbeiten bzw. Kollaboration zu unterstützen, stellt eine Gruppe die Basis für die Menge von Bearbeitern dar, die sich zur Lösung gemeinsamer Probleme zusammenschließen. Die Gruppe ist eine Umgebung, die den beteiligten Personen die benötigten Ressourcen und Mechanismen zur Verfügung stellt. Daher definieren wir:

Definition Gruppe
Eine Gruppe ist eine gemeinsame Umgebung für eine Menge von Personen mit einer gemeinsamen Aufgabenstellung. Diesen Personen werden benötigte Ressourcen (etwa Kommunikationskanäle, Dokumente etc.) und Mechanismen (Verhandlungsprotokolle etc.) bereitgestellt.

In der Definition wird die Wichtigkeit einer gemeinsamen Umgebung und Aufgabe hervorgehoben. Dies sind die wesentlichen Merkmale, die CSCW-Systeme von anderen Systemen unterscheiden.

Ein wesentlicher Aspekt einer Gruppe ist die sogenannte *Group Awareness*, die durch ein psychologisches Existenzbewußtsein [BORGHOFF UND SCHLICHTER 1995] eine Kooperation erleichtert[1]. Dadurch, daß sich die Gruppenmitglieder ihrer gegenseitigen Existenz und auch Aktionen bewußt sind, können sie entsprechend auf die anderen Mitglieder reagieren. Dies erlaubt die Möglichkeit zur Koordination in einer Gruppe, die nicht explizit durch das System geschieht, sondern durch die beteiligten Personen und deren Group Awareness erreicht wird.

Kooperation und Kommunikation

Kooperation tritt als Wort bereits in CSCW auf. Bei einer Kooperation ist es wichtig, daß persönliche Ziele gegenüber den Zielen einer Gruppe zweitrangig werden [BORGHOFF UND SCHLICHTER 1995]. D. h. letztendlich trifft die Gruppe gemeinsame Entscheidungen, wie ihre Ziele zu erreichen sind. Man könnte die Kooperation auch eine durch gemeinsame Entscheidungen und gemeinschaftliches Handeln erreichte Koordination nennen. Die Grundlage für Kooperation ist eine entsprechende Kommunikation und Interaktion (vgl. Abbildung 15a).

Diese Kommunikation kann man in *systembezogen* (oder technisch) und *sozialbezogen* unterteilen. Die systembezogene Kommunikation stellt die technischen Mittel zur Verfügung, um die Kommunikation zu ermöglichen. Die sozialbezogene Kommunikation beinhaltet die Fähigkeit der Gruppenteilnehmer zur Kommunikation und die existierenden sozialen Protokolle. Ein soziales Protokoll beschreibt die Regeln für die Vorgehensweise in der Kommunikation und für Entscheidungen zwischen Personen(-gruppen), z. B. nennt man im Usenet das soziale Protokoll ‘Netiquette’. Bindeglied ist ein technisches/soziales Protokoll, das die Verwendung des Kommunikationsdienstes des Systems unter Berücksichtigung der sozialen Aspekte beschreibt (Abbildung 15b).

3.3.2 Einteilung von CSCW-Systemen

In [SCHILL 1996] wird eine feingranulare Klassifikation für CSCW-Systeme aufgeführt, die folgende Kriterien aufweist:

- Koordinationsunterstützung: niedrig, mittel, hoch
- Koordinationsart: lenkend, kontrollierend
- Teilnehmer: Mensch, Interaktionswerkzeuge, Server

1. Es ist kein äquivalentes deutsches Wort für Group Awareness bekannt. Im folgenden wird gelegentlich “Gruppenbewußtsein” verwendet, auch wenn dessen Bedeutung nicht so umfassend ist.

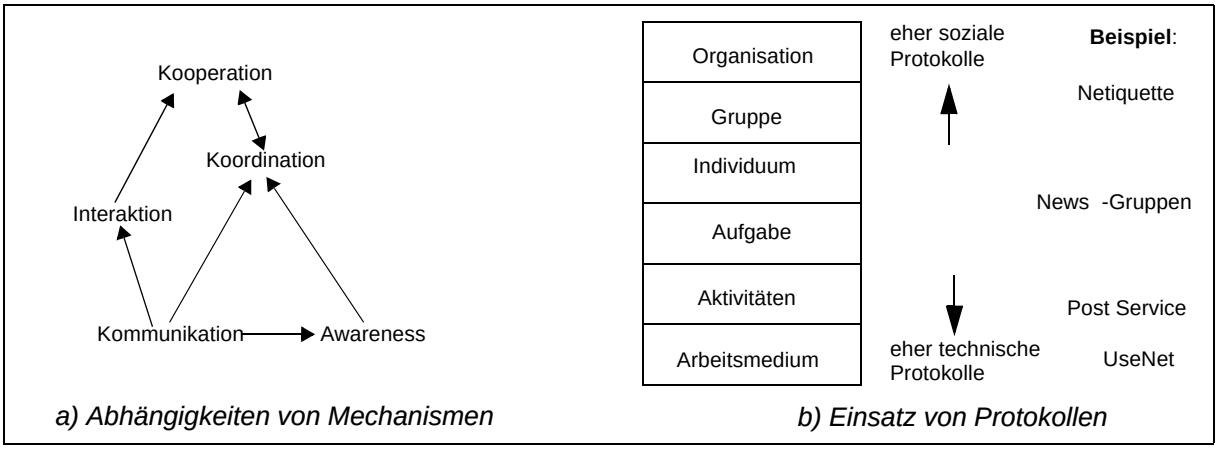


Abbildung 15: Grundlegende Aspekte und Protokolle in CSCW-Umgebungen

- Daten: Daten, Multimedia, Dokumente
- Aufgabenstruktur: hoch, schwach, unstrukturiert
- Zeitbedingungen: synchron, asynchron
- Zeitrahmen: lang, mittel, kurz
- Verteilung: keine, LAN, WAN
- Koordinationsstruktur: zentral, dezentral
- Interaktionsart: implizit, explizit

Die meisten Aspekte werden in diesem Kapitel angesprochen. Für diese Arbeit werden jedoch Verteilung und Daten nicht besonders betrachtet, da hier unseres Erachtens vor allem technische Rahmenbedingungen, wie die Datenübertragungsrate, ausschlagegebend für eine Klassifikation sind und somit für die allgemeine Funktionsweise von zweitrangiger Bedeutung sind.

Für die Klassifizierung von CSCW-Systemen wird oft auch die Form der Zeit/Ort-Matrix gewählt [GRUDIN 1994]. Auf der einen Seite wird hierbei zwischen asynchronen und synchronen Arbeiten unterschieden und auf der anderen zwischen örtlich getrennt bzw. zusammenfallend. Daß diese Aufteilung einige Mängel besitzt wurde bereits in [SCHMIDT UND RODDEN 1996] gezeigt und deswegen entsprechend erweitert, z. B. in [TEUFEL ET AL. 1995] (siehe Abbildung 16) und [SCHLICHTER ET AL. 1997]. Eine Klassifizierung anhand der Aspekte von Ort und Zeit bietet einige Vorteile, jedoch werden wir für unsere Aufgaben CSCW-Systeme bezüglich ihrer Funktionsweise in drei Klassen aufteilen, für die wir nachfolgend einige typische Anwendungen beschreiben:

- Nachrichtensysteme,
- Konferenzsysteme,
- Gruppenbasierter Entwurf.

Diese Aufteilung wurde gewählt, da hier schwerpunktmäßig die Grundfunktionen von CSCW-Anwendungen untersucht werden und daher eine Aufteilung nach Funktionalität sinnvoller ist. Die zu den oben definierten CSCW-Klassen zugehörigen Systeme sind in Abbildung 16 über die Zeit-Ort-Matrix eingetragen. Zum Beispiel umfassen die Nachrichtensysteme, die aus den Bulletin-Board-Systemen (BBS), Elektronischen Postsystemen und Workflow-Management-Systemen (WFMS) bestehen, mehrere Sektoren der dargestellten Matrix. Die Konferenzsysteme werden sowohl durch die Video-/Desktopkonferenzen als auch die Sitzungs- und Entscheidungsunterstützungssysteme repräsentiert. Der gruppenbasierte Entwurf ist mit Gruppeneditoren vertreten. Spezialisierte Datenbanken, spezialisierte Planungssysteme und verteilte Hypertext-Systeme sind je nach ihrem Einsatzgebiet den Gruppen zuzuordnen. So kön-

nen beispielsweise bestimmte Planungssysteme für den gruppenbasierten Entwurf zur Ablaufplanung genutzt werden oder andere als Nachrichtensystem zur Überwachung des Projektfortschritts.

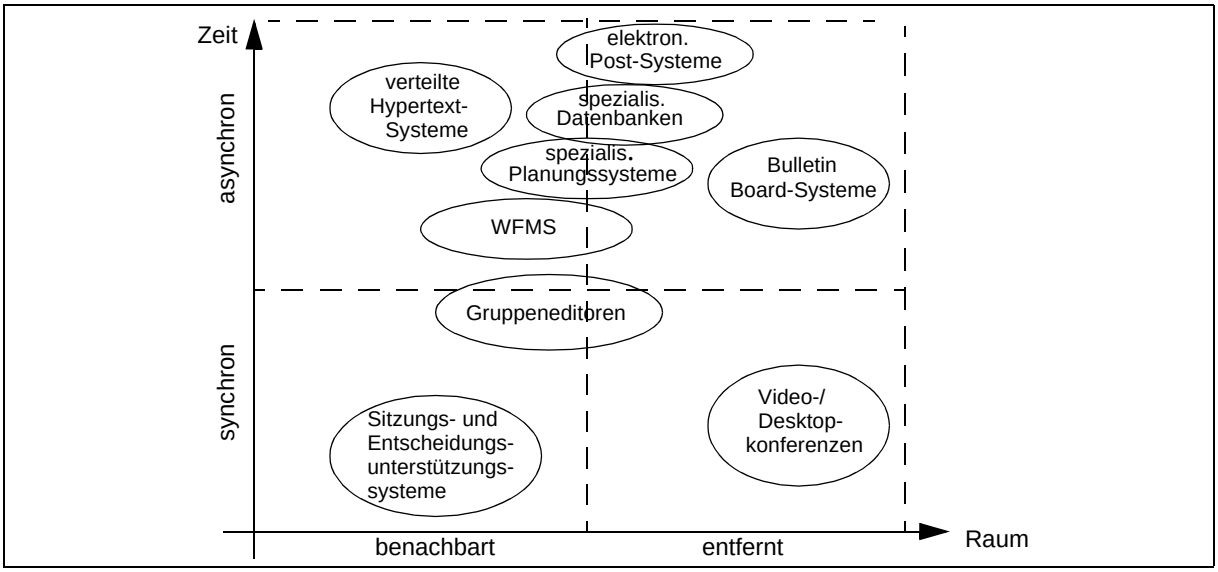


Abbildung 16: Zeit/Ort-Matrix nach [TEUFEL ET AL. 1995]

3.3.3 Beispielanwendungen

Da wir CSCW-Systeme bisher eher abstrakt behandelt haben, werden hier Beispielanwendungen vorgestellt. An der Bandbreite dieser Systeme kann man die weitreichende Anwendbarkeit von CSCW erahnen.

3.3.3.1 E-Mail

Die technisch wohl verbreitetste Art, um kooperative Arbeit zu ermöglichen, basiert auf dem Versenden elektronischer Nachrichten (z.B. E-Mail). Dadurch können einfache Nachrichten für die direkte Kommunikation zweier Parteien verschickt, oder auch Dokumente im Sinne einer Umlaufmappe weitergeleitet werden.

Es können Gruppen oder Einzelpersonen adressiert werden, die eine Nachricht erhalten sollen. E-Mail erlaubt es grundsätzlich asynchron und örtlich verteilt Nachrichten auszutauschen und ist heutzutage sehr stark verbreitet, nicht zuletzt wegen der Normierung des Nachrichtentransfers (X.400-Spezifikation, [TANENBAUM 1992]).

Um als CSCW-System zu gelten muß eine Gruppe ein soziales Protokoll entwickeln, das den Umgang mit E-Mail regelt, um so eine Kooperation zu ereichen.

3.3.3.2 Telekonferenz

Im Bereich der Telekonferenzen kann man zwischen verschiedenen Systemen, wie Tele-Teaching, *face-to-face* Sitzungen, verteilte elektronische Sitzungen unterscheiden. Hier werden die synchronen, aber örtlich verteilten elektronischen Sitzungen kurz vorgestellt, sog. Online-Konferenzen.

Die Entwicklung von Telekonferenz-Systemen wurde stark forciert (z.B. [STREITZ ET AL. 1994]). Personen, die über ein Thema diskutieren wollen, können dies trotz örtlicher Verteilung tun und dabei mit den Gesprächspartner(n) durch Audio- und Videoübertragung konferieren.

Deswegen können in örtlich verteilten Organisationen (oder auch zwischen verschiedenen Organisationen) kurzfristig anstehende Probleme besprochen werden, und es sind somit keine aufwendigen Reisen vorzunehmen.

Bei einer solchen Konferenz kann jeder Teilnehmer in seiner eigenen Umgebung, bzw. an seinem Arbeitsplatz, bleiben, wobei er aber über eine Audio/Video-Verbindung über ein gemeinsames Netzwerk, z.B. das Internet, mit den anderen Konferenzteilnehmern verbunden ist. Dabei kommt oft ein gemeinsamer *Workspace* (siehe Abschnitt 3.3.5) zum Einsatz, der es den verschiedenen Teilnehmern erlaubt die Ergebnisse der Konferenz für alle zusammenzufassen oder Präsentationen durchzuführen.

3.3.3.3 Gruppeneditor, Group Authoring

Größere Dokumente, wie zum Beispiel Studien oder Fachbücher, werden oft von mehreren Personen erstellt. Da die Autoren auf gemeinsame Teile des Dokumentes zugreifen wollen, kann dies leicht zu Inkonsistenzen führen. Um eine möglichst konfliktfreie Arbeitsweise anzubieten, muß entweder strikt geregelt werden, wie die einzelnen Arbeiten zu trennen sind (z.B. bei Versionierung), oder aber es wird eine kooperative Schnittstelle angeboten, mit der es den teilnehmenden Autoren möglich ist, gleichzeitig auf dem selben Dokument zu arbeiten.

Wenn die verschiedenen Parteien online die Änderungen an einem Dokument verfolgen, können sie wesentlich zügiger darauf reagieren, anstatt den geänderten Teil erst nach einiger Zeit allen zukommen zu lassen. Da in einem Gruppeneditor auch die aktuellen Änderungsstellen gesehen werden können (siehe Abbildung 17), etwa wie in [RITTER 1997] oder [KOCH 1995] beschrieben, werden “doppelte” oder inkonsistente Arbeiten vermieden. Das Prinzip wird kurz WYSIWIS (*What You See Is What I See*) genannt. Ähnliche Anforderungen existieren auch für andere Entwurfsumgebungen. Allgemein wird dieses Konzept unter dem Begriff *Group Authoring* zusammengefaßt.

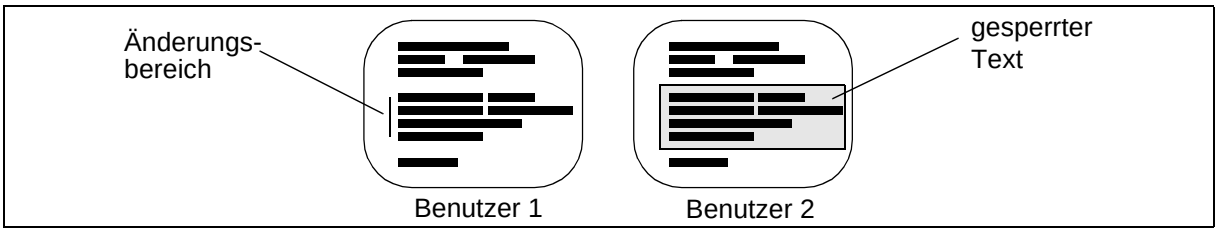


Abbildung 17: Textsperre in einem Gruppeneditor

3.3.3.4 Workflow-Management-Systeme

Workflow-Management-Systeme (WFMS) werden detaillierter in Abschnitt 3.4 besprochen. Wir geben hier lediglich einen kurzen Überblick um deren Einordnung im Bereich CSCW und ihre Eigenschaften aufzuzeigen.

WFMS unterstützen die Implementierung von Geschäftsprozessen in Unternehmen. Dabei steht die Steuerung des Ablaufs und nicht die Unterstützung der Ausführung von Arbeit im Vordergrund. Ein WFMS übernimmt die Zuteilung von Arbeiten (*Work Items*) an die betreffenden Personen anhand einer Workflow-Definition. Diese Workflow-Definition wird vorab entworfen und beschreibt die Daten zur Organisation (z. B. Rollen) und Informationen zu den zu verwendenden Werkzeugen und Arbeitsdaten (siehe Abbildung 18).

Wird ein Workflow gestartet, so wird die Prozeßdefinition instantiiert und das WFMS übernimmt die Verwaltung der Vorgänge. Aktivitäten, die zur Bearbeitung anstehen, werden als Work Items in den Arbeitslisten den möglichen Bearbeitern angeboten. Ein Teilnehmer des Workflows qualifiziert sich durch seine Rolle für bestimmte Arbeiten. Bei der Annahme eines

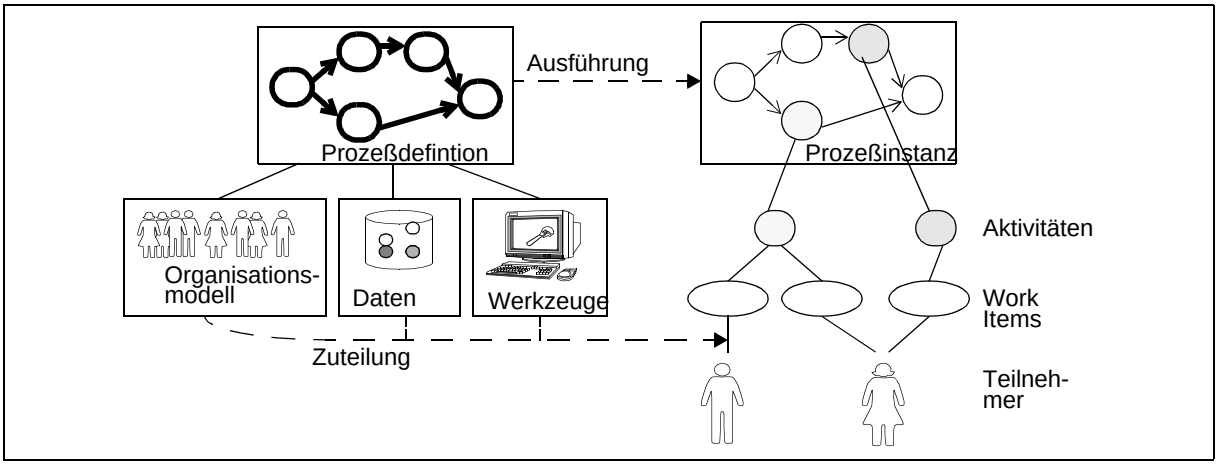


Abbildung 18: Durchführung von Workflows

Work Items werden der ausführenden Person, soweit möglich, die benötigten Daten und Applikationen bzw. Werkzeuge zur Verfügung gestellt. Falls Aktivitäten durch Maschinen durchgeführt werden können, werden sie diesen zugeführt und bearbeitet.

Wie zu sehen ist, ist der Kommunikationsaufwand zwischen den Workflow-Teilnehmern gering, die Koordination wird hauptsächlich durch das WFMS geleistet. Manche Workflow-Systeme bieten auch den Einsatz von sogenannten Ad-Hoc-Workflows an, wodurch eine "spontane" Änderung der Prozeßdefinition zur Laufzeit möglich ist. Somit wird eine größere Eingriffsmöglichkeit in den Ablauf erreicht.

3.3.4 Kommunikation und Koordination

CSCW-Systeme benötigen spezielle verteilte Dienste für die Realisierung von Kommunikation, Protokollen zur Koordination und gruppenbezogenen Aspekten. Die unterscheidenden Merkmale zu anderen verteilten System sind die Fokussierung auf Gruppen und die Unterstützung für kooperatives Handeln. Um die Kooperation zwischen Gruppenmitgliedern in CSCW-Anwendungen zu unterstützen sind entsprechende Kommunikations- und Koordinationsmechanismen zu verwenden. Dabei zeigen wir, daß diese Mechanismen auf verschiedenen Ebenen notwendig sind.

3.3.4.1 Kommunikation

Die oben vorgestellten Grundlagen der verteilten Systeme ermöglichen die Kommunikation in einer verteilten Umgebung. Wie bereits in Abschnitt 3.3.1 angesprochen, stellt die Kommunikation eine der grundlegendsten Anforderungen (siehe Abbildung 15a auf Seite 40) in CSCW-Systemen dar.

Das Spektrum von sozialbezogener Kommunikation bis zur systembezogenen Kommunikation hat das Ziel die Kooperation bestmöglich zu erreichen. Die systembezogene Kommunikation hat dabei die technischen Kommunikationsmechanismen zwischen den Gruppenmitgliedern zur Verfügung zu stellen. Die Systemkommunikation verwendet z. B. die in Abschnitt 3.2 angesprochenen Mechanismen, so daß die einzelnen Teile des gesamten CSCW-Systems innerhalb der Netzumgebung kommunizieren können (vgl. Abbildung 19). Die sozialbezogene Kommunikation ist der Austausch von Informationen zwischen Personen(gruppen) auf Gruppenebene. Das CSCW-System liefert die höherwertigen Kommunikationsmechanismen für diese (z.B. Gruppeneditor). Auf den technischen Ebenen fließen die Daten, die auf der abstrakten Gruppenebene für die Kooperation sorgen. Wie bereits erwähnt, kann auch ein System bereits Koordinationsmechanismen realisieren.

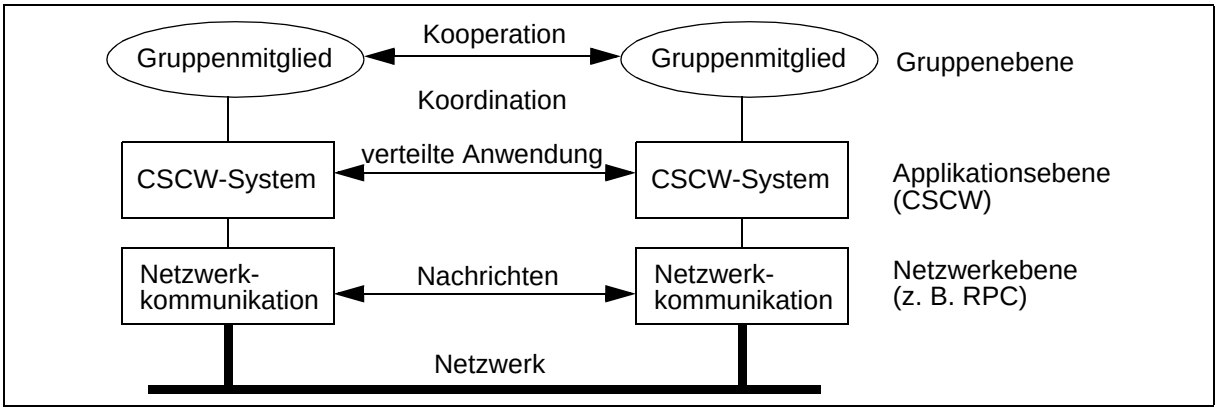


Abbildung 19: Kommunikationsschichten in CSCW-Systemen

Eine weitere Aufteilung der Kommunikation ist direkte und indirekte Kommunikation (vgl. auch [SCHLICHTER ET AL. 1997]). Die direkte Kommunikation erfolgt direkt und zielgerichtet durch ein Gruppenmitglied mit anderen, bspw. per E-Mail. Die indirekte Kommunikation wird typischerweise durch gemeinsame Ressourcen erreicht (siehe Abbildung 20). Wird etwa ein Datum exklusiv gesperrt, darf nur die Person darauf zugreifen, welche die entsprechende Sperre hält. Dadurch weiß der Auslöser eines fehlgeschlagenen Zugriffs (Benutzung der gesperrten Ressource), daß offensichtlich ein anderer dort eine Sperre hält. Man kann dabei unterscheiden, ob es sich um ein passives System handelt, d. h. erst beim versuchten Zugriff erkennt man die Sperre, oder ob es sich um ein aktives System handelt, d. h. die Sperre ist einem bereits bekannt, wie bei dem WYSIWIS-Paradigma (siehe 3.3.3.3). Der Raum der gemeinsamen Ressourcen wird auch *Workspace* genannt (vgl. Abschnitt 3.3.5).

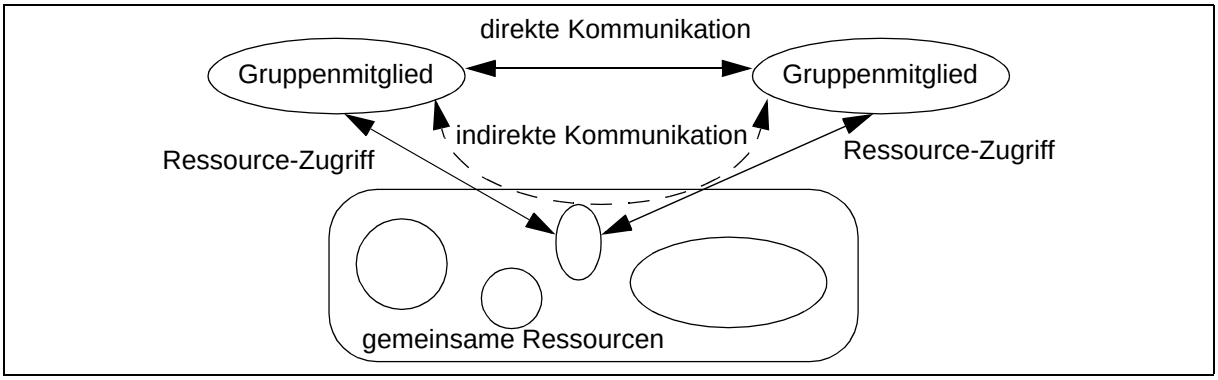


Abbildung 20: Direkte und indirekte Kommunikation

Indirekte Kommunikation ermöglicht grundsätzlich nur eine einfachere Form der Kooperation als die direkte Kommunikation [SCHLICHTER ET AL. 1997], da die Interaktion geringer ist. Das für eine ausgeprägte Kooperation notwendige gemeinsame Wissen ist nur schwer über indirekte Mechanismen, wie z.B. ein *Blackboard*, zu erreichen, da direkte Mechanismen oft eine höhere Awareness mit sich bingen (z.B. Telekonferenz). Prinzipiell erfolgt die direkte Kommunikation synchron und die indirekte Kommunikation asynchron. Gegenbeispiele sind jedoch E-Mail und teilweise auch WYSIWIS-Editoren.

Gegen die direkte Kooperation spricht, der abnehmende Grad der systemgestützten automatischen Koordination. Bei ACID-Transaktionen [HÄRDER UND REUTER 1983] wird die Konsistenz der zugrundeliegenden Daten durch das System gesichert. Dies ist bei einer direkten Kooperation nicht unbedingt gewährleistet, da die Entscheidung nicht beim unterstützenden System liegt.

Für die Herstellung von Awareness auf gemeinsamen Ressourcen eignet sich etwa ein Notifikationsmechanismus, um über die sich ändernden Zustände von gemeinsamen Ressourcen benachrichtigt zu werden. Dies kann aber sowohl nach einem *Pull*-Prinzip (aktives Selbstabholen) oder ein *Push*-Prinzip (automatische Benachrichtigung) geschehen.

3.3.4.2 Koordination

Die Kommunikationsmechanismen erlauben die Koordination der einzelnen Gruppenmitgliedern und ihrer Aufgaben. Ein einfaches Beispiel ist die indirekte Kommunikation über gemeinsame Ressourcen, die auch zu einer sofortigen Koordination führt, das heißt ein Konflikt auf diesen Ressourcen wird vermieden und somit koordiniert. Dies ist jedoch im allgemeinen nicht ausreichend, da weitere Anforderungen bestehen können. So kann es bspw. sinnvoll sein, daß zwei Personen zur gleichen Zeit ein Dokument bearbeiten können, oder daß eine gesperrte Ressource einer anderen Person zugesprochen werden sollte. Für derartige Anforderungen sind weitere Mechanismen notwendig. Eine Möglichkeit bietet sich zum Beispiel durch direkte Kommunikation, z. B. via E-Mail oder ein Telefongespräch, damit sich die betroffenen Gruppenmitglieder koordinieren können (nicht durch das System erzwungen).

Eine weitere Ebene der Koordination ist die Steuerung von Arbeitsschritten. Dies kann bereits indirekt durch die eben beschrieben Mechanismen erreicht werden. Eine direkte Koordination, also etwa durch ein unterstützendes System, greift aktiver und meist nach bestimmten Regeln in den Ablauf der Arbeiten ein. Entscheidungen, die bei einer Koordination getroffen werden können, umfassen laut [BURGER 1997]:

- Bestimmen und Aufteilen des gesamten Lösungsweges in Einzelschritte,
- Zuordnung der einzelnen Tätigkeiten zu Teamteilnehmern (oder Rollen),
- zeitliche Ordnung von Tätigkeiten,
- Zusammenführen von Ergebnissen.

Koordination kann man in ein Spektrum von ad hoc bis vorgeplant einordnen (in [SCHLICHTER ET AL. 1997] wird dies implizit und explizit genannt). Am einen Ende stehen unstrukturierte Aufgaben deren Koordination nur während der Erledigung geregelt wird, wie zum Beispiel das gemeinsame Schreiben einer Veröffentlichung[1]. Am anderen Ende stehen stark strukturierte Vorgänge, deren Erledigung im voraus planbar ist, wie zum Beispiel die Abwicklung eines Bankkredits, der festen Regeln zu folgen hat (siehe auch Workflow-Management in Abschnitt Abschnitt 3.4).

Wie man in Abbildung 21 sieht, bedeutet eine vorgeplante Koordination auch oft einen geringeren Bedarf an direkter Kommunikation (und umgekehrt), da viele Aktionen durch ein System vorgegeben werden können [SCHILL 1996] und nicht erst kurzfristig festgelegt werden müssen. Bei unstrukturierten Aufgaben besteht ein hoher Bedarf an Kommunikation, da die Gruppenmitglieder sich ad hoc koordinieren müssen. Ein System, welches ad hoc Entscheidungen unterstützt, stellt dabei nur die Möglichkeit für die Abstimmung zur Verfügung, ohne aktiv einzugreifen. D. h. mit der Aktivität der Gruppenmitglieder steigt häufig auch die Passivität eines CSCW-Systems und umgekehrt. Um so mehr eine ad-hoc-Koordination notwendig ist, desto wichtiger ist also die Funktion der Group Awareness [SCHLICHTER ET AL. 1997]. Typisch für unstrukturiertere Arbeiten ist die Unterstützung durch Kommunikationssysteme. Strukturiertere Arbeiten können hingegen bspw. durch Workflow- und Prozeßautomatisierungssysteme unterstützt werden (siehe auch Abschnitt 2.2).

1. Wir gehen hier von der Praxis aus. Theoretisch läßt sich der Ablauf auch grob vorab definieren.

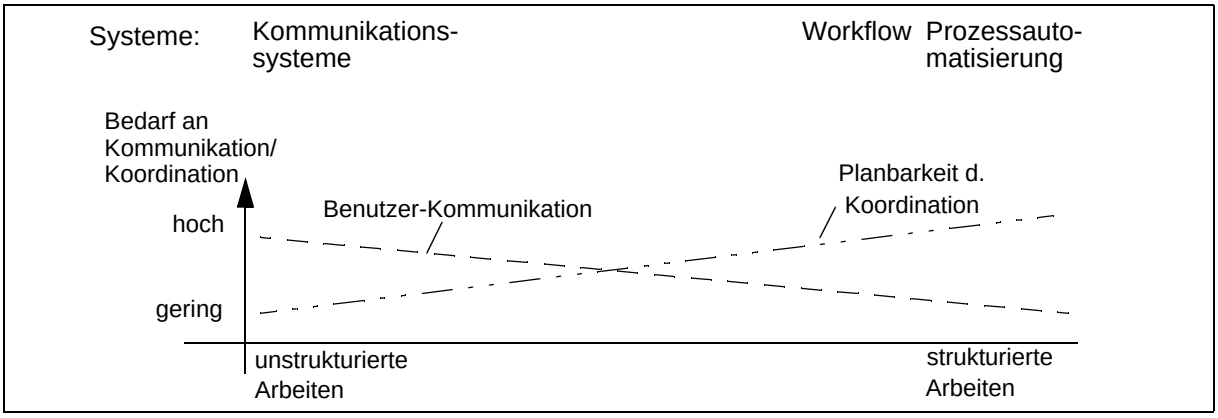


Abbildung 21: Spektrum an Koordination in CSCW-Systemen

3.3.5 Informationsraum und Group Awareness

Ein wesentlicher Punkt bezüglich der Kooperation ist wie oben schon genannt die Group Awareness. Ein *Workspace* umfaßt alle Ressourcen (Daten, Programme, Personen, Systemkomponenten, ... [MUCHITSCH 1998]), die zur Durchführung einer Aufgabe notwendig sind. Ein *Informationsraum* hingegen stellt nur Datenobjekte zur Verfügung und regelt deren Verwendung. Die Group Awareness kann über einen gemeinsamen Workspace oder einen Informationsraum und entsprechenden Notifikationstechniken realisiert werden.

Besonders im Bereich von schwach oder unstrukturierten Arbeiten besteht ein Bedarf an Koordination zwischen den Gruppenmitgliedern (siehe im vorherigen Abschnitt). Durch das gegenseitige Existenzbewußtsein und das Wissen über die Erwartungen und Arbeiten anderer Gruppenteilnehmer (auch *shared understanding* in [THOMAS 1995] genannt) kann diese Koordination leichter erreicht werden, da Konflikte besser vermieden werden können. Zur Erbringung dieser Group Awareness dient z. B. auch die Verwendung indirekter Kommunikation, die über einen gemeinsamen Informationsraum abgewickelt wird. Im Falle der direkten Kommunikation erreicht man natürlich eine besonders hohe Group Awareness, jedoch kann diese in der Regel nicht dauernd aufrechterhalten werden, da die Kommunikationsteilnehmer auch ihre Aufgaben erledigen müssen.

3.3.6 Der Gruppenprozeß

Die Dynamik innerhalb einer Gruppe wird in ihrer Gesamtheit Gruppenprozeß genannt. Man kann zwischen synchronen und asynchronen Phasen unterscheiden [BORGHOFF UND SCHLICHTER 1995]. Die synchronen Phasen dienen der Koordination der Gruppenmitglieder. Das heißt es werden Arbeiten abgeglichen und ein weiteres Vorgehen geplant. In den asynchronen Phasen arbeiten die Gruppenmitglieder eigenständig an ihren Aufgaben. Somit findet in einem Gruppenprozeß ein andauernder Wechsel mit synchronen und asynchronen Phasen statt, wobei die Synchronisierung nicht unbedingt für die gesamte Gruppe gelten muß, sondern manchmal nur für eine Untergruppe.

In Phase 1 der Abbildung 22 findet eine allgemeine Besprechung zur Festlegung der zu erreichenden Ziele statt. Dies kann zum Beispiel im Rahmen einer Telekonferenz (vgl. 3.3.3.2) geschehen. Danach arbeiten die verschiedenen Gruppenmitglieder in Phase 2 an den gestellten Aufgaben. In Phase 3 findet eine Synchronisation zwischen einigen Gruppenmitgliedern statt, um ihre bisherige Arbeit abzustimmen und die weitere Arbeit gegebenenfalls neu festzulegen. Danach (Phase 4) arbeiten die einzelnen Personen wieder an ihren Aufgaben. Schließlich wird in Phase 5 das Ergebnis der Gruppenarbeit ausgewertet.

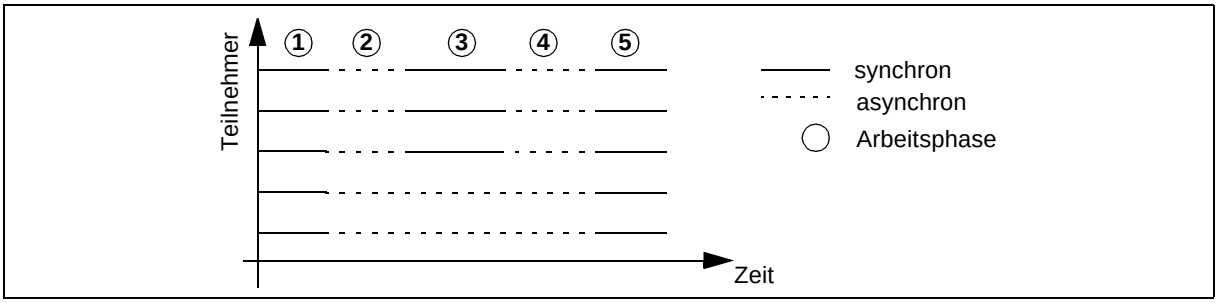


Abbildung 22: Ablauf eines Gruppenprozesses (vgl. [BORGHOFF UND SCHLICHTER 1995]*)*

3.3.7 Zusammenfassung grundlegender CSCW-Aspekte

Da die Spannweite der möglichen Anwendungen und der dafür notwendigen Techniken sehr groß ist, kann man für CSCW nicht einfach eine bestimmte Menge von Primitiven finden. Diese wären sehr allgemein (z.B. send/receive) oder sehr anwendungsspezifisch. Daher bietet es sich an, eine Einteilung nach Arten von CSCW-Anwendungen zu wählen, die eine ähnliche Menge von Basisfunktionen verwenden.

Bei der Untersuchung konnten die folgenden funktionalen Bereiche für CSCW-Anwendungen ermittelt werden:

- **Gruppen und Rollen**
  Fast alle Anwendungen bieten Gruppen und/oder Rollen an. Diese helfen, die Anwender zu strukturieren. Die Anwender werden entsprechend ihrer Aufgabe und/oder Verantwortlichkeit eingeteilt. Ein weiterer Aspekt bei der Gruppierung ist die Verwaltung von Ressourcen. Bei TeamRooms bzw. dessen kommerziellem Ableger TeamWave [WINDRIVER 2001] können z.B. verschiedene Personen auch verschiedene Workspaces nutzen und jeweils davon abhängige Rechte besitzen.
- **Kommunikation**
  Die Kommunikation ist in vielen CSCW-Anwendungen stark ausgeprägt. Die Unterscheidung liegt hier zum einen in den Mechanismen (indirekt/direkt und synchron/asynchron) und zum anderen in der Häufigkeit. Beispielsweise treten in Workflow-Management-Systemen kaum direkte Kommunikationen auf.
- **Gemeinsamer Informationsraum**
  Ein wichtiges Merkmal von CSCW-Anwendungen ist die Verwendung eines gemeinsamen Informationsraumes [TEEGE 1996]. Dieser beinhaltet gemeinsam genutzte Daten und die Möglichkeit zur indirekten Kommunikation. Innerhalb einer oder mehrerer Gruppen werden gemeinsame Informationen verwaltet. Entsprechend ist eine Kommunikation über die gemeinsamen Informationen notwendig, die direkt oder indirekt erfolgen kann (vgl. auch 3.3.4.1). Eine einfache Realisierung eines gemeinsamen Informationsraums stellt z.B. ein gemeinsames Dateisystem dar.
- **Group Awareness**
  Das gegenseitige Existenzbewußtsein ermöglicht Gruppen kooperativ zu arbeiten. Die Kenntnis welche Personen Teil einer Gruppe und auch momentan verfügbar sind (z. B. bei TeamRooms), erlaubt die Kommunikation mit diesen. Da Abhängigkeiten, sowohl bezüglich von Arbeitsschritten und gemeinsamen Ressourcen, den betroffenen Personen bekannt sind, können sie mit Rücksicht auf die anderen Personen handeln und somit kooperieren.

Wie in den vorhergehenden Abschnitten dargelegt wurde, basiert CSCW-Technologie auf den drei Prinzipien der Kommunikation, Koordination und Kooperation (siehe Abbildung 23a). Gleichzeitig lassen sich auch CSCW-Anwendungen hinsichtlich dieser Themen einordnen [TEUFEL ET AL. 1995] (siehe Abbildung 23b).

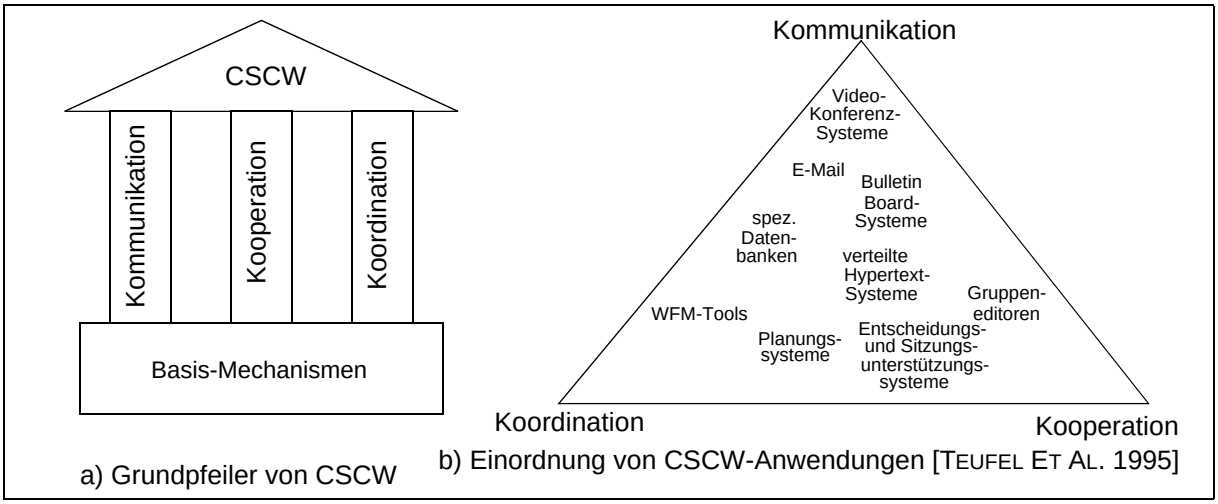


Abbildung 23: CSCW-Grundkonzepte und Einordnung von Applikationen

Wie zu sehen ist, sind CSCW-Anwendungen im Spektrum der drei Grundkonzepte (Kommunikation, Kooperation und Koordination) breit gefächert. Die prinzipielle Verknüpfung der Grundfunktionalität (siehe Abbildung 15a, Seite 39) ist mehr oder weniger stark ausgeprägt in allen Anwendungen vorhanden. So werden die zuvor ermittelten Mechanismen nicht in allen CSCW-Anwendungen in gleicher Art und Weise bzw. manchmal auch gar nicht eingesetzt.

Zur Realisierung allgemeiner CSCW-Systeme ist also eine weitgehende Vielfalt an Funktionalität notwendig. Dabei ist aus allen in 3.3.2 aufgeführten Bereichen eine Basis an Primitiven erforderlich, da Funktionen eines Bereiches oft von Funktionen anderer Bereiche abhängen: bspw. ist zur Lokalisierung eines Kommunikationspartners die Gruppenverwaltung zu befragen.

Von besonderem Interesse für die vorliegende Arbeit ist eine Kopplung von allgemeiner CSCW-Funktionalität und Workflow-Technologie. Deswegen wurde hier auch kurz Workflow-Management als Teil der CSCW-Forschung diskutiert. Dabei wurde festgestellt, daß es sich bei WFMS vor allem um Koordinationssysteme handelt, die nur wenige kooperative und kommunikative Anteile enthalten. Diese Aspekte sollen nun mit den oben vorgestellten Primitiven kombiniert werden. Im folgenden werden wir nun detailliert auf Workflow-Management-Systeme eingehen.

3.4 Workflow-Management

Workflow-Technologie spielt eine wichtige Rolle in der vorliegenden Arbeit, da sie die teilautomatisierbaren Aspekte der von uns betrachteten Prozesse unterstützen kann. Besonders für unser Aktivitätenmodell sind die hier erläuterten Grundlagen wichtig. Daß Workflow-Systeme auch immer stärker in Richtung Kooperation und Kommunikation eindringen, wird am Ende dieses Abschnitt deutlich. Dies ist von besonderer Bedeutung bzgl. der von uns geforderten Flexibilität und Kooperationsfähigkeit für Entwurfsprozesse.

Zunächst beschreiben wir die Anforderungen an Workflow-Management-Systeme (WFMS) und definieren die wichtigsten Begriffe aus diesem Bereich. Nachfolgend wird eine allgemeine Architektur für WFMS vorgestellt. Wir werden WFMS abgrenzen und einordnen. Zum Schluß werden Workflow-Arten für verschiedene Arten von Arbeitsformen beschrieben.

3.4.1 Grundlagen für Workflow-Management

In Unternehmen spielt die effiziente Organisation und Durchführung der relevanten betrieblichen Abläufe eine sehr bedeutende Rolle. Daher wird eine weitgehende Unterstützung durch Informationssysteme angestrebt, die eine optimale und auch zuverlässige Durchführung ermöglichen sollen. Ein Workflow-Management-System hat somit einen wesentlichen Teil eines Unternehmens darzustellen. Infolgedessen werden hohe Anforderungen an WFMS gestellt. Zudem besteht ein wesentlicher Vorteil eines WFMS in den schnellen Kommunikationswegen, die eine rasche Abwicklung voneinander abhängiger Arbeiten erlauben; wo früher der "Papiertransport" das Arbeitstempo wesentlich bestimmte, kann dieser Faktor fast vollkommen vernachlässigt werden.

Um betriebliche Abläufe rechnergestützt durchzuführen oder weitgehend zu automatisieren, ist es nötig, diese Abläufe, die auch Geschäftsprozesse genannt werden, zu analysieren und zu formalisieren. Dieser Schritt wird häufig im Rahmen des *Business Process Re-engineering* durchgeführt [SPURR ET AL. 1994] und dient der effektiven und effizienten Gestaltung der zugrundeliegenden Geschäftsprozesse. Dadurch kann eine bessere Wettbewerbsfähigkeit eines Unternehmen erreicht werden.

Die formale Spezifikation eines solchen Prozesses kann durch ein WFMS implementiert werden und so die gewünschte Steuerung automatisiert werden. Um der zentralen Bedeutung gerecht zu werden, muß ein WFMS bestimmten Bedingungen genügen:

- *Durchführbarkeit von Abläufen (Effektivität) und Effizienz*
  Es muß die Durchführbarkeit von Arbeitsläufen garantiert werden, wobei diese effizient bezüglich der Ressourcen wie Zeit, Betriebsmitteln und Mitarbeiter sind.
- *Zuverlässigkeit*
  Es soll ein reibungsloser und sicherer Betrieb innerhalb einer Organisation stattfinden. So sind zum Beispiel längere Ausfallzeiten oder der Verlust von unternehmens- und prozeßrelevanten Daten nicht akzeptabel.
- *Integration von Legacy-Systemen*
  Eine weitere Bedingung die an WFMS gestellt wird, besteht in der Integration bereits existierender Software-Systeme (z.B. Standardanwendungen) [JABLONSKI 1995]. Dies ist erforderlich, da alle benötigten Funktionalitäten eines Prozesses nicht allein durch ein WFMS unterstützt werden können. Außerdem sind Firmen noch oft im Besitz von Know-How und Daten ihrer bisherigen Informationssysteme, die eine erhebliche Investition darstellen. Weiter können für den Geschäftsbetrieb andere Systeme notwendig sein, wie zum Beispiel Produktionssteuerung, deren Integration in ein WFMS vorteilhaft ist.
- *Skalierbarkeit und Anpaßbarkeit*
  Um auch zukünftigen Anforderungen gewachsen zu sein, sollten WFMS skalierbar und an neue Hardware-, Software- und betriebliche Gegebenheiten anpaßbar sein.

Um grundlegende Eigenschaften von Workflow-Systemen zu besprechen, bietet es sich an, die Definitionen der Workflow Management Coalition (kurz: WfMC [WFMC 1995]) zu verwenden. Diese Organisation ist ein Zusammenschluß vieler Unternehmen, die im Workflow-Bereich tätig sind und sich um die Standardisierung von Workflow-Produkten, speziell im Hinblick auf Interoperabilität, bemühen. Da hier verschiedene Interessen berücksichtigt werden, sind die Vorschläge der WfMC teils als Kompromisse zu sehen und nicht so weitgehend, wie es möglich wäre (z.B. durch Erfahrung und Forschungsresultate).

Definition *Geschäftsprozeß*
Ein Geschäftsprozeß stellt eine Menge von logisch abhängigen Verfahren oder Aktivitäten dar, die in ihrer Gesamtheit ein Ziel oder eine Strategie innerhalb eines Un-

ternehmens verfolgen. Dieser Prozeß steht gewöhnlich dabei in einem direkten Organisationskontext, der aufgabenbezogene Rollen und Beziehungen enthält.

Diese Definition besagt, daß innerhalb eines solchen zielgerichteten Prozesses Abläufe unter Einbeziehung von Verantwortlichkeiten und Abhängigkeiten enthalten sind. Ein Beispiel hierfür ist die Abwicklung eines Kreditantrags. Dabei ist zu beachten, daß diese Definition keine klare Aussage über die Begriffe Aktivität oder Verfahren enthält. Im weiteren Verlauf dieses Berichts wird zu sehen sein, daß dem Wort Aktivität im Zusammenhang mit Workflow-Management eine besondere Bedeutung zukommt.

Definition *Workflow*
Ein Workflow ist eine ganze oder teilweise Automatisierung eines Geschäftsprozesses, innerhalb dessen Dokumente, Informationen oder Aufgaben von einem Prozeßteilnehmer zu einem folgenden, entsprechend bestimmter Verfahrensregeln, für die Bearbeitung weitergegeben werden.

Im Vergleich mit den in Abschnitt 2.2 besprochenen Arbeitsformen dienen Workflows also vor allem der Beschreibung definierbarer und damit automatisierbarer Prozesse. Eine explizite Kooperationsunterstützung wird nicht verlangt, vielmehr werden die “gemeinsamen” Daten nach festen Regeln sequentialisert zur Bearbeitung weitergeleitet. Zur Implementierung (“Automatisierung”) eines Geschäftsprozesses, also die Realisierung eines Workflows, sind vier Aspekte offensichtlich von großer Bedeutung:

1. Funktionaler Aspekt (*was* wird ausgeführt),
   der hier durch die (Teil-)Automatisierung eines Geschäftsprozesses mittels Verfahrensregeln repräsentiert wird.
2. Informationsaspekt (*welche* Informationen sind erforderlich),
   der in obiger Definition die Weitergabe von Dokumenten, Information und Aufgaben umfaßt.
3. Organisatorischer Aspekt (*wer* führt die Arbeit aus),
   durch den zum Beispiel die Prozeßteilnehmer erfaßt werden. Die Ablauforganisation beinhaltet die Organisationsstruktur und die Beziehungen zwischen organisatorischen Einheiten.
4. Kausalität (*warum* wird ein Workflow spezifiziert)
   Dies beinhaltet die Aussagen über die allgemeinen rechtlichen und unternehmensspezifischen Grundlagen eines Workflows, also die oben genannte logischen Abhängigkeiten. Dies sind Regeln für den Ablauf, wie zum Beispiel: die Voraussetzung einer Dienstreise ist der Auftrag, falls dieser jedoch frühzeitig widerrufen wird, soll diese Reise nicht durchgeführt werden. Die kausalen Beziehungen sind durch das WFMS zu überwachen und durchzusetzen, etwa durch die Auswertung von Vorbedingungen von Aktivitäten (s.u.).

In [JABLONSKI 1995] finden sich zudem drei weitere Aspekte, die zur Charakterisierung von WFMS dienen. Dies sind das Verhalten (Kontrollfluß), die Historie (*logging*) und der transaktionale Aspekt (Datensicherheit, *recovery*).

Definition *Workflow-Management-System*
Ein System, das Workflows zu definieren, verwalten und mittels weiterer Softwaresysteme, etwa Werkzeugen und Anwendungen, durchzuführen erlaubt, wird Workflow-Management-System genannt. Dieses (ggf. verteilte) System kann aus einer oder mehreren Workflow-Engines bestehen, die Prozeßdefinitionen interpretieren, mit den Workflow-Teilnehmern und -Anwendungen interagieren und falls nötig entsprechende IT-Werkzeuge und -Anwendungen starten.

Wie man aus der Definition ersieht, obliegt einem WFMS die Definition, Steuerung und Ausführung von Workflows. Um einen Workflow durchzuführen ist eine Verwaltung der Teilnehmer (organisatorischer Aspekt) und das Starten von (externen) Anwendungen nötig. Der Datenfluß oder eine weitgehende Steuerung der externen Anwendungen ist laut dieser Definition nicht explizit Bestandteil eines WFMS.

Definition Prozeßdefinition

Eine Prozeßdefinition umfaßt die Darstellung eines Geschäftsprozesses in einer Form, die automatisierte Manipulationen wie Modellierung oder Durchführung mittels eines WFMS zuläßt. Die Prozeßdefinition besteht aus einem Netzwerk von Aktivitäten und ihren gegenseitigen Beziehungen, Vor- und Nachbedingungen des Prozesses und Informationen über die individuellen Aktivitäten, so wie Teilnehmer, Applikationen, Daten usw.

Eine Prozeßdefinition umfaßt eine weitgehende Beschreibung des zu implementierenden Prozesses. Wie aus der Definition von WFMS ersichtlich wird, werden nicht alle diese Aspekte durch das WFMS verwaltet, sondern nur eine Untermenge, die wir hier *Workflow-Definition* nennen. Informationen über Daten und allgemeine Aktivitäten bspw. werden nicht unbedingt durch das WFMS verwaltet bzw. finden sich in anderen Spezifikationen wieder (vgl. Aktivitäten-Definition unten).

Definition Aktivität

Eine Aktivität ist die Beschreibung einer (Teil-)Arbeit, die einen logischen Schritt innerhalb eines Prozesses darstellt. Eine Aktivität kann sowohl interaktiv, d. h. nicht durch einen Rechner automatisiert, oder eine automatisierte Workflow-Aktivität sein. Eine interaktive Workflow-Aktivität benötigt menschliche Unterstützung, um die Prozeßdurchführung zu erlauben. Eine solche Workflow-Aktivität wird einem Workflow-Teilnehmer zugewiesen. Eine automatisierte Workflow-Aktivität wird ohne menschlichen Eingriff durch das Workflow-System und andere Software-Systeme durchgeführt.

Eine Workflow-Aktivität stellt so einen Schritt innerhalb eines Workflows dar, der letztendlich diesen Prozeß näher an sein Ziel bringen soll. Dabei kann dies vollautomatisch geschehen, wie zum Beispiel der Ausdruck eines Kundenbriefes, oder gesteuert, wie das Schreiben eines Kundenbriefes mit einer Textverarbeitung durch einen Workflow-Teilnehmer. Eine rein manuelle Aktivität, die nicht durch Software-Systeme unterstützt wird (bspw. die Aufgabe eines Briefes bei der Post), aber dennoch Teil eines Geschäftsprozesses ist, wird im folgenden auch als manuelle Workflow-Aktivität bezeichnet. Im weiteren wird bei eindeutiger Sachlage mit Aktivität stets die interaktive Workflow-Aktivität gemeint sein.

Diese Aktivitäten stellen einzelne Bearbeitungsvorgänge innerhalb eines Workflows dar. Die (logische) Verknüpfung dieser Aktivitäten entspricht der in der Prozeßbeschreibung. Zu beachten ist hierbei, daß ein solcher Schritt bezüglich der Prozeßdefinition eine atomare Einheit ist [WFMC 1995], die von einem WFMS durchgeführt wird.

Definition Prozeß-Instanz

Die Prozeß-Instanz ist die Repräsentation einer einzelnen Ausführung eines Prozesses einschließlich der Instanzdaten. Jede Instanz stellt eine eigene Ausführungseinheit des Prozesses dar, die für gewöhnlich zu einer unabhängigen Steuerung und eines Audit während des Verlaufes fähig ist.

Die Prozeß-Instanz ist somit die Grundlage für einen konkreten Prozeß, der auf seiner entsprechenden Prozeßdefinition beruht. Verschiedene Instanzen einer Prozeßdefinition sind voneinander unabhängig. Instanzen des Prozesses *Kreditvergabe* können etwa die *Kreditvergabe an Frau Schneider* oder die *Kreditvergabe an die Firma IBM* sein.

3.4.2 Vom Arbeitsablauf zur Workflow-Definition

Wie erhält man nun eine Workflow-Definition? Das prinzipielle Vorgehen besteht aus der Analyse der zu unterstützenden betrieblichen Abläufe und deren formalen Definition für eine Übertragung in eine für das WFMS interpretierbare Form (vgl. Abbildung 24). Die Analysephase (Schritt A) kann im Sinne des Business Process Re-Engineering (BPR) stattfinden. Jedoch ist die Aufgabe des BPR nicht nur als eine für das Workflow-Management-System vorbereitete Prozeßbeschreibung zu sehen, sondern allgemeiner. Das BPR dient dem Verstehen der Vorgänge eines Unternehmens und kann zu deren Optimierung [TAUDES ET AL. 1996] angewandt werden [SPURR ET AL. 1994]. Die Geschäftsprozeßbeschreibungen können Dank ihres formalen Charakters (Schritt B) relativ problemlos in Workflow-Definitionen überführt werden (Schritt C), wie zum Beispiel bei INCOME [JAESCHKE 1996].

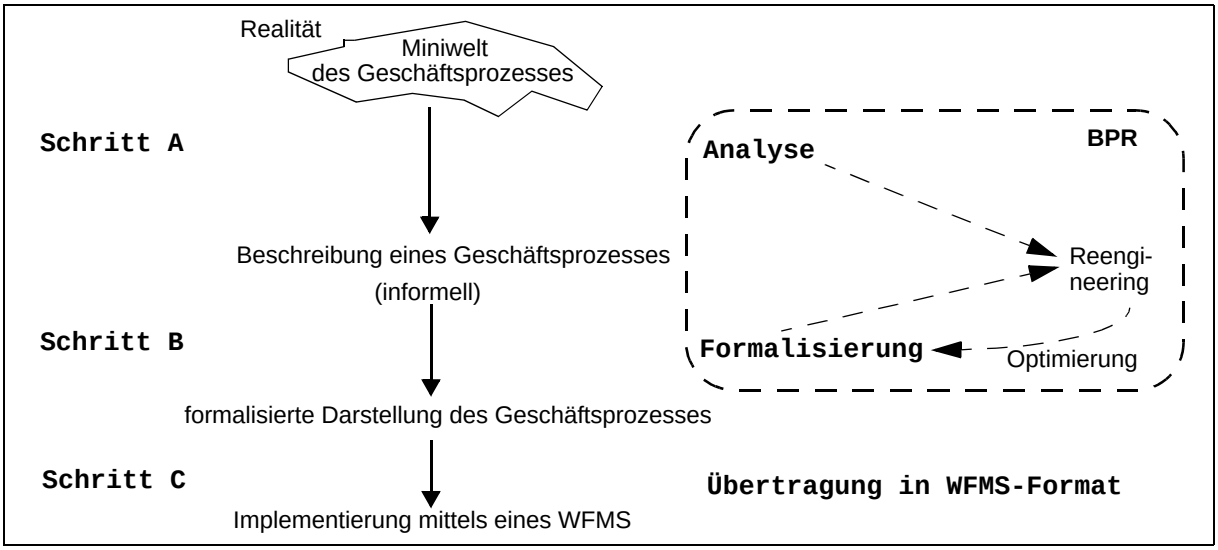


Abbildung 24: Vorgehen zur Implementierung von Geschäftsprozessen

In der Analysephase werden die zentralen Prozesse und deren Abhängigkeiten ermittelt. Diese Vorgänge sind gegeneinander abzugrenzen und deren wesentliche Charakteristika zu bestimmten [JAESCHKE 1996]. Daraus erhält man Informationen, wie diese Prozesse koordiniert werden, welche Informationen ausgetauscht werden müssen und inwieweit sie sich rechnergestützt realisieren lassen. Letztendlich sollte daraus eine Liste entstehen, welche die Prozesse nach dem Wert der Aufgabenstellung darstellt. Darauf folgt eine Analyse, die feststellen soll, welche Prozesse formal darstellbar sind, wie häufig diese auftreten und welcher Art sie sind (z. B. unstrukturiert, zyklisch usw.).

Aus dieser Liste werden die zu implementierenden Geschäftsprozesse extrahiert, analysiert und schließlich modelliert. Diese Modellierung soll die Gesamtstruktur eines Geschäftsprozesses zum Ziel haben, das heißt es werden Aktivitäten, Kontrollfluß, Informationsfluß, Datenablage und Organisationsstruktur aufgezeigt (vgl. Abbildung 25). Die Organisationsstruktur ermöglicht eine geordnete Zuordnung von Bearbeitern zu Aktivitäten. Die Prozeßstruktur beschreibt den Ablauf der Aktivitäten (Kontrollfluß) und den dazugehörigen Informationsfluß. Zur Definition des Informationsflusses ist es notwendig, ein entsprechendes Datenmodell und eine Datenverwaltung bereitzustellen.

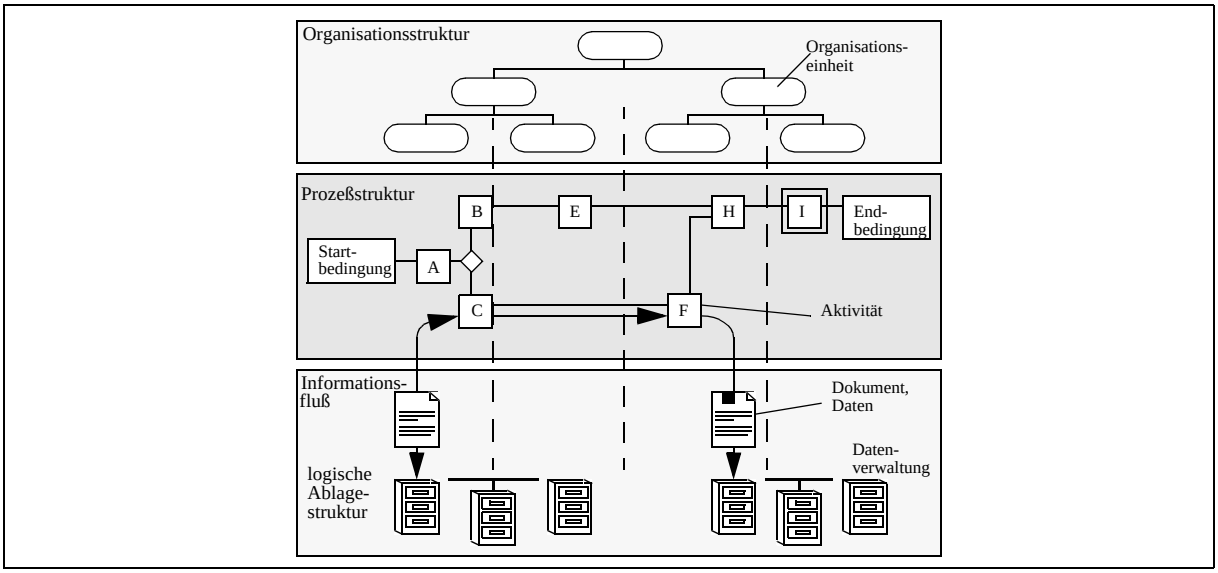


Abbildung 25: Aspekte in der Modellierung von Büroprozessen (nach [RATHGEB 1994]*)*

Falls möglich wird der Geschäftsprozeß optimiert. Dies kann der Fall sein, wenn Engpässe offensichtlich werden, wie etwa ein Konflikt auf Ressourcen. Zur Implementierung eines so bestimmten Prozesses mit einem Workflow-System sind alle Ebenen bis auf die logische Ablagestruktur wichtig. Die logische Ablagestruktur kann mit Hilfe anderer Mittel modelliert werden (z.B. E-R-Modell). Die Stellenstruktur findet in einem Organisations- oder Rollenmodell ihren Niederschlag, die von dem WFMS verwendet wird (siehe auch 3.4.4.2). Die Prozeßstruktur und der Informationsfluß werden, in eine Prozeßdefinition überführt. Aus der Beschreibung des Geschäftsprozesses und den weiteren Analyse-Ergebnissen kann nun letztendlich die Prozeßdefinition und Organisationsmodellierung gewonnen werden. Weitere Faktoren, die beschrieben werden müssen, sind z. B. die zu verwendenden Applikationen, Regeln und Ressourcen.

Die grafische Formalisierung eines (Teil-)Geschäftsprozesses in Abbildung 26 ist an das Modellierungswerkzeug ARIS-Toolset [ARIS 1996] angelehnt. Die Rechtecke stellen Arbeitsschritte in einem Prozeß dar, die Rauten symbolisieren Ereignisse und die Ellipsoide sind die verantwortlichen Organisationseinheiten für einen Arbeitsvorgang. Desweiteren existieren für die Steuerung des Flusses Verknüpfungen wie *UND*, *EXKLUSIVES ODER* usw.

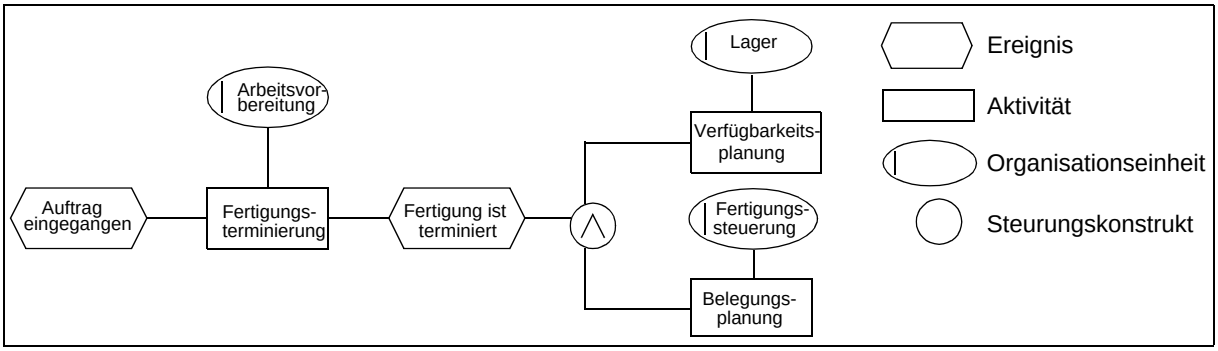


Abbildung 26: Grafisch formalisierter Geschäftsprozeß in ARIS

3.4.3 Modellierung und Ausführung von Workflows

Im folgenden wird die Modellierung von Workflows ausführlicher beschrieben, da aufgrund der gewünschten (Teil-)Automatisierung von Entwurfsprozessen eine Aktivitätenbeschreibung für diese Arbeit von großer Bedeutung ist.

3.4.3.1 Elemente einer Workflow-Beschreibung

Wie zuvor beschrieben findet eine Übertragung der Prozeßbeschreibung in eine Workflow-Definition statt. Es müssen zudem verwendete Ressourcen, externe Applikationen, Unternehmensstruktur etc. dem System bekannt gemacht werden. Glücklicherweise können mittlerweile große Teile einer Definition durch ein grafisches Modell geschehen (vgl. auch Abbildung 26).

Um komplexe Vorgänge befriedigend definieren zu können, sind entsprechende Beschreibungselemente, wie Kontrollkonstrukte, Datentypen, etc. notwendig. Anhand der zu bewältigenden Aufgaben kann man die wichtigsten Forderungen an solche Beschreibungselemente ermitteln. Eine Prozeßmodellierung umfaßt die bereits in Abbildung 3.4.2 beschriebenen Kernbereiche:

- Organisationsstruktur,
- Prozeßstruktur,
- Informationsstruktur.

Die Beschreibung eines Workflows wird häufig mit der Ablaufstruktur gleichgesetzt, jedoch sind auch die anderen beiden Bereiche zu definieren und zu referenzieren (vgl. auch Abbildung 33 auf Seite 61). Ein Workflow selbst läßt sich in einer aktivitätsbezogenen Sicht grundsätzlich in Aktivitäten und deren Abhängigkeiten gliedern (Abbildung 27)[1]. Abhängigkeiten können Bedingungen sein, wie Aktivität X benötigt ein Ergebnis von Aktivität Y oder Aktivität X wird nach Aktivität Y ausgeführt.

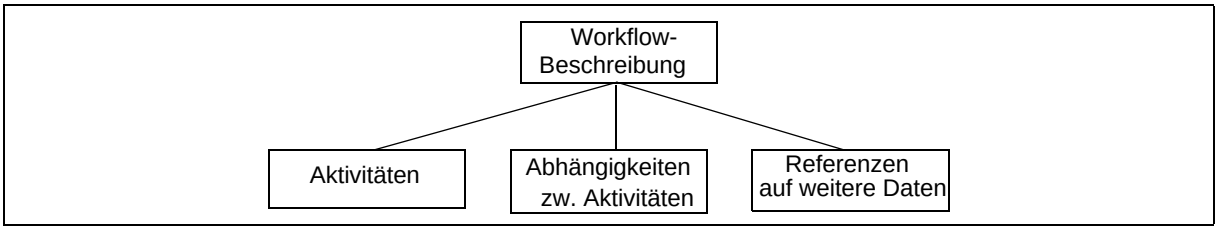


Abbildung 27: Grundbestandteile einer Workflow-Beschreibung

Damit sind die Aktivitäten, die als atomare Einzelschritte gesehen werden können, ein wesentlicher Bestandteil einer Workflow-Beschreibung. Die Organisation dieser Aktivitäten ist durch die Definition der Abhängigkeiten beschränkt und durch eine übergeordnete Beschreibung des Ablaufes (Kontrollfluß) gegeben. Die Abhängigkeiten zwischen Aktivitäten können zum Beispiel durch Vor- und Nachbedingungen für deren Durchführung ausgedrückt werden. Dadurch entstehen bereits (implizit) mögliche Abläufe, wenn nämlich durch die Erfüllung von Nachbedingungen einiger Aktivitäten die Vorbedingung(en) anderer erfüllt werden.

Es sind also (mindestens) zwei Ebenen der Beschreibung notwendig. Zum einen eine Beschreibung der Aktivitäten eines Workflows und zum anderen eine (globale) Beschreibung des Verhaltens und Zusammenspiels dieser Aktivitäten.

Beschreibung von Aktivitäten

Aktivitäten sind das zentrale Konzept jeder Prozeßdefinition. Eine Aktivität sollte durch die im folgenden genannten Aspekte genauer beschrieben sein (siehe auch Abbildung 28). Damit einhergehend findet eine Verfeinerung der Prozeßdefinition statt:

- *Typ* (z. B. manuell/automatisch)
  Wie in Abschnitt 3.4.2 definiert gibt es mehrere Arten von Aktivitäten: interaktive, manu-

1. Dies gilt für vorgangsorientierte Systeme. WFMS, die sich z. B. aus dem Dokumenten-Management entwickelt haben, sind oft dokumentorientiert und es wird der Fluß einer Dokumentmappe modelliert.

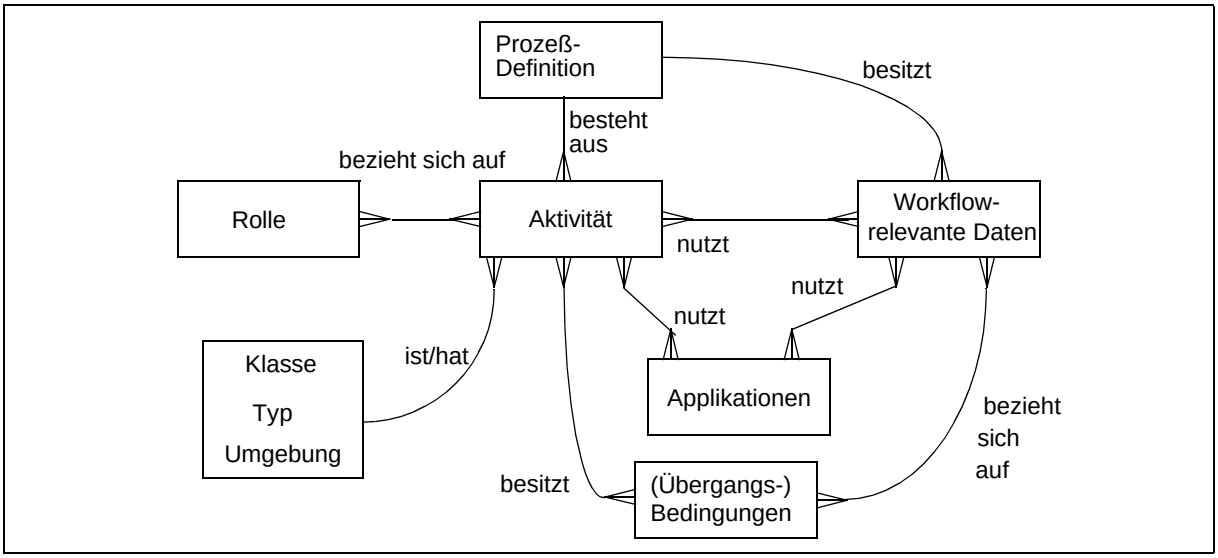


Abbildung 28: Metamodell der Prozeßdefinition (nach [WFMC 1995]*)*

elle und automatisierte.

- *Klasse* (z. B. Bearbeitung eines Dokumentes oder Bewertung eines Sachverhaltes)
  Für die Wiederverwendbarkeit von Aktivitätsdefinitionen bietet es sich natürlich an, vorgefertigte Definition anzuwenden und diese sogenannte Klassendefinition bei Bedarf nur geringfügig zu ändern und anzupassen.
- *Umgebung*
  Anforderungen an die Durchführungsumgebung, die eine Aktivität unterstützen soll, können Parameter wie das Betriebsystem, bestimmte Sub-Netzwerke oder anderes enthalten. Dies ist insbesondere für heterogene Umgebungen interessant.
- verwendete *Applikationen*
  Im Rahmen der meisten Aktivitäten muß die zu erledigende Arbeit mittels bestimmter Applikationen (z. B. Textverarbeitung) durchgeführt werden. Je nach Spezifität der Bearbeitung reicht hierbei eine Applikationsklasse aus, wie etwa Texteditor, oder eine spezielle Anwendung, wie zum Beispiel eine bestimmte Datenmaske. Applikationstypen können wiederum in einer entsprechenden Definition beschrieben werden.
- *Bedingungen*
  Um eine Aktivität starten zu können, muß sichergestellt sein, daß alle Voraussetzungen für die Durchführung erfüllt werden können (Vorbedingungen). Über eine (korrekte) Beendigung einer Aktivität entscheiden die Nachbedingungen. Diese Bedingungen sollen einen korrekten Ablauf garantieren. Im Fehlerfall sind entsprechende Maßnahmen zu ergreifen. Diese können automatisiert erfolgen, oft jedoch muß der Benutzer oder Administrator eingreifen, wenn zum Beispiel ein Workflow nicht mehr korrekt durchführbar ist (siehe auch Fehlerbehandlung in Abschnitt 3.4.5.1).
- *Workflow-relevante Daten*
  Bei den Daten kann zwischen reinen Eingabe- bzw. Ausgabedaten und Ein-/Ausgabedaten unterschieden werden [JABLONSKI UND BUßLER 1996]. Die reinen Eingabedaten und Ausgabedaten dienen nur der Parametrisierung. Die Ein-/Ausgabedaten sind Daten, die durch die Ausführung verändert werden können.
- *Rolle*
  In Bezug auf die Organisationsstruktur sollte eine Aktivität einem (oder mehreren) Bearbeiter entsprechend ihrer Rollenzugehörigkeit zugeteilt werden.

Beschreibung des Kontrollflusses

Die Definition des Ablaufes eines Arbeitsvorganges entspricht dem verhaltensbezogenen Aspekt in [JABLONSKI 1995] und bestimmt wann Workflows, Aktivitäten und Applikationen gestartet werden. Um eine genügend mächtige Beschreibung zu finden, ist zu untersuchen, welche Modelle sich zur Beschreibung von Workflows eignen. Da parallele Arbeiten unterstützt werden sollen, gelangt man schnell zu Petrinetzen (vgl. auch [VOSSEN 1995]), die eine entsprechende Mächtigkeit bieten und zudem ein ausgereiftes Konzept darstellen, um dynamische Abläufe zu definieren.

Zur Beschreibung eines Kontrollflusses können alternativ auch die aus den Programmiersprachen bekannten Kontrollkonstrukte herangezogen werden. Dies sind allgemein die sequentielle und parallele Ausführung und eine bedingte Ausführung. Aus diesen Basiskonstrukten lassen sich weitere, wie etwa Schleifen, ableiten.

Typische Konstrukte zur Festlegung des Ablaufs eines Workflows sind in Abbildung 29 aufgeführt. Die serielle Ausführung (a) startet eine Aktivität nach der Beendigung der vorherigen. Die Iteration (b) führt eine oder mehrere Aktivitäten wiederholt aus, bis das Abbruchkriterium erfüllt ist. Den *XOR-Split* (c) kann man als bedingte Verzweigung bezeichnen, d. h. es wird in Bezug auf eine Bedingung nur eine Aktivität aus den zur Auswahl stehenden gestartet. Im Gegensatz dazu werden beim *AND-Split* (d) mehrere Aktivitäten parallel gestartet. In *AND-Joins* (f) werden parallele Ausführungen von Aktivitäten wieder zusammengeführt (synchronisiert). *XOR-Joins* (e) führen keine Synchronisation durch, sondern beenden alternative Abläufe.

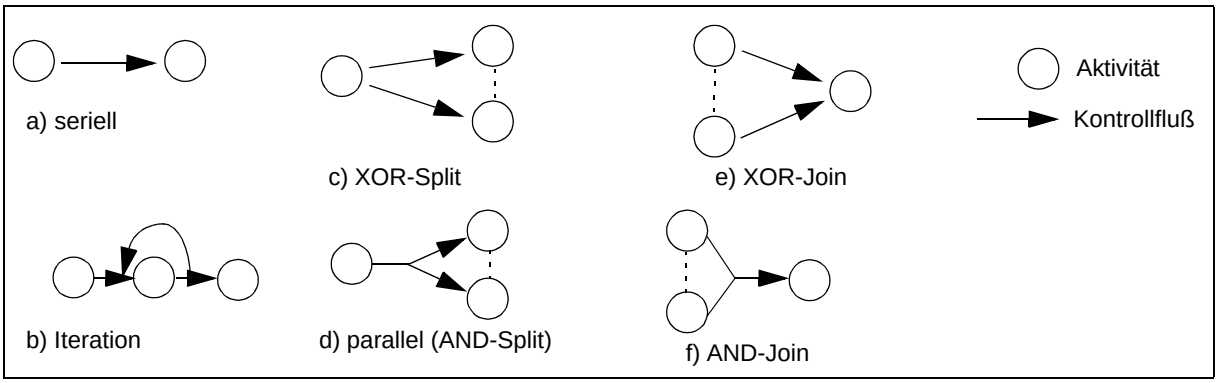


Abbildung 29: Typische Ablaufkonstrukte

Beschreibung des Datenflusses

Der Datenfluß innerhalb eines Workflows ist nicht gleich dem Kontrollfluß [REINWALD 1995], jedoch beinhaltet jeder Kontrollfluß einen gewissen Datenfluß. Beispielsweise ist es oftmals notwendig, daß Daten, die durch eine Aktivität “generiert” werden, an mehrere Stellen weitergeleitet werden, z.B. Ablage, Fakturierung und Bericht, ohne, daß dies eine weitere Aktivität zur Folge hätte.

Zuvor wurde bereits der Vorteil des digitalen Informationstransports gegenüber eines einfachen Papiertransportes angesprochen. Dabei ist zu beachten, daß alle für die Bearbeitung eines Arbeitsschritts notwendigen Daten auch zur Verfügung stehen [SHETH ET AL. 1996]. Idealerweise sollte der Benutzer sich nicht um die Besorgung der für eine Aktivität notwendigen Daten kümmern müssen. Hierfür sollten die Datenfluß- und die Kontrollflußdefinition beitragen.

Hierarchisierung mit Subworkflows

Wie sich schon aus der Definition von Workflows erahnen läßt, besteht eine Definition aus verschiedensten Teilen. So ist es nicht nur nötig, den Vorgang an sich zu beschreiben und zu regeln, sondern auch verwendete Anwendungen und Daten miteinzuschließen. Desweiteren bietet es sich aus Gründen wie Wiederverwendbarkeit, Komplexitätsreduktion und strukturierter Bearbeitung an, Workflows hierarchisch zu bearbeiten. In [JABLONSKI UND BUßLER 1996] wird ein hierarchisches Modell vorgestellt, in dem Aktivitäten nicht direkt auftreten. Stattdessen existieren elementare Workflows, d.h. Aktivitäten, die sich zu zusammengesetzten Workflows (rekursiv) erweitern lassen. Dies hat den Vorteil, daß nicht explizit zwischen Workflow und Aktivität unterschieden wird, sondern eine einheitliche Modellierungsweise und Schnittstelle entsteht.

Bei der Definition eines Workflows erhält man so einen Baum mit einem Top-Level-Workflow, der eine Anzahl von Subworkflows als Kinder hat und dessen Blätter elementare Workflows sind (vgl. Abbildung 29). Diese Superworkflows werden auch komplexe Workflows genannt. Der Top-Level-Workflow ist somit der Superworkflow seiner Subworkflows, die wiederum Superworkflows weiterer Subworkflows sein können.

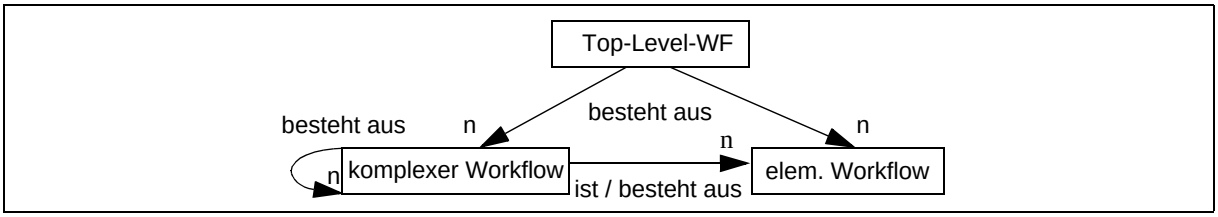


Abbildung 30: Hierarchisches Modell für Workflows

Das Konzept hierarchischer Workflows hat mehrere Vorteile, so wird die Wiederverwendbarkeit erhöht, eine Modularisierung wird direkt unterstützt und somit auch der Entwurf an sich erleichtert und übersichtlicher wird.

3.4.3.2 Ausführung von Workflows

Zur Ausführung von Workflows ist eine Infrastruktur notwendig, die eine optimale Benutzung, sowohl für Prozeßteilnehmer, wie auch für administrative Stellen, erlaubt. So ist es für den Fortgang des gesamten Workflows wichtig, diesen verfolgen zu können (*Monitoring* von Status und Historie), und für die Prozeßteilnehmer ist es wichtig, sich ihrer Aufgaben bewußt zu sein.

Zustände von Aktivitäten und Workflows

Zur Laufzeit eines Workflows sind, ähnlich wie Prozesse eines Betriebssystems, die Zustände und der Kontext von den Subworkflows zu verwalten. So werden im folgenden auch ähnliche Modelle benutzt.

Zuerst werden die möglichen Zustände von Aktivitäten betrachtet (bzw. elementarer Workflows [JABLONSKI UND BUßLER 1996]) und danach die von zusammengesetzten Workflows. Die Zustandsdiagramme in der Literatur (z.B. [JABLONSKI 1995], [WFMC 1995]) unterscheiden sich hier leicht, was durch die verschiedenen Interpretationen bei der Durchführung und Benutzung von Workflows gegeben ist. Hier wird ein Modell vorgestellt, das an die vorher genannten angelehnt ist.

Wie in Abbildung 31 dargestellt, gibt es den Anfangszustand *bereit*, der angibt, daß der Ausführung keine (formalen) Gründe widersprechen, also alle Vorbedingungen der Aktivität erfüllt sind. Die Aktivität wird somit zur Ausführung in die Arbeitsliste eingetragen. Als Endzustände sind *abgebrochen* und *beendet* definiert, die eine korrekte, d.h. die Endbedingungen wurden korrekt erfüllt, bzw. eine unvollständige Durchführung bedeuten. Soll eine Aktivität gestartet

werden, z.B. bei entsprechender Auswahl in der Arbeitsliste, muß diese instantiiert und initialisiert[1] werden. Danach befindet sie sich im Zustand *laufend*, und es müssen die notwendigen Daten und Anwendungen zur Verfügung gestellt werden. Die Abarbeitung der Aktivität kann angehalten (*pausieren*), korrekt beendet, abgebrochen oder zurückgesetzt werden. Der Zustand *beendet* kann auch explizit aus *bereit* erzwungen werden. Dies soll u. a. ermöglichen, daß ein Workflow ohne Ausführung der betroffenen Aktivität weitergeführt werden kann, etwa zu Testzwecken oder unter besonderen Bedingungen (niemand konnte die Aktivität bearbeiten). Genauso kann eine bereitstehende Aktivität abgebrochen werden, was die Aktivität als nicht korrekt beendet kennzeichnet. Unter bestimmten Umständen kann es nötig sein, eine abgebrochene Aktivität erneut auszuführen (*Retry*), wenn der (neue) Kontext eine korrekte Durchführung erhoffen läßt.

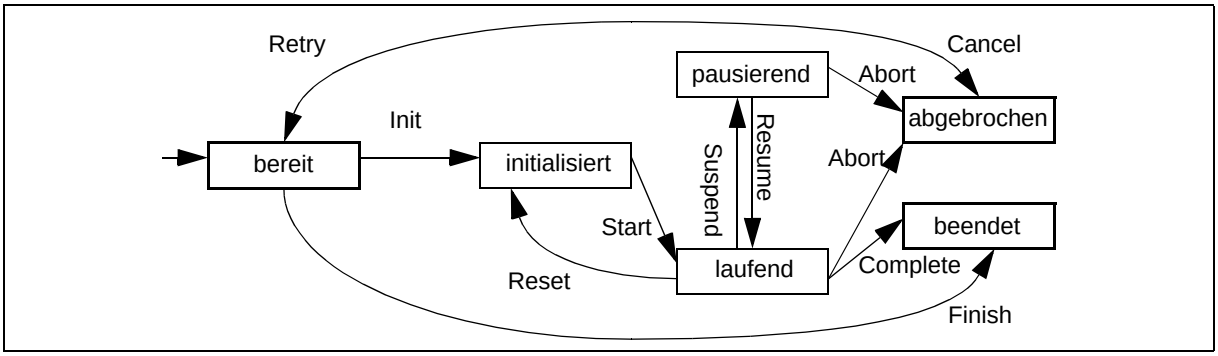


Abbildung 31: Zustände von Aktivitäten

Das Zustandsdiagramm für Subworkflows entspricht im wesentlichen dem von Aktivitäten. Jedoch sind hierbei die (End-)Zustände der eingebetteten Aktivitäten und Subworkflows von Bedeutung. D.h. das Erreichen eines Endzustandes hängt von der korrekten Ausführung der Subworkflows ab.

Steuerung des Kontrollflusses und der Datenversorgung

Der Kontrollfluß entspricht der Ausführung der Top-Level-Workflow-Instanz, d. h. das WFMS führt den gesamten Prozeß durch. Damit zeigt sich die modulare Definition von Workflows durch Hierarchisierung als vorteilhaft, so kann dieser Gesamtablauf in den Ablauf von Sub-Workflows und letztendlich Aktivitäten unterteilt werden. Hierbei obliegen dem WFMS ähnliche Pflichten, wie einem Betriebsystems, das seine Prozesse und deren Subprozesse (threads) und Ressourcen verwalten muß. Bei der Ausführung sind drei unterschiedliche Gruppen von Daten des Workflows zu unterscheiden, zuerst die WFMS-internen Daten, die der Verwaltung dienen, die Workflow-Daten, die z. B. als Parameter für die einzelnen (Sub-)Workflows verwendet werden und Daten, die ausschließlich von den auszuführenden Applikationen verwendet werden.

Die WFMS-internen Daten werden (logisch) zentral vom WFMS gehalten und verwaltet und sind von außen generell nicht sichtbar. Bei der Analogie mit Betriebssystemen entsprechen diese etwa Informationen über Prozesse, Kontrollstrukturen usw. Die Workflow-Daten finden hauptsächlich ihren Niederschlag im Datenfluß des Workflows. Sie dienen als Ein- und/oder Ausgabeparameter von Workflows und Aktivitäten. Applikationsrelevante Daten werden während der Ausführung von Applikationen modifiziert.

1. Oft werden auch die Zustände *bereit* und *initialisiert* als ein gemeinsamer Zustand dargestellt, was in der Praxis auch ausreichend sein dürfte. Jedoch soll hier der Unterschied zwischen ausführbarer und instantiierter Aktivität hervorgehoben werden.

Die applikationsrelevanten Daten oder auch Produktionsdaten werden im allgemeinen nicht durch das Workflow-System verwaltet. Dies ist nur der Fall, wenn solche Daten auch als Parameter für eine Aktivität genutzt werden. Häufig werden Daten aber auch im Rahmen der Workflow-Daten referenziert, wie zum Beispiel zu bearbeitende Dokumente. Dies ist auch ein Punkt, in dem sich unsere Anforderungen für die Unterstützung von Entwurfsanwendungen wesentlich abheben, da wir eine Systemunterstützung für gemeinsam genutzte Daten anstreben, also auch während diese durch eine Aktivität bearbeitet werden.

Ausführung von Workflows

Grundlegende Funktionen für die Ausführung von Workflows (z. B. *start*, *suspend*, *resume*, *complete*) gelten sowohl für Aktivitäten, als auch für zusammengesetzte Workflows. Die Steuerung des 'Aktivitätenflusses' (*task flow* in [LEYMANN 1995B]) spiegelt sich vor allem in der Arbeitsliste wider. Das WFMS muß für anstehende Aufgaben den ausführenden Agenten festlegen. Bei manuellen Aktivitäten also die Rolle oder den konkreten Bearbeiter ermitteln und bei automatischen Aktivitäten die ausführende Maschine, Applikationen etc. Ein Mitarbeiter kann nun über die Arbeitsliste einen Auftrag übernehmen und ihn bearbeiten. Entsprechend wird dem WFMS bekannt gemacht, wenn dieser Benutzer den Vorgang beendet, abbricht oder auch eventuell zurückstellt. Der Administrator sollte trotz allem noch in das Geschehen eingreifen können, um bestimmte Situationen (besser) lösen zu können.

3.4.4 Das Referenzmodell der Workflow Management Coalition

Das Referenzmodell der WfMC für WFMS nimmt sich vor allem der Interoperabilität zwischen Workflow-basierten Systemen durch die Definition von standardisierten Schnittstellen an (Abbildung 32). So soll es verschiedenen Produkten verschiedener Hersteller möglich sein, untereinander zu kommunizieren. Beispiele für eine solche Interaktion sind Lotus Notes und FlowMark, die über eine standardisierte Schnittstelle verbunden werden können.

Das Referenzmodell gliedert sich in die Darstellung der einzelnen Komponenten eines Workflow-Systems und die Definition der nötigen Schnittstellen.

3.4.4.1 Basiskomponenten

Ein Teil des WfMC-Referenzmodells umfaßt einen Systemaufbau (vgl. Abbildung 33), der sich in Kern und angebundene Anwendungen aufteilen läßt. Das Kernsystem ist die zentrale Einheit für die interne Verwaltung und Steuerung (vgl. 3.4.4.2) und beinhaltet den Dienst zur Workflow-Durchführung und die Workflow-Engines (Abbildung 32). Diese kommunizieren über die mit den *Workflow APIs and Interchange formats* (WAPI) angeschlossenen Komponenten, die vor allem für die Kommunikation mit den verschiedenen Benutzergruppen notwendig sind.

Ziel des Referenzmodells ist die Definition der Schnittstellen (Interface 1 - 5) zwischen dem Kernsystem und den 'generischen' Komponenten, um deren Interoperabilität zu gewährleisten. Die Gesamtheit der Schnittstellen wird WAPI genannt. Dabei sind die Funktionen innerhalb der WAPI nicht disjunkt zwischen den einzelnen Interfaces aufgeteilt. Damit sollte es verschiedenen Herstellern möglich sein, spezialisierte Komponenten für verschiedene WFMS anzubieten. So kann ein Workflow-System aus verschiedensten Komponenten zusammengestellt werden, die eine optimierte Lösung für eine konkrete Workflow-Anwendung darstellen.

Die Prozeß-Definitions-Komponente dient der Analyse, Modellierung, Spezifikation und Wartung von Geschäftsprozessen. Die Schnittstelle (Interface 1) spezifiziert vor allem den Austausch folgender Informationen:

- Start- und Endbedingungen für Prozesse,

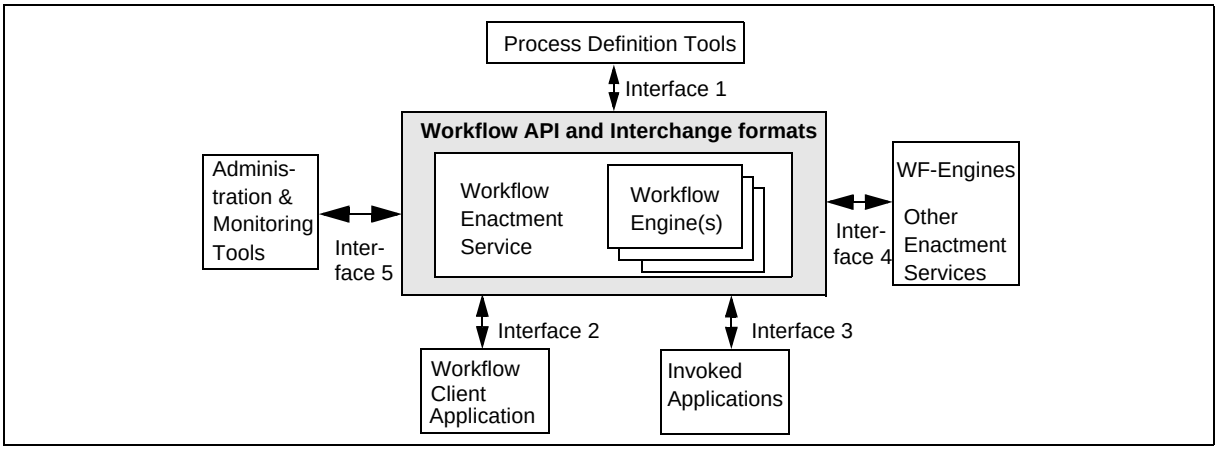


Abbildung 32: Das Referenzmodell der WfMC

- Identifizierung von Aktivitäten, deren Anwendungen und Workflow-relevanten Daten,
- Identifikation von Datentypen und Zugriffspfaden,
- Definition von Übergangsbedingungen und Ablaufregeln und
- Information über Betriebsmittelbelegungen.

Die Administrations- und Kontrollwerkzeuge stellen Dienste zur Laufzeitüberwachung, Analyse und administrativen Steuerung über die Schnittstelle 5 zur Verfügung. So kann zum Beispiel über einen Leitstand der korrekte Ablauf eines oder mehrerer Prozesse überwacht werden, oder das Management kann Analysen durchführen und so Engpässe ermitteln.

Die weiteren Workflow-Engines, die über Schnittstelle 4 angesprochen werden, sind kein direkter Bestandteil eines einzelnen WFMS, sondern für die Interoperabilität verschiedener WFMS notwendig. So können in einem Unternehmen verschiedene WFMS eingesetzt werden, die miteinander koordiniert werden können. Denkbar ist zum Beispiel, daß ein Mitarbeiter Aufträge mehrerer Workflow-Engines (verschiedener Hersteller) in seiner Arbeitsliste erhält. Grundlegende Aufgaben einer Workflow-Engine sind:

- die Interpretation von Prozeßdefinitionen,
- die Steuerung von Prozeßinstanzen (Starten, Anhalten, Beenden, ...),
- die An- und Abmeldung von Teilnehmern,
- eine Benutzungsschnittstelle,
- die Behandlung von Workflow-Kontrolldaten,
- eine Schnittstelle zum Aufruf externer Applikationen und
- Kontrollfunktionen für Verwaltung und Audit.

Die Kernkomponente ist der Dienst für das Durchführen von Workflows (*enactment service*), die für die Steuerung der Prozeßinstanzen zuständig ist. Diese kann aus einer oder mehreren Workflow-Engines bestehen. Wichtig ist die Unterscheidung gegenüber den Endbenutzer-Anwendungen, die zur Erledigung einzelner Arbeitsschritte notwendig sind.

Die Schnittstelle 2 stellt die Anbindung von Client-Applikationen, die einem Teilnehmer dessen Arbeitsliste (inkl. Zustände von Aktivitäten) anzeigen und manipulieren läßt zur Verfügung. Durch diese spezifizierte Schnittstelle, kann ein WFMS in ein anderes System eingebunden werden, wie zum Beispiel Lotus Notes. Der Endbenutzer erhält eine dem System entsprechende Sicht auf die von ihm zu leistenden Arbeiten. Grundlegende Funktionen in Zusammenarbeit mit den Workflows-Engines sind:

- Starten und Beenden einer Session,

- Steuerung von Prozeßinstanzen (Starten, Anhalten, Beenden, ...),
- Zustandsfunktionen von Prozeßinstanzen (Abfragen),
- Behandlung der Arbeitsliste,
- Prozeßüberwachung, Administration (Aktivitätssteuerung),
- Behandlung von Workflow- oder Applikationsdaten und
- Applikationsaufrufe.

Die Schnittstelle 3 ist schließlich für den Aufruf (externer) Applikationen gedacht, die für WFMS vorbereitet sind (man sagt dazu auch *workflow enabled*). Alle anderen Applikationen müssen über einen sog. Applikationsagenten integriert werden. Ein solcher Agent funktioniert wie ein Wrapper, der die WfMC-Schnittstelle 3 auf die proprietäre Schnittstelle der Applikation abbildet. Die Funktionen dieser Schnittstelle umfassen:

- Starten/Beenden einer Applikationssitzung,
- Aktivitätenmanagement (Starten, Anhalten, ...),
- Aktivitätsende-Notifikation, Ereignissignalisierung,
- Abfrage von Aktivitätsattributen und
- (Workflow-)Datenweitergabe.

3.4.4.2 Interner Aufbau eines WFMS

Wie man in Abbildung 33 sieht, kann ein solches System in drei horizontale Schichten geteilt werden: die Definitionsschicht, die Durchführungs- und Wartungsschicht und die Benutzerschicht. Im folgenden werden die wichtigsten funktionalen Komponenten eines generischen WFMS beschrieben.

Die Definitionsschicht beinhaltet die notwendige Funktionalität und Daten für eine Prozeßdefinition. Dazu sind Definitions-Werkzeuge und die Verwaltung von Prozeßdefinitionen und des Organisationsmodells notwendig. Die Prozeßdefinition beinhaltet Verweise auf Anwendungen, welche die elementaren Aktionen (Aktivitäten) durchführen sollen. Desweiteren bezieht sich die Prozeßdefinition auch auf das Organisations- und Rollenmodell.

In der Durchführungsschicht wird, wie oben schon kurz beschrieben, die Ausführung der Prozesse gesteuert. Den Kern der Durchführungsschicht bildet der sogenannte *Workflow Enactment Service*, der vor allem durch die *Workflow Engines* gebildet wird. Eine Workflow Engine führt Workflow-Instanzen unter Beachtung der zugehörigen Spezifikationen und Regeln aus. Dazu wird die Prozeßdefinition interpretiert. Im Rahmen dieser Durchführung werden elementare Aktionen in Form von Applikationsaufrufen getätigt oder ein Eintrag in eine sogenannte Arbeitsliste (*Work List*) vorgenommen. Der Eintrag in eine Arbeitsliste erfolgt anhand des Rollen- bzw. Organisationsmodells und der Prozeßdefinition. Beispielsweise ist die Lieferanfrage für ein Produkt an die zuständigen Personen der Lagerhaltung zu leiten.

Die Laufzeitdaten eines Workflow-Systems umfassen die Workflow-Kontrolldaten, die u. a. der Steuerung und Historie dienen, die Workflow-relevanten Daten, die auch von (externen) Applikationen manipuliert werden können, jedoch zusätzliche Bedeutung für die Workflow-Steuerung haben können (Datenflußspezifikation), und schließlich die Applikationsdaten, die nicht für das WFMS relevant sind und nur durch die Applikationen geändert werden.

Die Benutzungsschnittstelle ermöglicht die Interaktion zwischen Benutzer und dem *Worklist Handler* und auch zwischen Benutzer und Applikationen. Durch den Worklist Handler werden dem Benutzer die zu bearbeitenden *Work Items* präsentiert, welche die zu bearbeitenden Aufgaben beschreiben und eine entsprechende Aktivität repräsentieren. Durch die Auswahl eines Items kann bei Bedarf eine für die Aktivität notwendige Applikation gestartet werden.

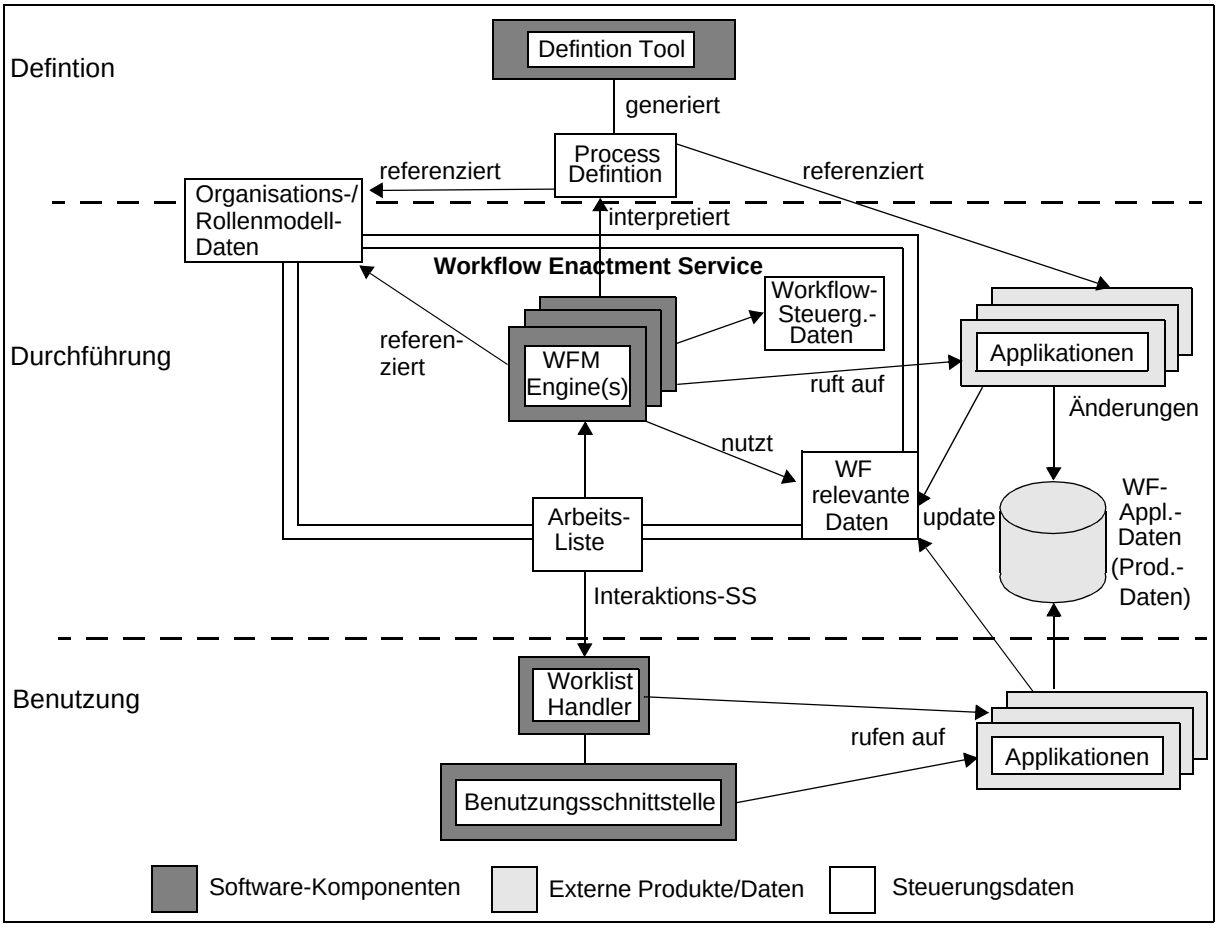


Abbildung 33: Generischer Aufbau für Workflow-Produkte (nach [WFMC 1995]*)*

3.4.5 Weitere Aspekte des Workflow-Managements

Nachdem die Grundlagen von Workflow-Technologie besprochen wurden, gehen wir im folgenden auf wichtige Aspekte ein, die zur Nutzung von Workflow-Systemen von Bedeutung sind. Dies betrifft vor allem die korrekte Durchführung von Workflows (transaktionskonzepte und Fehlerbehandlung) und das Paradigma zur Anwendungsentwicklung mit Workflows.

3.4.5.1 Fehler- und Ausnahmebehandlung in WFMS

Es gibt prinzipiell zwei Arten von Fehlern bei der Ausführung von Workflows. Logische Fehler, wie zum Beispiel eine undurchführbar spezifizierte Prozeßdefinition, und Systemfehler, wie zum Beispiel, daß das DBMS nicht läuft. Aufgabe eines WFMS ist es, soweit als möglich, solche Fehler zu erkennen und möglichst automatisch zu beheben, bzw. zur Behebung Hilfestellung zu leisten. Dabei muß jederzeit ein konsistenter Zustand des Systems gewährleistet werden.

Logische Fehler können bereits bei der Entwurfsphase eines Workflows berücksichtigt werden oder sie bedürfen menschlichen Eingreifens [HEINL UND STEIN 1996], z. B. das wiederholte Starten einer fehlgeschlagenen Aktivität. Das menschliche Eingreifen ist notwendig, da ein WFMS nicht das (semantische) Wissen besitzt, um eine beliebige Fehlersituation in einen semantisch korrekten Zustand zu überführen.

Systemfehler sollten durch das WFMS bzw. die Systeminfrastruktur (automatisch) behoben werden. Dazu zählt auch das Wiederaufsetzen nach einem fatalen Fehler. Systemfehler können die WFMS-Struktur betreffen oder auch außerhalb dieser liegen; so kann zum Beispiel ein Netzwerk “zusammenbrechen”.

3.4.5.2 Transaktionskonzepte für Workflow-Management

Wegen der schon in der Einleitung besprochenen Bedeutung eines WFMS innerhalb eines Unternehmens, muß die Sicherheit und Verfügbarkeit von Prozessen und Produktionsdaten gewährleistet sein. Zudem greifen die Teilnehmer eines Workflows ja gegebenenfalls in Konkurrenz auf Datenobjekte und Ressourcen zu. Diese Zugriffe dürfen den Ablauf und die Daten (Konsistenz) eines Workflow-Systems nicht gefährden und haben effizient zu erfolgen. Daher wird zum Beispiel die Einführung von *Concurrency Control*, *Recovery* und Transaktionsbehandlung notwendig [GEORGAKOPOULOS ET AL. 1995]. Systeme wie Lotus Notes verwenden ein Versionierungskonzept, um Inkonsistenzen zu vermeiden [REINWALD UND MOHAN 1996], manche WFMS besitzen dagegen bereits weitergehende Transaktionsunterstützung (z.B. APRICOTS [SCHNEIDER 2001]).

Die Lösung für die Zuverlässigkeit (in Bezug auf Recovery, Datensicherheit etc.) sehen viele im transaktionalen Workflow ([GEORGAKOPOULOS UND RUSINKIEWICZ 1996], [VOSSEN 1995], [SHETH UND RUSINKIEWICZ 1993] und [KAMATH UND RAMAMRITHAM 1996]). Dabei ist zu beachten, daß hier ein gewisser Unterschied der bisherigen Fokussierung auf Daten bei DBMS im Gegensatz zur Fokussierung (u. a.) auf Prozesse bei WFMS besteht.

Zur Zeit scheint es noch keine allgemeingültige Formel für transaktionale Behandlung von Workflows zu geben. Jedoch bestehen bereits einige aussichtsreiche Ansätze (z.B. das kommerzielle IBM MQ Series Workflow, oder das umfassende APRICOTS [SCHNEIDER 2001]). Ziel ist eine hohe Systemkonsistenz und das korrekte Wiederaufsetzen von Workflows im Fehlerfall.

3.4.5.3 Workflow-Management als Applikations-Builder, Workflow-basierte Anwendungen

Inzwischen gehen Firmen wie zum Beispiel IBM dazu über, WFMS als Applikationsentwicklungswerkzeug für Organisationen zu sehen [LEYMANN 1995B], bzw. als “High-Level-Betriebssystem”, das die Arbeiten und Applikationen einer Organisation koordiniert. Entsprechend können Workflow-Prozesse auch wie Software-Prozesse verstanden werden [OSETERWEIL UND SUTTON 1996].

Vergleicht man Aktivitäten innerhalb eines Workflows und Unterprogrammaufrufe so kann man gewisse Parallelen feststellen. Wie in einer Programmiersprache existieren für eine Workflow-Definition auch Programmkonstrukte, wie bedingte Verzweigung oder Schleifen. Es existieren Abhängigkeiten sowohl zwischen den Ausführungen von Aktivitäten, als auch der Aufrufstruktur (die nicht hierarchisch sein muß, etwa mittels RPC oder Interprozeßkommunikation) von Unterprogrammaufrufen. Dem WFMS obliegt nun, ähnlich einem Betriebssystem, die Kontrolle der Durchführung. Dabei sind auch die Schnittstellen bzw. Systeme zur Implementierung verteilter Applikationen, wie z. B. CORBA, SOM, DCOM, etc., zu beachten.

Wie bei den Programmiersprachen, läßt sich ein objektorientiertes Paradigma sinnvoll für Workflow-Management nutzen. Die Definition von hierarchisch aufgebauten Workflows zeigt, daß eine Kapselung und damit auch Modularisierung sehr nützlich ist. Auch die Sichtweise, daß sich ein Workflow aus bestimmten Komponenten, wie Standardapplikationen usw., zusammensetzen läßt, erlaubt den Einsatz objektorientierter Techniken [LEYMANN 1995B].

Das “Programmieren” Workflow-basierter Anwendungen kann man in zwei Stufen aufteilen [LEYMANN 1997B]. Auf der oberen Stufe geschieht das sog. *programming in the large*, d.h. hier wird der übergreifende Fluß der Implementierung festgelegt, also die Beziehungen der Aktivitäten beschrieben. Auf der unteren Ebene werden die einzelnen Aktivitäten “ausprogrammiert” (*programming in the small*), d. h. die spezifischen Algorithmen der einzelnen Aktivitäten wer-

den implementiert. So kann leicht der Programmfluß, bzw. Workflow, geändert werden, wobei die Aktivitäten unberührt bleiben können. Oder es kann neue Funktionalität durch eine neue Aktivität eingeführt werden, die sich leicht in den Workflow integrieren läßt.

In [ALONSO 1997] wird dargestellt, daß verteilte (prozeßbezogene) Anwendungen nicht allein mit heutigen WFMS vollständig realisierbar sind, jedoch das Zusammenspiel von TP-Monitoren (Transaktionssicherheit), Queuing-Mechanismen (asynchrone Kommunikation), verteilten Diensten wie CORBA (standardisierte Kommunikation von Anwendungen) und WFMS die Prozeßdurchführung solche Anwendungsszenarien ermöglicht.

3.4.6 Klassifikation verschiedener Workflow-Arten

Aufgrund des Spektrums an verschiedenen Arbeitsabläufen existieren verschiedene Realisierungsarten von Workflows. Demzufolge unterstützen moderne Systeme auch teils unterschiedliche Workflow-Arten. Wir haben in Kapitel 2 bereits angedeutet, für welche Arbeitsformen Workflow-Systeme geeignet sind. Neben administrativen Prozessen, rücken auch kollaborative in den Vordergrund. Heute ist man vor allem an der Entwicklung von sogenannten Ad-Hoc-Workflows und kollaborativen Workflows (siehe unten) sehr interessiert, um damit einerseits ein flexibleres Arbeiten mit dem WFMS zu ermöglichen und andererseits auch das Unterstützungsverhalten des WFMS den Wünschen flexibel anpassen zu können.

In der Literatur scheint sich die Unterscheidung in folgende Workflow-Arten durchzusetzen [ALONSO ET AL. 1997], [LEYMANN 1997A], [GEORGAKOPOULOS ET AL. 1995]:

- *administrativ*
  Mit administrativen Workflows sind vor allem "bürokratische Prozesse" gemeint, die einer festen Menge von Regeln unterliegen, die allen Prozeßbeteiligten bekannt sind [ALONSO ET AL. 1997]. Meistens sind diese Prozesse konkret strukturiert und leicht automatisierbar [GEORGAKOPOULOS ET AL. 1995] und wurden auch bereits durch das sogenannte *Office Automation* unterstützt. Ein Beispiel für solche Prozesse ist das Einschreibeprozedere an einer Universität oder der Reiseantrag eines Mitarbeiters. Diese Art von Workflows entsprechen auch oft dem Document-Routing-Konzept (siehe unten).

- *produktionsbezogen*
  Produktionsbezogene Prozesse stellen das Kerngeschäft eines (produzierenden) Unternehmens dar. Diese unternehmenskritischen Prozesse werden häufig durchgeführt, wobei im Gegensatz zu administrativen Prozessen komplexe und unterschiedlichste Informationsstrukturen verwendet werden [GEORGAKOPOULOS ET AL. 1995]. Die Automatisierung ist deswegen auch aufwendiger, da hier generell eine komplexere Strukturierung vorliegt. Beispiele für derartige Prozesse sind die Vergabe von Krediten oder Versicherungsabwicklungen [LEYMANN 1997A].
- *ad hoc*
  Prozesse, die nur selten auftreten bzw. häufiger aber dann meistens in abgewandelten Formen, werden durch Ad-Hoc- oder auch adaptive Workflow-Systeme implementiert. Solche Workflows werden oft in Ausnahme- bzw. Fehlersituationen oder bei einmaligen Prozessen angewandt. Es existiert kein einheitliches Verständnis für diese Art von Workflows. In [WAINER ET AL. 1996] bedeuten diese, daß der vorgegebene Ablauf zur Laufzeit geändert werden darf. Dort werden auch andere Definitionen vorgestellt, wie zum Beispiel nach dem "pass the buck" Schema, d.h. erst nach der Beendigung einer Aktivität legt der Bearbeiter das nächste Bearbeiter-/Aktivitätspaar fest. Wir werden im folgenden auch adaptive Workflows in die ad-hoc-Workflows einordnen, da hier keine allgemein-akzeptierte Unterscheidung besteht.

- *kollaborativ*
  Kollaborative Prozesse sind ähnlich wie Ad-Hoc-Prozesse eher selten bzw. sie treten in ähnlichen aber doch unterschiedlichen Ausprägungen immer wieder auf. Im Gegensatz zu Ad-Hoc-Workflows sind kollaborative Prozesse eher länger andauernd und umfassen eine größere Personengruppe, die einer Gruppendynamik unterworfen ist [ALONSO ET AL. 1997]. Ein weiterer Unterschied zu den bisherigen Typen ergibt sich daraus, daß ein solcher Workflow nicht unbedingt einen monotonen "Vorwärtsprozeß" darstellt, d. h. ein Prozeß dieser Art kann durch gewisse Iterationen und Rückschritte bestimmt sein (Gruppendynamik). Daher taucht auch die Frage auf, ob man diese allgemein mittels WFMS implementieren sollte, da der Großteil der Koordination durch die Prozeßbeteiligten und kollaborativ durchgeführt wird [ALONSO ET AL. 1997]. Diese Problematik wurde bereits in Abschnitt 2.2 angesprochen und genau hier besteht der Bedarf zur Integration von Workflow- und kooperativen Konzepten.

Es ist zu beachten. daß auch oft die Ad-Hoc-Workflows mit den kollaborativen gleichgesetzt werden (z. B. in [GEORGAKOPOULOS ET AL. 1995]) oder auch beides einfach dynamisch genannt wird; eine klare Trennung der einzelnen Arten von Workflows ist nur schwer möglich. Eine weitere Unterteilung von Workflows orientiert sich bspw. an der Implementierungstechnik [GEORGAKOPOULOS ET AL. 1995]. Hier lassen sich folgende Bereiche unterscheiden:

- Mail-basiert,
- Dokumenten-basiert und
- Prozeßbasiert.

Diese drei Merkmale lassen sich prinzipiell auf die vorher genannten Arten abbilden. So kann man kollaborative und Ad-Hoc-Workflows häufig als mail-basiert bezeichnen, administrative als dokumentenbasiert und Produktions-Workflows oft als prozeß-basiert (vgl. dazu [GEORGAKOPOULOS ET AL. 1995] und [ALONSO ET AL. 1997]). Eine weitere Klassifikation unterteilt Workflow-Arten nach unternehmerisch, forschungsbezogen und ad hoc [WAINER ET AL. 1996].

In einem Unternehmen spielen häufig wiederkehrende Prozesse eine wichtige Rolle bezüglich ihrer wirtschaftlichen Relevanz. Jedoch gibt es weitere Prozesse, deren Unterstützung durch ein WFMS vorteilhaft ist. Die Prozesse, die innerhalb eines Unternehmens anfallen, können auch anhand der Häufigkeit ihres Auftretens und der Relevanz bezüglich des operativen Betriebs eines Unternehmens eingeordnet werden [LEYMANN 1997A], [ALONSO ET AL. 1997]. Daraus ergeben sich vier Gruppen von Prozessen (vgl. Abbildung 34), wobei die Einteilungen von [LEYMANN 1997A] und [ALONSO ET AL. 1997] im wesentlichen miteinander übereinstimmen.

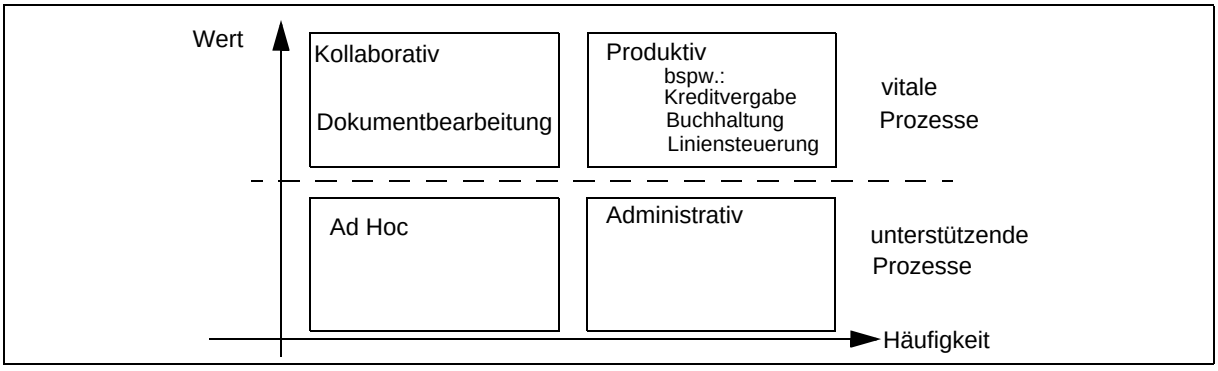


Abbildung 34: Prozesse eingeteilt nach Wert/Häufigkeit [LEYMANN 1997A]

Eine weitere Klassifikation aus [RITTER 1997B] bzw. [DEITERS ET AL. 1996A] unterscheidet allgemeine Abläufe mittlerer Aufgabenstrukturierung (semi-strukturiert) anhand der Vorplanbarkeit in Bezug auf die Informationsbasis, den Lösungsweg und die Kooperationspartner. Semi-

Strukturierte Abläufe sind hier eher mit kollaborativen bzw. Ad-Hoc-Prozessen zu vergleichen. In Abbildung 12 auf Seite 34 wurde bereits ein zusammenfassender Vergleich von administrativen Prozessen und Entwurfsprozessen vorgestellt, wobei die Entwurfsprozesse entsprechend der besprochenen Workflow-Arten als kollaborativ zu kennzeichnen sind.

Zusammenfassend gibt es folgende Kriterien bzw. Dimensionen, die nicht unbedingt orthogonal zueinander sind, nach denen sich Workflow-Arten einteilen lassen:

- Aufgabenstrukturierung
- Aufgabenkomplexität
- Wert für das Unternehmen
- Häufigkeit des Auftretens
- Automatisierbarkeit
- Vorplanbarkeit der Informationsbasis
- Vorplanbarkeit des Lösungswegs
- Vorplanbarkeit der Kooperationspartner.

Die verschiedenen Ausprägungen erfordern auch Beachtung bei der Implementierung. Produktions-Workflows und administrative Workflows sollten ein sehr hohes Maß an Datensicherheit bieten, z. B. durch entsprechende Transaktionskonzepte und Recovery-Fähigkeit. Ad-Hoc-Workflows sollten eine möglichst effiziente Bedienweise bieten und kollaborative Workflows sollten große Flexibilität unterstützen.

3.4.7 Einordnung und Abgrenzung von Workflow-Management

Wie der Einsatz zahlreicher Workflow-Produkte belegt, werden viele Anforderungen an Workflow-Systeme bereits erfüllt. Produktions- und administrative Prozesse werden bereits gut durch WFMS unterstützt. Viele Hersteller bieten auch schon Ad-hoc-Workflow-Fähigkeiten an. Die Definition von Workflows werden grundsätzlich grafisch durchgeführt, wenn auch häufig eine Skriptsprache existiert.

Viele Systeme stützen sich auf DBMS zur Verwaltung ihrer Daten (MQ Series Workflow [IBM 2000], Prominand [IABG 1999] etc.) und bieten hiermit eine gewisse Sicherheit, aber transaktionaler Workflow wird kaum realisiert. Ein Teil der Systeme nutzt Document-Routing als Basis der Arbeitsverteilung an Stelle von prozeß-bezogenen Paradigmen.

Einordnung

Zur Einordnung von WFMS ist auch deren geschichtliche Entwicklung von Interesse. So stammen viele der heutigen Produkte ursprünglich aus den Bereichen Büro-Automation, Imaging, Dokumentenmanagement und auch CIM (Computer Integrated Manufacturing) [SCHULZE UND BÖHM 1996], [JABLONSKI 1995], [JABLONSKI UND BUßLER 1996]. Weitgehendere Grundlagen für das Workflow-Management sind dabei verteilte Betriebssysteme, entsprechende Middleware- und Datenbanksysteme. Die Gruppe der CSCW-Systeme soll allgemein das kooperative Arbeiten unterstützen und wird deswegen oft als Obergruppe angesehen, wobei sich WFMS vor allem den prozeßorientierten Vorgängen widmen [SCHULZE UND BÖHM 1996].

WFMS werden nach verschiedensten Kriterien eingeordnet. Dies liegt zum einen an den verschiedenen Einsatzgebieten, von der Unterstützung von Büroarbeiten über die Produktionssteuerung (z.B. Liniensteuerung) bis hin zum unternehmensweiten Einsatz. Zum anderen liegt dies auch an den verschiedenen Sichtweisen der Forschung und Entwicklung (Programmiersystem, Datenbanken, usw.). Prinzipielle Dimensionen nach denen sich WFMS einordnen lassen umfassen u. a. das Einsatzgebiet, die Art der unterstützten Arbeit und die Art und Weise der Unterstützung der Prozeßregelung (vgl. Abschnitt 3.4.6).

Abgrenzung

Technologien, die zum Teil ähnliche Ziele wie WFMS verfolgen sind CAx-Frameworks, CSCW und DBMS. CAx-Frameworks bilden einen Verbund von Werkzeugen und Software-Systemen, um z. B. den Entwurf im CAD-Bereich zu unterstützen (vgl. Abschnitt 3.6). Die Abläufe die während eines Entwurfsprozesses notwendig sind, werden nur begrenzt durch den Framework aktiv unterstützt. Die Integration der verschiedenen Komponenten ist weitgehender als in Workflow-Systemen, da die Komponenten aufeinander abgestimmt sind.

Datenbanksysteme bieten nur eine geringe direkte Applikationsfähigkeit. Jedoch stellen diese auch zentrale Systeme in Unternehmen dar und verfügen über eine ausgereifte Technologie. TP-Monitore realisieren die Steuerung von Applikationsabläufen, aber bieten dafür nicht die Mächtigkeit von Workflow-System, jedoch transaktionsbasierte Konzepte.

Im Bereich CSCW besteht ein Schwerpunkt auf der Kommunikation und (ad hoc) Kollaboration von Personen [BORGHOFF UND SCHLICHTER 1995]. Workflow-Management wird als eine Untergruppe aufgefaßt, die speziell zur Unterstützung strukturierter und prozeßbasierter Arbeit gedacht ist. CSCW ist somit eine Grundlage für Software-Systeme, die Gruppen von Personen für kollaboratives Arbeiten unterstützen. Das Zusammenwachsen dieser beiden Technologien zeigen wir im folgenden Abschnitt.

3.4.8 Bedeutung der Workflow-Technologie für kooperative Prozesse

Da Workflow-Management wie auch Groupware die Zusammenarbeit von Personen abbildet, ist es interessant zu untersuchen, welche prinzipiellen Unterschiede und Gemeinsamkeiten existieren. In vielen CSCW-Veröffentlichungen (beispielsweise [BORGHOFF UND SCHLICHTER 1995], [HASENKAMP ET AL. 1994], [SCHILL 1996]), wird Workflow-Unterstützung als Teilbereich der CSCW-Forschung gesehen. In Abschnitt 3.3.7 wurde bereits gezeigt, daß sich Workflow-Systeme gegenüber anderen CSCW-System durch ihren Koordinationscharakter auszeichnen. Abbildung 35 zeigt einen Vergleich in dem Spektrum Systemorientierung/Menschorientierunug. WFMS orientieren sich mehr auf die Systeme und CSCW zum Menschen.

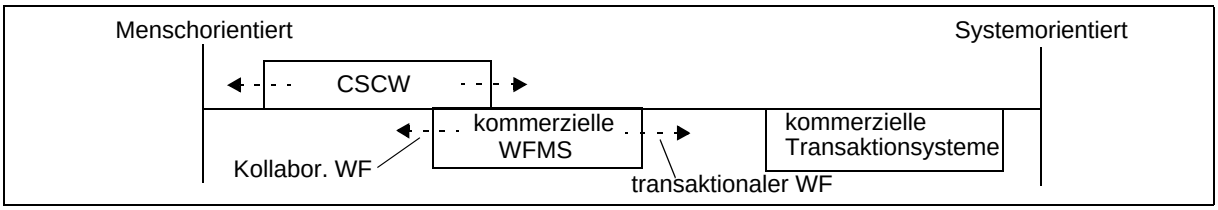


Abbildung 35: Technologie-Einordnung in das Spektrum Mensch-System [SHETH ET AL. 1996]

In [SCHALLER UND SCHWAB 1997] wird das Projekt PlanKo beschrieben, das eine integrierte Kooperationsplanung zur Verfügung stellt. Ein Bestandteil davon ist ein WFMS. Ebenso wird in [DEITERS ET AL. 1996A] die Möglichkeit der Verschmelzung von CSCW-Systemen und WFMS diskutiert und vorgestellt. CSCW-Systeme bieten im allgemeinen freiere Koordinations- und Kommunikationsmöglichkeiten als WFMS [SCHWAB 1995], [DEITERS ET AL. 1996A]. Jedoch gibt es Überlappungen beider Bereiche, wobei in jedem Bereich versucht wird, Eigenschaften des anderen einzubringen. So wird zum Beispiel in TACTS, einem CSCW-System, Workflow-Funktionalität eingebracht [TEEGE 1993], [TEEGE 1996A].

Wegen der weitgehenden Anforderungen an WFMS ist heute fraglich, ob dieser Name für moderne Systeme überhaupt treffend ist. In [SHETH ET AL. 1996] wird deswegen der Begriff *Work Activity Coordination* gewählt. In dieser Publikation werden auch die Probleme von heutigen Systemen, die Arbeit rechnerbasiert unterstützen, zusammengefaßt. Zur Zeit existieren Systeme, die eher unstrukturierte Arbeit unterstützen (Groupware) oder eher strukturierte Arbeit

(WFMS). Jedoch ist eine große Zahl der Prozesse nicht auf einen dieser Bereiche beschränkt (vgl. Abbildung 36). In dieser Abbildung ist zu sehen, daß sich existierende Technologien, wie Workflow- und Groupware-Systeme nur für einen geringen Anteil der Prozesse einsetzen läßt. Die Mehrzahl der Prozesse liegt aber zwischen den ausgeprägten Technologien, wo wir auch die von uns betrachteten kooperativen Prozesse einordnen.

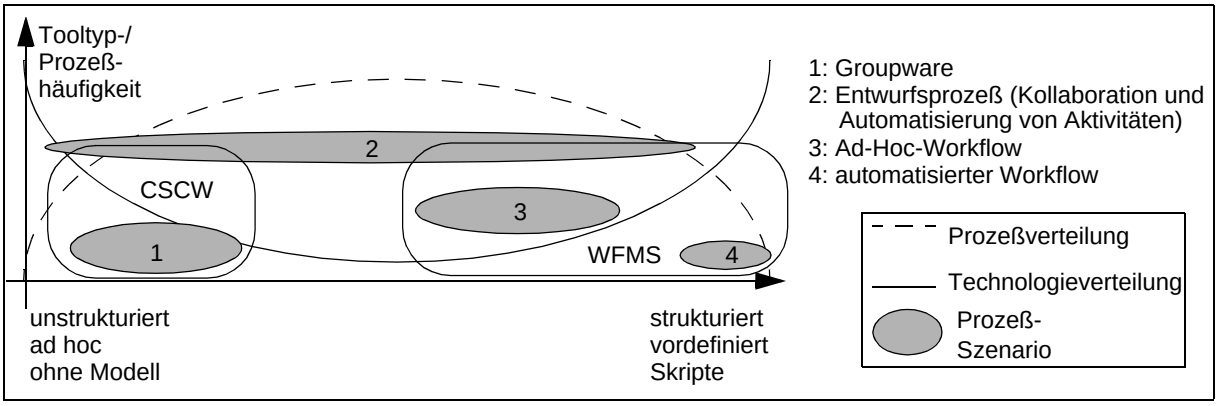


Abbildung 36: Technologieverteilung und Prozeßverteilung (vgl. [SHETH ET AL. 1996]*)*

Wie aus dieser Darstellung klar wird, sollte der Schwerpunkt der Entwicklung und Forschung die Möglichkeit zur Unterstützung sowohl von unstrukturierten wie auch strukturierten Prozessen bieten. Das scheint auch einleuchtend, d. h. es werden Routinearbeiten bzw. häufig wiederholte Prozesse durch das System implementiert. Arbeiten, die sich flexibel entwickeln und/oder deren Ablauf nicht festgelegt ist, sollten zwar eine Unterstützung durch das System erfahren, jedoch ist deren Koordination im wesentlichen den beteiligten Personen zu überlassen. Wünschenswert wäre es dabei, daß beide Arten der Unterstützung durch das selbe System erfolgen, um somit eine integrierte Arbeitsweise zu ermöglichen. Genau dieser Ansatz wird von uns verfolgt. Zum Beispiel ist im Entwurfsbereich die Unterstützung flexibler Kooperationstechniken als auch die Automatisierung wiederkehrender Arbeiten notwendig [MITSCHANG ET AL. 1996]. Daher sollte in modernen Systemen möglichst das gesamte Spektrum der Arbeitsunterstützung geboten werden, also von unstrukturiert bis hin zu strukturiert.

In Abbildung 36 ist zu sehen, daß ein großes Potential an Systemen in den Randbereichen existieren. Für die große Anzahl tatsächlicher Arbeitsprozesse, die zwischen diesen Bereichen existieren, sind kaum Systeme vorhanden. Diese Lücke muß in Zukunft geschlossen werden. Das heißt, daß moderne Workflow-Systeme eine allgemeine Arbeitsunterstützung bieten müssen oder leicht mit anderen Systemen für eine flexible Arbeitsunterstützung zu kombinieren sein sollten. Genau in dieser Lücke sehen wir auch unsere Aufgabenstellung, die kooperative unstrukturierte Prozeßanteile mit strukturierten verbinden will.

3.5 Agententechnologie

Workflow-Systeme fokussieren sich auf das Vorantreiben und Koordinieren von Abläufen. Eine unserer Anforderungen ist die Unterstützung von Kommunikation und Kooperation für Personen und Komponenten, die nur in geringem Maße durch Workflow-Systeme realisierbar ist. Verteilte Agentensysteme hingegen zeichnen sich u.a. durch die Anwendung fortgeschrittener Verhandlungsmechanismen aus. Wir werden im folgenden verdeutlichen, daß einige Aspekte für uns nutzbringend angewendet werden können.

3.5.1 Grundlagen für Agenten

Im Rahmen dieser Arbeit betrachten wir verteilte Agentensysteme, wobei wir jedoch nur die Aspekte der verteilten künstlichen Intelligenz (VKI) vertiefen, die für uns von Bedeutung sind. Wichtige Eigenschaften eines Agenten sind:

- handelnd
  Ein Agent handelt, d.h. er bewirkt Aktionen und Reaktionen (lat. agere: handeln, vorantreiben). Dies kann man als das wichtigste Merkmal kennzeichnen.
- wahrnehmend
  Um zu handeln bzw. zu reagieren, muß ein Agent auch seine Umwelt wahrnehmen.
- kommunizierend
  Ein Agent ist nicht von seiner Umgebung isoliert, sondern kommuniziert mit anderen (Agenten).
- planend
  Die Aktionen und Reaktionen sind deliberativ und folgen Regeln oder Vorgaben um ein Ziel zu erreichen.
- entscheidend
  Der Agent entscheidet über auszuführende Aktionen oder Interaktionen. Dies kann vollkommen autonom erfolgen.

Je nach der Ausprägung der einzelnen beschriebenen Merkmale und den Einsatzgebieten kann man verschiedene Arten von Agenten unterscheiden, wie zum Beispiel Benutzeragenten, Sensoragenten, Roboteragenten, mobile Agenten usw. Im Bereich der VKI sind vor allem autonome Agenten hervorzuheben, die selbstständig ihre Aktionen entscheiden. Prinzipiell kann man die Architektur eines Agenten in dem Insekten-Sinnbild aus Abbildung 37a ausdrücken [HAUGENEDER 1994]. Einen Agenten kann man in drei Teile strukturieren. Der Kommunikationsteil (Kommunikator und Kommunikationskanäle) ermöglicht den Kontakt mit seiner Umgebung, wie zum Beispiel Kommunikation mit anderen Agenten oder der Nutzung von Ressourcen. Der Kopf ist der Vermittler zwischen der eigentlichen Funktionalität eines Agenten und dem Kontext des Problems, d.h. es wird entschieden wie (Protokolle, Kommunikationsmechanismen) und welche Aktionen (Funktionalität) ausgeführt werden. Der Körper umfaßt das Reservoir an Aktionen und den Zustand des Agenten:

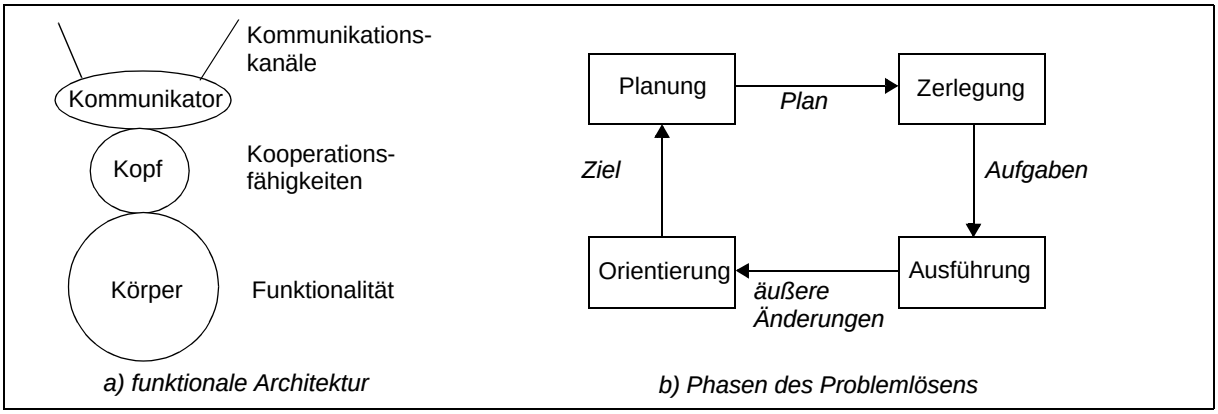


Abbildung 37: Architektur von Agenten und Phasen des Problemlösens

In Abbildung 37b werden die Phasen bei der Problemlösung für einen Agenten dargestellt. Nach diesem Schema gehen viele Agentenimplementierungen vor. Zuerst wird der Kontext evaluiert (Orientierung) und ein abgeleitetes Ziel festgelegt. Die Erreichung des Ziel wird an-

schließend geplant. Der resultierende Plan wird in einzelne Teilaufgaben (tasks) zerlegt, die dann letztendlich ausgeführt werden. Diese Ausführung führt ggf. zu einer Änderung des Zustands des Agenten und seiner Umwelt. Diese Zustände werden dann wiederum evaluiert usw.

3.5.2 Kommunikation

Durch die oben erläuternden Aspekte ist es möglich ein System aus Agenten zu bilden, die miteinander kommunizieren und gemeinsame Aufgaben lösen (Agentengesellschaft). Dies kann sowohl kooperativ als auch im Wettbewerb geschehen. Dafür ist natürlich eine gemeinsame Sprache notwendig, also eine Agent Communication Language (ACL). Hier gibt es Standardisierungsbemühungen der ARPA (Advanced Research Projects Agency), die in einer Dreiteilung einer ACL münden:

- Vokabular
  Dieses ermöglicht eine anwendungsspezifische Sprache zu definieren, die als weitere Grundlage für die Kommunikation dient.
- Knowledge Interchange Format (KIF)
  Das KIF basiert auf einem erweiterten Kalkül der Prädikatenlogik und wird als Austauschformat genutzt um Inhalte darzustellen, z.B die Grundfläche des 'chip1' ist größer als die des 'chip2': `(> (* (width chip1)(length chip1)) (* (width chip2)(length chip2)))`
- Knowledge Query and Manipulation Language (KQML)
  Die KQML erlaubt nun die Definition von Aktionen für die Agentenkommunikation, die auch Performatives genannt werden. Solche Performatives sind Sprechakte, die gewisse Kommunikationsanfragen ausdrücken, wie Gesuche, Befehle, Antworten oder einfache Mitteilungen. Ein Beispiel für eine solche Nachricht und deren Antwort, wird anhand der Nachfrage für den Preis einer IBM-Aktie dargestellt [BORGHOFF UND SCHLICHTER 2000]:

Anfrage	**Antwort**
(**ask**	(**tell**
:sender *A*	:sender *share agent*
:content (share value *IBM ?price*)	:content (share value *IBM 96.625*)
:recipient *share agent*	:recipient *A*
:reply to *IBM share*	:in reply to *IBM share*
:language *LProlog*	:language *LProlog*
:vocabulary *NY-Stock-Exchange-TICKS*)	:ontology *NY-Stock-Exchange-TICKS*)

3.5.3 Interaktion

Die Kommunikationsmöglichkeiten sind durch eine ACL sehr weitgehend. So lassen sich damit komplexe Protokolle realisieren. Ein Agent verfolgt eine Strategie, und versucht deren Ziele zu erreichen. Daher bieten sich Protokolle wie Konversationsnetze oder Sprechakte an, die komplexe Verhandlungen ermöglichen. Durch die Möglichkeiten des KIF sind auch die Inhalte von Konversationen fast beliebig wählbar und erlauben eine weitgehende semantische Darstellung.

Ein Ziel, speziell der künstlichen Intelligenz, ist es flexible Kommunikationsabläufe zu unterstützen. D.h. ein Agent verwendet nicht nur fest vorgegebene Nachrichten sondern kann auch dynamisch Nachrichten erstellen und verstehen. Die Realisierung solcher intelligenten Agenten ist jedoch nicht Thema dieser Arbeit.

Ein weiterer Ansatz, der zur Zeit viel Beachtung findet sind Konversationspläne und *Conversation Policies* (z.B. [LAURENCE UND LINK 1999]). Konversationspläne beschreiben und leiten die Interaktion zwischen Agenten anhand der Definition des Ablaufs des Nachrichtenaustauschs und der Zustandsänderung [BARBUCEANU UND LO 1999]. Dadurch lassen sich Interaktionsmuster zwischen Agenten beschreiben, ohne deren Implementierung vorzuschreiben.

3.5.4 Scheduling

Agentensysteme können für die Planung und Durchführung von Prozessen genutzt werden [LEVI UND HAHNDEL 1993]. Dies soll unter anderem eine dynamischere Realisierung von Prozessen erlauben. Da wir in dieser Arbeit für unsere strukturierten Anteile keinen Bedarf sehen, existierende Workflow-System durch Agentensysteme zu ersetzen und für Kooperationen, das Scheduling nicht von großer Bedeutung ist, gehen wir hier auch nicht näher auf diese Möglichkeiten ein.

3.5.5 Zusammenfassung und Ausblick

Die Agententechnologie besitzt eine besondere Philosophie bzgl. verteilten Systemen, da sie von der Betrachtung einzelner unabhängiger Komponenten ausgeht, die im Zusammenspiel die gewünschte Funktionalität erreichen. Die dafür entwickelten Methoden und Technologien sind sehr fortgeschritten und erlauben es weitreichende Problematiken anzugehen. Wir sehen auch kritische Aspekte, wie die Garantie der Funktionsfähigkeit eines System autonomer Agenten.

Im folgenden Kapitel wird beschrieben, wie insbesonders die adaptiven und flexiblen Kommunikationsfähigkeiten von Agenten für uns eine Realisierungsmöglichkeit von Kooperationsszenarien ermöglichen. Für dies wären auch Konversationsnetze [WINOGRAD UND FLORES 1986] prinzipiell geeignet, jedoch stellen die existierenden Ansätze für Agenten bereits eine breite Palette an anwendbaren Modellen und Werkzeugen bereit, die vielversprechend für den Einsatz bzw. eine Implementierung sind. Wir denken hier speziell an die Kooperationspläne, die eine weitgehende geregelte Kooperationsunterstützung ermöglichen.

3.6 CAD-Frameworks und Designflow-Management

Da Entwurfsprozesse im CAD-Bereich ein Ausgangspunkt für diese Arbeit waren, sind CAD-Frameworks zu betrachten. Prinzipiell stellt ein CAD-Framework eine Möglichkeit dar, alle Werkzeuge und Systeme integriert für den CAD-Entwurf zu nutzen. Aufgrund der großen Bedeutung von CAD-Frameworks wurde u.a. auch die CAD-Framework Initiative gegründet, die zur Standardisierung und Entwicklung beigetragen hat [CFI 1991]. Zudem hat sich die Meinung gefestigt, daß man auch die Prozeß-Steuerung in einen CAD-Framework integrieren sollte. Für die tatsächliche Durchführung existieren verschiedene Vorschläge. Im folgenden Abschnitt vertiefen wir das Thema CAD-Framework und danach das Designflow-Management und die Kooperationsunterstützung im CAD-Entwurf.

3.6.1 Eigenschaften von CAD-Frameworks

Ein CAD-Framework ist eine gemeinsame Anwendungsumgebung für CAD-Werkzeuge [RITTER 1997B]. Ein mögliche Darstellung der Architektur eines solchen Frameworks ist in Abbildung 38 zu sehen (vgl. [BOSCH 1995], [VAN DER WOLF ET AL. 1990], [BRETSCHNEIDER 1993]). Das Tool-Framework-Interface dient der Anbindung von Werkzeugen und Erweiterungen. Dazu bietet der Framework-Kern die gemeinsamen Dienste an und bedient sich auch der Entwurfsdatenverwaltung. Dies ist für die Konsistenz im Entwurf wichtig, da eine Regelung der Zugriffe auf die Daten stattfinden kann. Werden zusätzlich Metadaten verwendet, kann man von einem Repository sprechen. Die angebundenen Werkzeuge können anhand der Integrationsweise und Funktionsweise unterschieden werden. Die integrierten Werkzeuge kommunizieren direkt mit dem Framework Interface. Die gekapselten Werkzeuge besitzen hingegen nicht direkt die Fähigkeit mit dem Framework zu kommunizieren und werden mit Hilfe sog. Wrapper eingebunden. Ein Wrapper realisiert die Abbildung von Werkzeug auf Framework Interface und umgekehrt. Als letzte Gruppe sind die Framework-Werkzeuge zu nennen, über welche die

Benutzer die Funktionalität des Frameworks direkt nutzen können. In dieser Kategorie befinden sich auch Werkzeuge für die Benutzung erweiterter Funktionalitäten (wie z.B. die schraffierte Designflow-Komponenten in der Abbildung, siehe dazu auch Abschnitt 3.6.2).

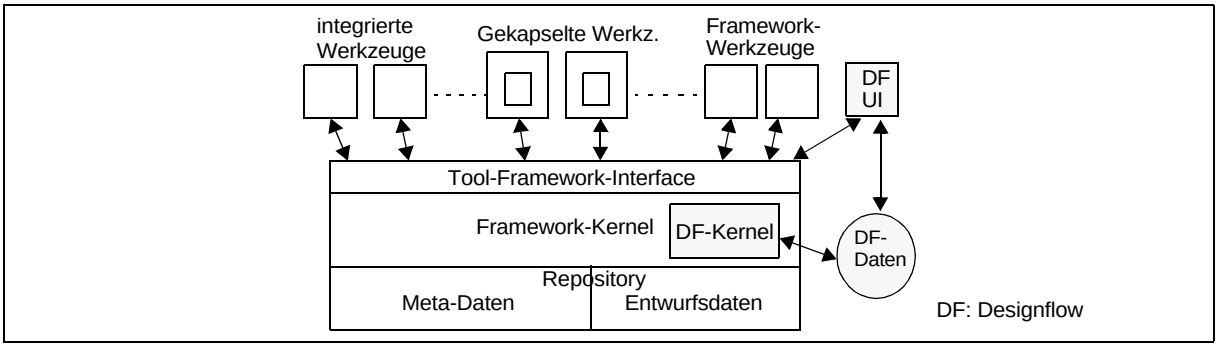


Abbildung 38: Beispielarchitektur eines CAD Frameworks mit Designflow-Erweiterung

Nach [HARRISON ET AL. 1990] bzw. [RITTER 1997] sind die wichtigsten Framework-Dienste:

- Prozeß- und Dateiverwaltungsdienste,
- Datenrepräsentationsdienste,
- Datenverwaltungsdienste,
- Versionierungsdienste,
- Benutzungsschnittstellendienste,
- Werkzeugintegrationsdienste,
- Entwurfsmethodikdienste und
- Entwurfsprozeß-Administrationsdienste.

3.6.2 Designflow-Management mit CAD-Frameworks

Das Konzept des CAD-Frameworks hat bedeutende Vorteile durch den Framework-Charakter, der sowohl eine weitgehende Erweiterbarkeit als auch Anpaßbarkeit ermöglicht. Außerdem lassen sich damit die konsistente Nutzung von Funktionen und Daten realisieren. Die Framework-Werkzeuge sind der Ansatzpunkt zur Integration von Designflow-Management [BOSCH 1995], welches die Benutzer bei der Durchführung eines Entwurfsprozesses unterstützen soll.

Der Schwerpunkt eines Designflow-Prozeß im Bereich CAD bedeutet häufig die Regelung der einzelnen Werkzeuganwendungen, also eine feingranulare Steuerung. Ein Vorteil ist die Integration der Daten in den Framework. Dadurch wird es erst möglich auch die gesamten Arbeitsdaten in einen Entwurfsprozeß systemunterstützt einzubeziehen, etwa im Gegensatz zu den Applikationsdaten im Workflow-Management. Problematisch ist es jedoch die Einbringung externer Daten in den Framework, da viele Anwendungswerkzeuge proprietäre Formate unterstützen (siehe bspw. [SELLENTIN ET AL. 1999]).

Beispiele für weitentwickelte CAD-Frameworks aus der Forschung sind etwa CONCORD [RITTER ET AL. 1994], Nelsis [VAN DER WOLF ET AL. 1990] und Odyssey [BROCKMAN ET AL. 1992]. Diese nutzen auch die oben erläuterte Architektur aus, um Designflow-Management zu integrieren. Zudem ist in allen diesen Ansätzen die Modellierung der Entwurfsdaten besonders ausgearbeitet (z.B. OTO-D in Nelsis).

3.6.3 Kooperationsunterstützung für CAD-Umgebungen

Zunehmend gewinnt die Kooperationsunterstützung an Bedeutung, wie in [RITTER 1997B] beschrieben, wenn dort auch ein Schwerpunkt auf Vorplanbarkeit und transaktionale Konzepte gelegt wurde im Gegensatz zu denen von uns gewünschten flexibleren Kooperationsprotokollen. CONCORD ermöglicht es, kooperativen Entwurfsaktivitäten jeweils eine definierte Menge von Daten gemeinsam zu nutzen. Die beteiligten Aktivitäten besitzen Rechte (*owner*, *write*, *read*) auf diese Daten. Um zu kooperieren stehen die Funktionen *permit*, *transmit* und *revoke* zu Verfügung, um dynamisch die Rechte anzupassen und Daten austauschen zu können. Dadurch können bereits viele indirekte Kooperationsmuster realisiert werden.

Ein weiterer Ansatz ist in [ROLLER UND ECK 1999] beschrieben. Dort wird die Modellierung von Daten im Rahmen von Struktur und semantischer Beziehungen gezeigt, die in Verbindung mit der Verwendung von Constraints für diese Beziehungen sehr mächtig ist. Die Kooperation wird durch Gruppentransaktionen ermöglicht, die aufgrund fehlender ACID-Eigenschaften (z.B. Serialisierbarkeit) informelle Korrektheitskriterien in Zusammenhang mit einem Notifikationsmechanismus erlauben. Dazu muß eine Menge von ECA-Regeln definiert werden, die bei einem erkannten Zugriffskonflikt bestimmen, in welchen Fällen welche Beteiligten benachrichtigt werden. Dadurch können Konflikte auf gemeinsamen Daten durch die Entwerfer erkannt werden.

An diesen beiden Beispielen sieht man bereits zwei verschiedene Kooperationsmuster: CONCORD verhindert Konflikte durch eine feste Rechtevergabe, ermöglicht es jedoch diese Rechte zwischen kooperierenden Aktivitäten zu übertragen. [ROLLER UND ECK 1999] verhinderen Konflikte nicht unbedingt vorab, aber erkennen diese und propagieren sie zu den beteiligten Benutzern. Beide Ansätze sind sehr fortgeschritten. In der hier vorliegenden Arbeit werden wir sehen, daß wir auch beide Ansätze in einem Modell nutzen können, aber auch zusätzliche Kooperationsmuster unterstützen, wie z.B. direkte Verhandlungen zwischen Benutzern oder Aktivitäten.

3.7 Weitere relevante Ansätze

In diesem Abschnitt werden weitere Technologien vorgestellt, die zur Unterstützung von Entwurfsprozessen herangezogen werden können. Es wird die Modellierung von Software-Prozessen mit sog. Prozeß-Einheitselmenten vorgestellt, die Interaktionen zwischen Prozeß-Schritten darstellen können. Der TOGA-Ansatz beschreibt die Integration heterogener Anwendungen auf Basis eines gemeinsamen Informationsraums, was für die Realisierung der Anforderung zum gemeinsamen Datenzugriff bedeutend ist. Schließlich werden Netzpläne als Mittel zur Darstellung von grobgranularen Prozessen und den dort bestehenden Abhängigkeiten vorgestellt.

3.7.1 Prozeßmodellierung für das Software Engineering

Nach [BALZERT 1998] bildet das sogenannte Prozeßeinheitselement (Abbildung 39) die Grundlage einer Prozeß-Architektur. Dort wird auch die ETXM-Spezifikation (Entry, Task, Exit, Measurement) aus [HUMPHREY 1989] vorgestellt:

- Vorbedingungen: Resultate, die vor Beginn der Aufgabe erfüllt sein müssen
- Ergebnisse: Resultate, die erzeugt werden müssen
- Rückkopplung: von und zu anderen Aufgaben (Ein/Aus)
- Aufgabe: Was zu tun ist, durch wen, wie und wann, einschließlich entsprechender Standards, Verfahren und Verantwortlichkeiten
- Maße: geforderte Aufgabenmaße (Aktivitäten, Ressourcen, Zeit), Ausgaben (Anzahl,

Größe, Qualität) und Rückkopplungen (Anzahl, Größe, Qualität)

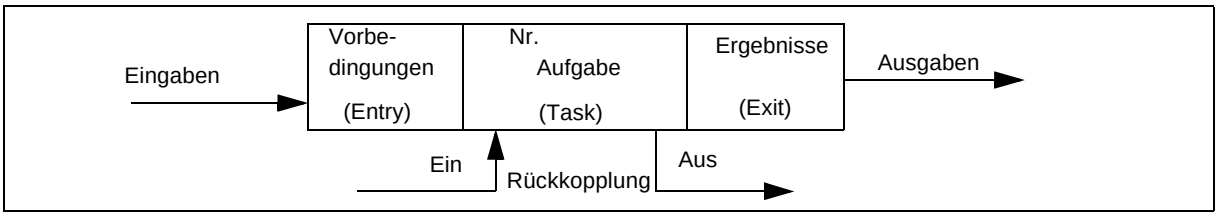


Abbildung 39: Prozeß -Einheitselement

Die in obiger Abbildung dargestellten Elemente können nun zu Prozeß-Modellen zusammengeschaltet werden (siehe Abbildung 40). Die Ausprägung eines Prozeß-Elements wird in [BALZERT 1998] Aktivität genannt, wir werden stattdessen einfach Element schreiben um einen Unterschied zu Workflow-Aktivitäten anzudeuten. Ein Prozeß-Modell entsteht durch die Verknüpfung der Ein- und Ausgabe-Schnittstellen der (spezifizierten) Elemente. Zudem sind bidirektionale Rückkopplungen zwischen verschiedenen Elementen möglich:

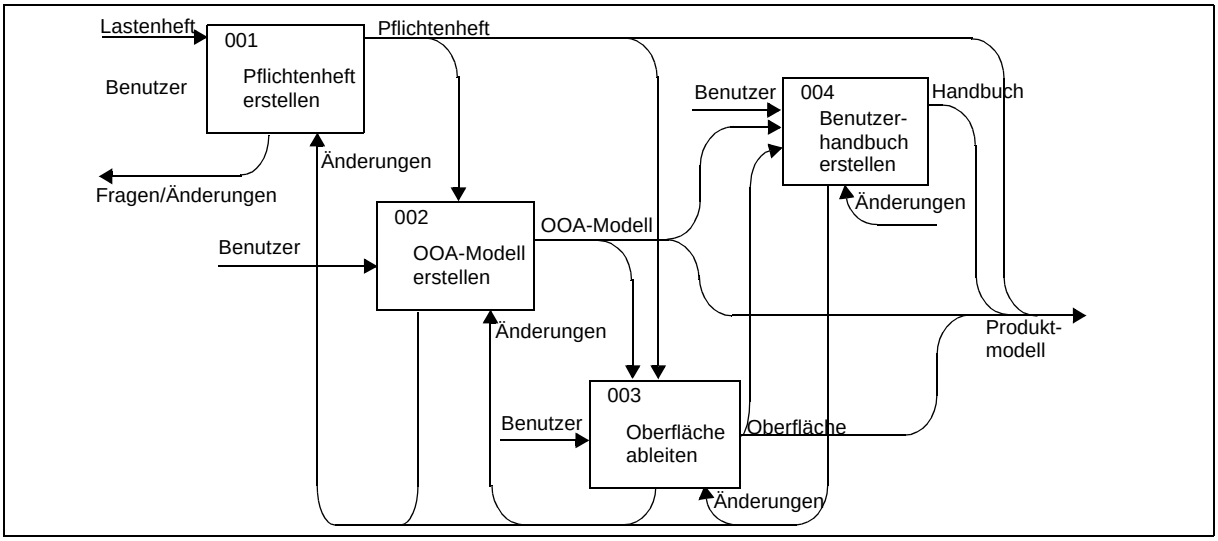


Abbildung 40: Software-Definitionsprozeß modelliert mit Prozeß-Einheitselementen

Ein solchermaßen beschriebener Prozeß unterscheidet sich zunächst lediglich gering von gewöhnlichen Workflow-Definitionen. Ein Unterschied findet sich aber in der relativ grobgranularen Spezifikation der Elemente. Dies ist im Workflow-Management zwar ohne weiteres möglich, aber nicht unbedingt sinnvoll [BECKER UND ZUR MÜHLEN 1999].

Die Rückkopplung stellt ein mächtiges Konstrukt für die Modellierung dar. Dieses ist nicht exakt mit typischer Workflow-Semantik darstellbar, da dort kein dynamischer Datenaustauch zwischen simultan stattfindenden Aktivitäten vorgesehen ist (nur Ein-/Ausgabe-Daten für den Beginn bzw. das Ende einer Aktivität). Ein lediglich ähnliches Verhalten ließe sich bspw. durch eine Schleife simulieren, in der die Kopplungspartner entsprechend wiederholt gestartet werden. Wobei die zuvor gestartete Instanz einer Aktivität "korrekt" beendet werden muß, was nicht trivial automatisiert werden kann. Bei vielen vernetzten Elementen erschwert sich dabei auch der Aufbau einer solchen Schleife, was den Workflow sehr unübersichtlich machen dürfte.

Es stellt sich die Frage, warum nicht feinere Elemente definiert werden. Ein tatsächlicher Software-Entwicklungsprozeß ist nur schwer in feingranulare Tätigkeiten im voraus definierbar. Dies liegt an vielen Faktoren, speziell kann etwa die Reihe der Werkzeug-Aufrufe nicht vorhergesehen werden, da dies der Entwickler dynamisch entscheidet. Ein anderes Beispiel sind die

von Kundenseite häufig auftretenden Anforderungsänderungen, die zu entsprechenden Prozeß-Änderungen führen können. Daher bieten sich grobe Aktivitäten an, die bei solchen Hindernissen ihre Gültigkeit über die Prozeß-Dauer hinweg behalten.

Zusammenfassung

Wir halten die Prozeß-Einheitselemente für eine interessante Methode zur Spezifikation von Teamarbeit. Speziell durch die Möglichkeit der Rückkopplung unterstützen sie synchrones Arbeiten auf gemeinsamen Daten. In der vorgeschlagenen Verwendung sehen wir zudem eine Definitionsmöglichkeit von Vorgängen, die nicht dem feingranularen Wunsch bezüglich Workflow-Definitionen entspricht. Daher denken wir, daß dieses Modell zwar eine gute Beschreibungsmöglichkeit für Software-Entwicklungsprojekte ist, sich aber nicht besonders für Workflow-Unterstützung eignet.

Es existieren auch Referenzmodelle für Software-Entwicklungsprozesse, wie das Wasserfall-, V-, Spiral-, Prototypen oder evolutionäre Modell. Einige sind strikter spezifiziert (z.B. V-Modell), andere weniger (z.B. Prototypen-Modell). Diese bieten nur ein grobgranulare Beschreibung der Prozeßklasse “Software-Entwicklung”. Auf die meisten dieser Modelle können die Prozeß-Einheitselemente angewandt werden.

Zusammenfassend bewerten wir die Prozeß-Einheitselemente als eine gute Art zur Beschreibung von Software-Prozessen, die im Rahmen von Team-Arbeit erstellt werden, da dort bereits kooperative Aspekte berücksichtigt werden. Dies scheint uns wichtig, da wir übereinstimmen mit der Meinung, daß Software-Entwicklung gegenseitige Abstimmung benötigt, die auch durch die vorgestellte Rückkopplungsmöglichkeit unterstützt werden kann. Wie wir bereits dargelegt haben, sind die Rückkopplungen nur schwer durch Workflow-Konstrukte darstellbar. Daher denken wir auch, daß klassisches Workflow-Management nicht das sinnvolle Mittel zu einer vollständigen Unterstützung des Software-Prozesses ist.

3.7.2 TOGA

Mit dem TOGA-Prototyp [SELLENTIN ET AL. 1999][FRANK ET AL. 2000] wurde gezeigt, wie heterogene Applikationen einen gemeinsamen Informationsraum bilden können und die dort enthaltenen Daten entsprechend auch nutzen können (siehe Abbildung 41). Bedeutende Ansätze waren hierbei die Definition von verschiedenen Schnittstellenschichten, um die Implementierung zwischen heterogenen Applikationen zu erlauben. Dies erlaubt es, ein gemeinsames Modell für die zu koordinierenden Daten zu bilden, also einen gemeinsamen Informationsraum.

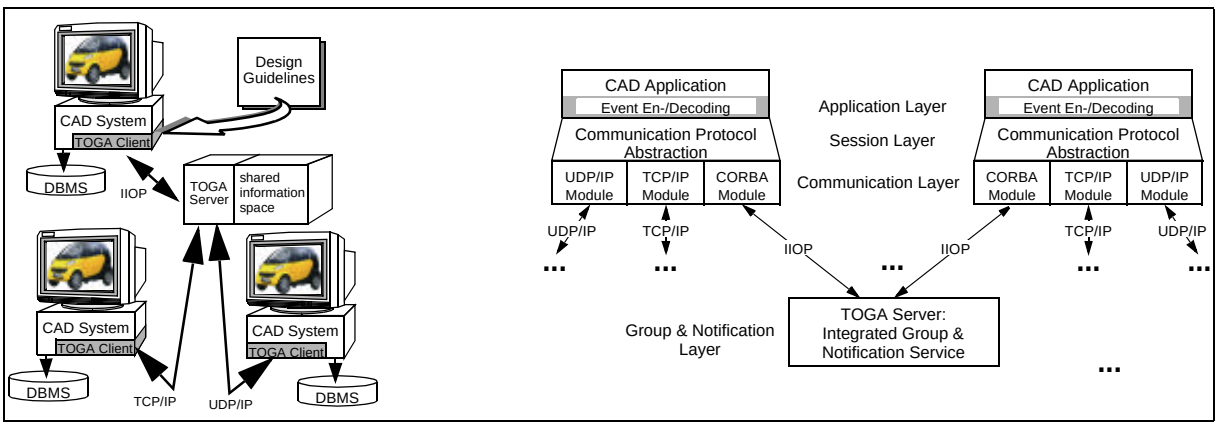


Abbildung 41: Entwurfskoordination mittels TOGA [FRANK ET AL. 2000]

Ein weiterer Vorteil des TOGA-Ansatzes ist die variable Definition des Datengranulats und der auf den Daten erlaubten Operationen. Je nach Einsatzszenario kann dadurch ein Spektrum von loser bis enger Kopplung zwischen den Applikationen erreicht werden. D.h. es kann festgelegt werden, welche Operationen Konflikte erzeugen und welches Granulat sie besitzen. Beispielsweise kann dies schon das Modifizieren eines einzelnen CAD-Elements (z.B. Rechteck) bedeuten oder aber erst die Modifikation einer ganzen Elementengruppe (z.B. Motor).

Durch TOGA ist es somit möglich, Entwurfs-Ergebnisse zwischen verschiedenen Applikationen bekannt zu machen. Dadurch wird auch Group Awareness erreicht. Zusätzlich erlaubt das implementierte Konzept von Undo-Operationen und der Abstimmung mittels Two Phase Commit Konflikte im Entwurf zu vermeiden. In [FRANK ET AL. 2000] wird gezeigt wie TOGA auch auf Entwurfsprozesse anwendbar ist.

3.7.3 Prozeßplanung/Netzplantechnik

Die Netzplantechnik ist anwendbar auf die Beschreibung von Projekten und Prozessen, insbesondere um zeitliche Prozeßfortschritte zu planen (und zu kontrollieren) und deren Abhängigkeiten zu dokumentieren [KLEEDÖRFER 1998].

Es ist auch zu beachten, daß das Prozeß-Modell erst durch einen Projektplan verfeinert werden soll. Dabei werden die in Elementen spezifizierten Aufgaben in Vorgänge untergliedert. Diese sind dann z.B. in Netzplänen darstellbar. Uns ist jedoch nicht vollkommen ersichtlich, ob diese Darstellung tatsächlich allgemein als feiner zu bezeichnen ist (siehe Diskussion unten). Vorteilhaft ist jedoch sicher die grafische Darstellung asynchroner Arbeit und deren zeitliche Einordnung.

Aus Prozeß-Modellen lassen sich Netzpläne erstellen, da hier ähnliche Konstrukte verwendet werden. Dies sind u.a. die Ende-/Anfangs-Beziehungen. Zudem können zeitliche Überlappungen von Aktivitäten durch die Wahl geeigneter Zeitabstände erreicht werden. Aus diesen Plänen lassen sich sehr gut zeitliche Planungen und die Verfolgung eines Projekts durchführen. Leider fehlen in diesen Diagrammen so wichtige Informationen, wie etwa die Rückkopplung im Prozeß-Modell. So ist es wohl nur sinnvoll beide gleichzeitig einzusetzen. Zudem sind durch die übliche Darstellung in Gantt-Diagrammen auch keine Schleifen von Aktivitäten direkt darstellbar (was für eine Zeitplanung auch nicht passend wäre).

Daher sehen wir in der Abbildung von obigen Prozeß-Modellen auf einen Netzplan zum einen keine Verfeinerung eines Vorgangs sondern eine konkrete zeitliche Umsetzung eines Prozeß-Modells. Zum anderen sind die Rückkopplungen in den bekannten Netzplanmodellen nur gering realisiert. Daher wurden auch im Rahmen von [HÜBNER 1998] und [KLEEDÖRFER 1998] dynamische Effekte und Änderungen von Abhängigkeiten und deren Darstellung in Netzplänen weiterentwickelt. Dies kann eine sinnvolle Ergänzung für die Bedienunterstützung bei Entwurfsprozessen darstellen, da dadurch ein zusammenfassende und übersichtliche Information über den aktuellen Zustand eines Prozesses gegeben werden kann.

3.8 Anwendung auf kooperative Prozesse

Die zuvor besprochenen Eigenschaften verschiedener Arbeitsformen, erfordern auch verschiedene Mechanismen zu deren Unterstützung. Die Vorstellung unterstützender Technologien läßt uns einige Schlüsse bezüglich eines umfassenden Systems ziehen. Da Entwurfsprozesse ein großes Spektrum bezüglich der Strukturierbarkeit von Prozessen besetzen, ist es nicht sinnvoll allein Workflow-Management-Technologie einzusetzen. Jedoch ist eine eingeschränkte Beschreibung des Entwurfsablaufs vorab möglich und sinnvoll. Durch ein Aktivitäten-Management können auch die einzelnen Aktivitäten in einem Entwurfsprozeß besser koordiniert

werden. Die Verwendung eines Aktivitätskonzepts ist für uns daher eine Anforderung, obwohl wir nicht den Blackbox-Charakter von Workflow-Aktivitäten beibehalten wollen, sondern auch interaktive Mechanismen integrieren wollen, speziell bezüglich der Datenbehandlung.

Groupware unterstützt die kollaborative Arbeit in Teams. Diese Systeme setzen den Fokus auf kooperative Mechanismen, wie Kommunikation, gemeinsame Datenbearbeitung und Arbeitsunterstützung. Kennzeichnend sind die Aspekte der Gruppe (bzw. Team), der gemeinsamen Aufgaben, Ressourcen und Kommunikationsmedien, wie auch der sog. Group Awareness, welche die Koordination durch das Bekanntmachen der Arbeit verschiedener Personen erleichtert. Beispiele für Groupware sind TeamWave [WINDRIVER 2001], Telekonferenzen [STREITZ ET AL. 1994], E-Mail, Lotus Notes [IBM 2001] und MS Exchange [MICROSOFT 2001].

Workflow-Management-Systeme unterstützen die automatisierte Durchführung des Ablaufs eines Prozesses. Da hier auch verschiedene Personen beteiligt sind und zusammenarbeiten, werden Workflow-Systeme dem Forschungsbereich CSCW hinzugerechnet. Sie nehmen jedoch eine Sonderstellung ein, da sie besonders gut für strukturierte und regelbare Prozesse einsetzbar sind und nur wenige direkte Kooperationsmechanismen unterstützen (siehe Tabelle 1). Workflow-Systeme sind sehr gut geeignet um strukturierte Prozesse zu modellieren und durchzuführen, die häufig wiederholt werden. Kooperationen können nur direkt durch ein Workflow-System unterstützt werden, wenn diese vorab modellierbar sind. Um die Schwächen von Workflow-Systemen auszugleichen können Groupware-Systeme genutzt werden. Daher streben wir die Integration dieser Technologien an.

Prozeßmerkmal\Technologie	Workflow Management	Groupware
hohe Struktur	sehr gut, z.B. Workflow-Modellierung	befriedigend, nicht Schwerpunkt
semi-strukturiert	nur teilweise, z.B. durch adaptive oder flexible Workflows	gut bis sehr gut
regelbar	sehr gut	gut bis sehr gut (z.B. Lotus Notes)
gering regelbar	schlecht	gut bis sehr gut
häufige Durchführung	eher gut, falls strukturiert	befriedigend bis gut
seltene Durchführung	nur bedingt, da eine Prozeßmodellierung sehr aufwendig ist	gut
hohe Bedeutung	sehr gut, z.B. Durchführungsgarantien, Wiederholbarkeit etc.	befriedigend
geringe Bedeutung	abhängig vom Aufwand	gut
kooperativ	möglich falls vormodellierbar, bzw. falls keine Systemeinbindung nötig	sehr gut, z.B. Telekonferenz oder gemeinsame Informationsräume

Tabelle 1: Prozeßmerkmale und Systemunterstützung

Vereinfacht dargestellt, sequentialisieren Workflow-Definitionen abhängige Arbeitsschritte und benötigen deshalb auch keine Unterstützung von interaktiven Mechanismen. Da, wie beschrieben, es jedoch nicht möglich ist Entwurfsprozesse vorab vollständig zu beschreiben oder gar zu “sequentialisieren”, benötigen wir andere Konzepte zur Vermeidung oder Auflösung von Konflikten. Ein wesentlicher Aspekt ist dabei die Koordination auf Ressourcen bzw. speziell auf Entwurfsdaten. Dies ermöglicht auch die systemgestützte Steuerung von Werkzeuganwendungen, jedoch nicht im Rahmen eines klassischen Aktivitätskonzepts (sog. implizite Kooperation/Koordination). Daher benötigen wir Kooperationsverfahren, welche die Mängel bezüglich einer alleinigen Ablauforientierung ausgleichen sollen.

Die wichtige Bedeutung der gemeinsamen Nutzung von Daten in kooperativen Prozessen erfordert eine entsprechende Verwaltung und Organisation dieser Daten. Außerdem ist aufgrund der (Daten-)Abhängigkeiten der einzelnen Entwurfsschritte eine Propagation von Zwischenergebnissen und eine Interaktion bezüglich dieser Daten erstrebenswert. Dies führt auch zur Anwendung von Kooperationsmechanismen. Hier spielen natürlich entsprechende Verfahren aus dem CSCW eine bedeutende Rolle, wie etwa die Verwendung von Informationsräumen.

Einen weiteren wichtigen Punkt stellt die Datenmodellierung dar. Dies betrifft zum einen die Zugriffsprotokolle, die je nach Datenart und Anwendungsszenario unterschiedlich sein können. Zum anderen ist die Strukturierung der Daten (z.B. Stücklisten) und deren Granulat auch an das jeweilige Szenario anzupassen. Dabei ist zu beachten, daß das Granulat von Daten auch durchaus Auswirkung auf das Granulat von Aktivitäten besitzt. So werden im CAD-Bereich häufig einzelne Funktionsanwendungen von Werkzeugen bereits als Aktivitäten gesehen. Jedoch muß die von uns gewünschte Unterstützung von Abläufen auch ein gröberes Granulat unterstützen, wie beispielsweise, die Begutachtung eines potentiellen Bauplatzes.

3.9 Lösungsansatz

Lösungen versprechen wir uns aus dem Workflow-Management-Bereich bezüglich der Entwicklung eines Aktivitätenmodells. Die von uns gewünschte Datenunterstützung ist jedoch besser auf CSCW-Technologien abbildbar. Die Flexibilität des Ansatzes wollen wir einerseits durch die Anwendung von agentenähnlichen Technologien für die Protokolle, sowohl zwischen Benutzern als auch auf Daten, erreichen. Andererseits ist der Framework-Charakter, wie er in CAD-Frameworks zu finden ist ein guter Ansatz, um ein System an verschiedene Anwendungsszenarien anpassen zu können

Unser Lösungsansatz betrachtet die Integration dieser Technologien und Modelle zur gesamtheitlichen Unterstützung der in den Szenarien beschriebenen Arbeitsvorgänge. Wir fokussieren unsere Lösung auf die gleichberechtigte Kombination von koordinativen und kooperativen Mechanismen. Aus der Erkenntnis heraus, daß kooperative Prozesse diese Anforderungen stellen, stammt die in Abbildung 42 dargestellte Architektur, die eine angepaßte Zusammensetzung und Nutzung der unterstützenden Technologien ermöglicht. Die zu erreichende Prozeßunterstützung nennen wir *Designflow-Funktionalität*, die den Nutzern über eine entsprechende Benutzerassistenz zur Verfügung steht [RITTER UND MITSCHANG 1997]. In dieser Architektur werden die von Workflow-Systemen und Groupware angebotenen Konzepte sowie weitere Basisdienste durch einen Integrationsdienst kombiniert, um schließlich die gewünschte Designflow-Funktionalität zu erreichen.

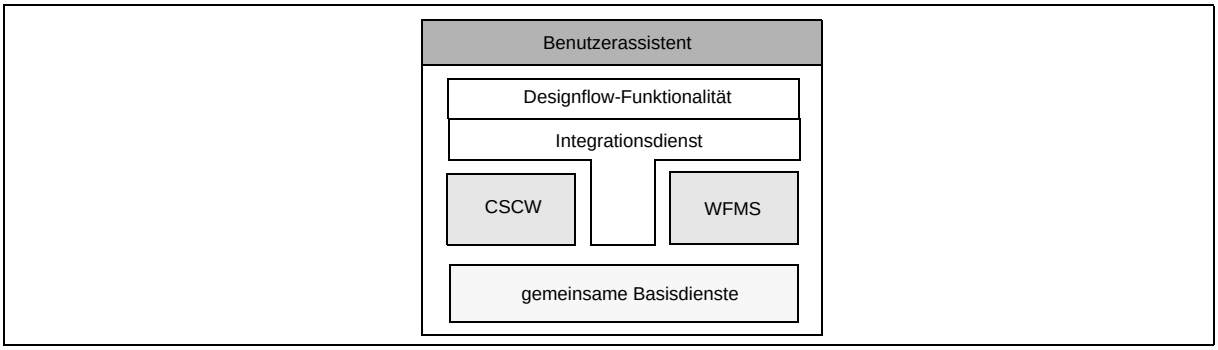


Abbildung 42: Grobarchitektur des Lösungsansatzes

In Tabelle 2 werden die von uns gewünschte Funktionalität den entsprechenden Technologien gegenüber gestellt, die diese Aspekte erfüllen können. Wie man der Tabelle entnehmen kann, liegt das Hauptaugenmerk zur Realisierung auf dem Aktivitätenmodell, dem gemeinsamen In-

formationsraum und den Kooperationsprotokollen. Das Aktivitätenmodell soll eine Strukturierung von einzelnen Arbeitsschritten erreichen. Die Daten (-Objekte) stehen über einen gemeinsamen Informationsraum allen Aktivitäten zur gemeinsamen Nutzung zur Verfügung. Die Objekte selbst sind strukturierbar, um die Zerlegung von Entwurfsartefakten und die resultierenden Abhängigkeiten abzubilden. Die Kooperationsprotokolle ermöglichen schließlich die Interaktion zwischen Aktivitäten und die Nutzung der Objekte und bilden somit einen wichtigen Integrationsbestandteil. Das resultierende ASCEND Designflow Model (ADM) wird in Kapitel 4 beschrieben:

Merkmal	**Technologie**
Ablaufunterstützung	Workflow-Management, Aktivitätenmodell
konkurrierende Datennutzung	übergreifendes Objektformat, Informationsraum, Kooperationsprotokolle
Entwurfsdaten	Objektmodell mit Strukturelementen, Spezifikation von Entwurfszielen
Unterstützung kooperativer Abläufe	Aktivitäten-Delegation, Delegations- und Integrationsprotokolle, dynamische Aktivitätenstruktur, Entwurfsobjekte und deren Spezifikationen
Anpaßbarkeit an verschiedene Anwendungen	Aktivitäten-Delegation, anpassbare Delegations- und Integrationsprotokolle, dynamische Aktivitätenstruktur, Entwurfsobjekte und Spezifikationen
Kooperationsunterstützung	Informationsraum, Interaktions-/Kooperationsprotokolle, Einbindung von Groupware
Technische Realisierung	Middleware zur Integration existierender Dienste und Systeme, gemeinsames Aktivitäten-Management über Groupware und Workflow hinweg mit zusätzlicher Funktionalität für spezifische Entwurfsaktivitäten

Tabelle 2: Technologien für die Umsetzung der Anforderungen

4 Das ASCEND Designflow Model

In diesem Kapitel besprechen wir das ASCEND Designflow Model, welches die Unterstützung der in Kapitel 2 beschriebenen kooperativen Prozeßklasse erreicht. In Frage kommende Technologien zur Realisierung wurden in Kapitel 3 besprochen. Hier werden wir nun ein Modell vorstellen, welches diese Technologien aufgreift, geeignet kombiniert und erweitert.

4.1 Überblick

Zur Vereinfachung bezeichnen wir im folgenden die von uns untersuchten kooperativen Prozesse als *Entwurfsprozesse*[1]. Die grundlegende Technik, um die Komplexität von Entwurfsprozessen zu beherrschen, ist das 'divide et impera'. Dabei wird ein Entwurfsartefakt in separate Entwurfsobjekte zerlegt. Diese Entwurfsobjekte werden von verschiedenen Entwerfer-Teams bearbeitet, die jeweils für den Entwurf 'ihres' Objekts verantwortlich sind.

Da die separat entwickelten Entwurfsobjekte später wieder zu integrieren sind, ist es wichtig, dies bezüglich der parallel stattfindenden Entwurfsaufträge der verschiedenen Entwerfer-Teams zu berücksichtigen. Es ist jedoch schwer, Konfliktsituationen zu erkennen oder sogar zu vermeiden. Leider verfügen gängige CAX-Werkzeuge nur über geringe Koordinationsmechanismen für die Steuerung von Entwurfsaktivitäten. Die Teilnehmer in einer solchen Umgebung müssen generell andere Mittel zur Kooperation nutzen (z.B. E-Mail, Telefon), oder bestimmte systemexterne Richtlinien für den Entwurf einsetzen. Da diese Techniken nur außerhalb des Systems bekannt sind, kann die Korrektheit von Entwurfsaktivitäten vom System nicht automatisch kontrolliert werden. Dies führt zu einer aufwendigen und fehlerträchtigen Entwurfskontrolle, bzw. macht die Entwurfskontrolle gar unmöglich.

Der Fokus im Projekt ASCEND (Activity Support in Co-operative ENvironments for Design issues) ist es, diese Probleme zu adressieren und dafür passende Lösungen zu entwickeln. Durch das Aktivitätenmodell des ADM (ASCEND Designflow Model) kann ein spezifischer Designflow an den jeweiligen Entwurfsprozeß und dessen Methodik angepaßt werden. Ein Designflow beschreibt komplexe Entwurfsaktivitäten, die sich aus strukturierten Anteilen und weniger strukturierten Anteilen zusammensetzen. Zur Integration und Kooperationserweiterung werden für Aktivitäten ein gemeinsamer Informationsraum und Interaktionsprotokolle eingesetzt.

Nachdem die Problemstellung und mögliche Realisierungstechnologien besprochen wurden, wird in diesem Kapitel das ADM vorgestellt. Dieses Modell bildet auch die Basis für den Prototypen CASSY (Cooperative Activity Support System), der im nachfolgenden Kapitel vorgestellt wird. Das ADM baut auf drei Säulen auf: das Aktivitätenmodell, dem Informationsraum und dem Interaktionsmodell. Wir beginnen in Abschnitt 4.2 damit den Integrationsgedanken zu skizzieren und einige grundlegende Begriffe zu definieren, um in Abschnitt 4.3 einen Überblick über das ADM vorzustellen. Daraufhin besprechen wir das Aktivitätenmodell des ADM in Abschnitt 4.4 und wenden dies in Abschnitt 4.5 auf das Planungsszenario aus Abschnitt 2.3.3 an. Danach führen wir den Informationsraum und dessen Kooperationsprimitive in Abschnitt 4.6 und die Interaktionsprotokolle in Abschnitt 4.7 ein. Schließlich fassen wir das ADM in Abschnitt 4.8 zusammen.

1. Leider existiert noch keine allgemein akzeptierte und eingrenzende Bezeichnung, der von uns untersuchten Prozesse. Daher bedeutet Entwurfsprozeß im folgenden einen Arbeitsvorgang, der im Sinne der Aufgabenstellung durch unser Modell zu unterstützen ist.

4.2 Ausgangssituation und Begriffsfestlegung

Es ist bekannt, daß Workflow- und Groupware-Technologien für die Unterstützung von Entwurfsaktivitäten in Bereichen wie CAD, Software-Entwicklung oder allgemeiner Computer-gestützter Planung erfolgreich eingesetzt werden können [RITTER ET AL. 1994]. Jedoch gibt es kaum Workflow- oder Groupware-Systeme, die sämtliche Belange der von uns anvisierten Prozesse (vgl. Abschnitt 2.3) unterstützen. Ein Grund dafür ist die eingeschränkte Fokussierung beider Technologien: So konzentriert sich Workflow-Technologie auf weitgehend festgelegte und vordefinierte Ablaufmodellierung und schränkt dadurch die im Entwurfsprozeß nötigen Freiheitsgrade unakzeptabel stark ein. Eine Workflow-Definition wird viele Male instantiiert und der beschriebene Ablauf jeweils automatisch durchgeführt. Groupware unterstützt dagegen die Kooperation zwischen Gruppenmitgliedern. Sie konzentriert sich auf mächtige Kooperationsprotokolle, die einen gemeinsamen Workspace nutzen. Da diese Kooperationen jedesmal unterschiedlich durchgeführt werden, lassen sich auch Groupware-Aktivitäten nicht vorab bezüglich deren Ablauf beschreiben. Somit ist für kooperative Prozesse eine Kombination beider Technologien sinnvoll, um eine umfassende Assistenzfunktion zu realisieren, die häufig auch als Designflow-Management [SUTTON ET AL. 93] bezeichnet wird.

Wie zuvor motiviert, besitzen Entwurfsprozesse hohe Anteile an Kooperationsbedarf und Flexibilität zu ihrer Unterstützung. So läßt sich in der Regel nicht umfassend vorab definieren, wie ein solcher Prozeß Schritt für Schritt ablaufen soll. Jedoch streben wir eine weitgreifende Systemunterstützung an, um die Koordination von Werkzeuganwendungen (speziell bezüglich Daten) und die Konsistenz im Prozeß zu erhöhen. Die Unterstützung der von uns beschriebenen Prozesse nennen wir *Designflow-Management*. Ein *Designflow* verbindet die zuvor beschriebenen Aspekte von geregelter (Workflow) und ungeregelter (Groupware) Arbeitsunterstützung. Wir wollen nun die wichtigsten Begriffe bezüglich des Designflow-Managements definieren:

Definition **Designflow**
Ein Designflow beschreibt einen Prozeß, der sowohl durch den kreativen und planerischen Charakter als auch dessen ingenieursmäßige Vorgehensweise bestimmt ist. Ein Designflow umfaßt wesentliche Vorgänge und Abhängigkeiten zwischen Schritten, die eine Unterstützung für kooperatives Vorgehen verlangen. Die Kooperation wird auf Basis gemeinsamer Daten und der interaktiven Ablaufsteuerung erreicht. Ziel eines Designflows ist die Erstellung des Entwurfsartefaktes.

Definition **Designflow-Management**
Das Designflow-Management ist die systemgestützte Durchführung von Designflows. Dieses stellt zum einen eine Koordination einzelner Arbeitsschritte zur Verfügung und realisiert andererseits die Bereitstellung von Mechanismen für kooperative Interaktionen.

Definition **Entwurfsartefakt**
Das Entwurfsartefakt (*design artifact*) ist die intellektuelle Leistung, die im Rahmen eines Designflows erstellt wird. Das Entwurfsartefakt ist typischerweise aus einzelnen Entwurfsobjekten zusammengesetzt (*design items*).

Definition **Entwurfsobjekt**
Ein Entwurfsobjekt (*design item*) ist ein abgegrenzter Teil eines Entwurfsartefakts. Entwurfsobjekte können Abhängigkeiten zu weiteren Entwurfsobjekten besitzen.

Definition **Entwurfsziel**
Das Entwurfsziel wird durch eine Spezifikation des Entwurfsartefakts beschrieben.

Ein Designflow besitzt viele Merkmale, wie die Ablaufbeschreibung, die Datensicht, die Akteure, bestimmte (Entwurfs-)Regeln usw., die ähnlich einem Workflow sind. Jedoch besitzt ein Designflow im Gegensatz zu einem Workflow weitergehende Möglichkeiten zur Beschreibung von gemeinsam genutzten Daten und Kooperationsszenarien, wodurch die fehlende Vorabbeschreibbarkeit des Ablaufs eines Entwurfsprozesses ausgeglichen wird.

4.3 ADM-Überblick

In diesem Abschnitt beschreiben wir kurz die Idee unseres Architekturansatzes. Daraufhin werden zusammenfassend die Grundkomponenten des ADM vorgestellt. Diese Grundkomponenten werden in den Abschnitten 4.4, 4.6 und 4.7 vertieft.

4.3.1 Integration von Groupware- und Workflow-Technologie

Um die Anforderung bezüglich der Unterstützung von strukturierten Arbeitsschritten und des Kooperationsbedarfs auf gemeinsamen Objekten zu erfüllen, kombinieren wir bereits existierende Technologien - Workflow-Management-Technologie und Kooperationstechnologie [FRANK 1997].

Ein Designflow basiert auf einem hierarchischen Aktivitätenmodell. Eine Aktivität beschreibt einen abgegrenzten Arbeitsvorgang. Da wir kooperative Prozesse unterstützen, benötigen wir jedoch auch Mechanismen, die es verschiedenen Aktivitäten ermöglichen zu kooperieren. Durch die Einführung eines gemeinsamen Informationsraums mit der Regelung von Objektzugriffen und der Delegation von Aktivitäten mittels Interaktionsprotokollen durchbrechen wir die Isolation von Workflow-typischen Aktivitäten. Diese Funktionalität wird in Abbildung 43 in der Designflow-Schicht zusammengefaßt. Unsere Systemarchitektur basiert auf dem Gedanken, existierende Systeme mit geeigneter Middleware in Form von Basisdiensten zu verbinden: Aktivitäten, Protokollmaschine und gemeinsame Objekte. Diese Basisdienste werden aus bereits existierenden Systemen oder allgemeinen Diensten abgebildet. Letztendlich muß eine Benutzungsschnittstelle die Designflow-Funktionalität geeignet anbieten. Damit erreicht der ASCEND-Ansatz durch die Kombination und Anpassung ausgewählter Technologien eine umfassende Unterstützung der in dieser Arbeit betrachteten Prozesse.

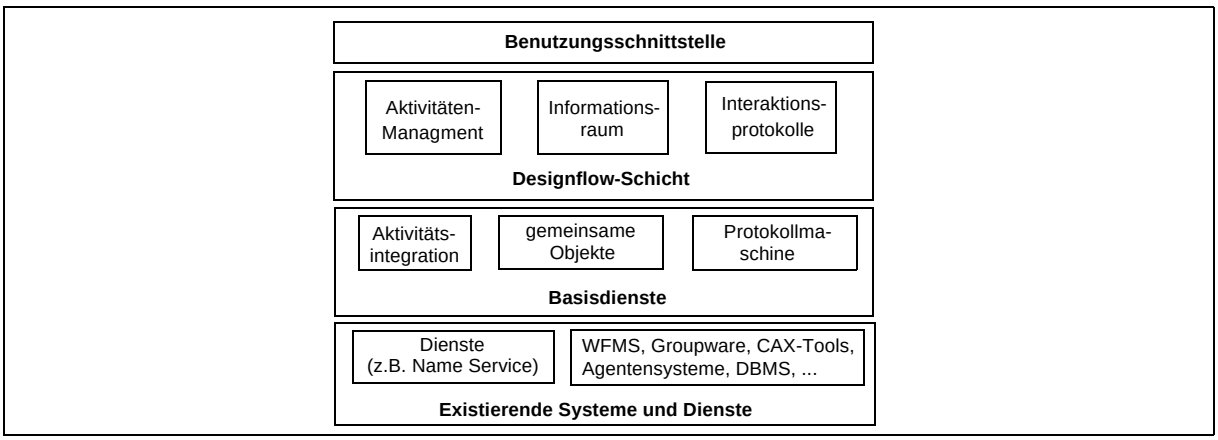


Abbildung 43: Schematischer Realisierungsansatz im ASCEND-Projekt

Die Arbeitsschritte innerhalb des ADM werden durch Aktivitäten dargestellt, die hierarchisch aufgebaut werden können, wie dies auch im Workflow-Management angewendet wird (s. Abschnitt 3.4.3). Dagegen sind ADM-Aktivitäten nicht als Blackbox zu sehen, da zur Laufzeit einer Aktivität Funktionen zur Interaktion verfügbar sind, ähnlich wie es die Modellierung durch Prozeßeinheitselemente vorschlägt (Abschnitt 3.7.1). Aufgrund dieser ’Öffnung’ von

Aktivitäten sind auch entsprechende Koordinationsmechanismen zu überarbeiten, da hier offensichtlich Aktionen innerhalb von Aktivitäten von Bedeutung sind (siehe Abschnitt 4.4). Solche Aktionen betreffen bspw. Datenzugriffe und direkte Interaktionen zwischen Aktivitäten (z.B. eine Delegationsverhandlung). Diese werden durch die Anwendung von Kooperationsprotokollen erreicht.

Neben der Aktivitätsunterstützung bildet die Kooperationsunterstützung einen weiteren wichtigen Pfeiler des ADM. Eine Kooperation erfolgt zum einen aufgrund der gemeinsamen Objekte, die von Aktivitäten genutzt werden können (siehe Abschnitt 4.6). Zum anderen stehen generische Interaktionsprotokolle zur Verfügung, die für direkte Verhandlungen zwischen Aktivitäten, Objekten und Akteuren verwendet werden können (siehe Abschnitt 4.7).

Das ADM selbst besitzt einen Framework-Charakter, aufgrund dessen anwendungsspezifische Implementierungen erstellt werden können. Wir demonstrieren dies anhand der CASSY-Implementierung in Kapitel 5. Die Generizität zeigt sich etwa durch Schnittstellen, die es erlauben verschiedene Workflow- und Groupware-Systeme zu verwenden. Die Realisierung von ADM-Aktivitäten ist auf das jeweilige Anwendungsfeld anpaßbar. Besonders hervorzuheben ist auch die weitgehende Definierbarkeit der Protokollkomponente zur Realisierung verschiedenster Interaktionen und daraus resultierender Kooperationen.

4.3.2 Die Grundkomponenten des ADM

Die drei Hauptbereiche des ADM ermöglichen eine übergreifende Systemunterstützung:

- Informationsraum
  Die Entwurfsdaten stehen im Rahmen eines gemeinsamen Informationsraums zu Verfügung. Auf diesen werden Kooperationen durchgeführt (vgl. auch Abschnitt 4.6). So kann eine weitgehende Systemunterstützung, etwa zur Konsistenzsicherung dieser Daten, erreicht werden. Nicht-Entwurfsobjektdaten und weitere Ressourcen sind auch bedeutend innerhalb eines Entwurfsprozesses. Eine Aufteilung scheint sinnvoll, da sich der Entwurf an den Daten für das Entwurfsobjekt orientiert. Ressourcen, wie etwa Berichte, Lagerlisten oder Maschinen, werden daher in das Modell mit aufgenommen, um auch hierfür eine Systemunterstützung zu gewährleisten. Diese sind damit auch Teil des gemeinsamen Informationsraums.
- Aktivitätenmodell
  Das Aktivitätenmodell koordiniert mit Hilfe der anderen Aspekte den Fortschritt im Entwurf. Im wesentlichen werden hier Aktivitäten und deren Zusammenspiel beschrieben, die somit den Verhaltensaspekt darstellen. Eine Aktivität ist eine Einheit, die eine Tätigkeit repräsentiert und von einem Akteur durchgeführt wird. Zwischen diesen Aktivitäten bestehen Ablaufabhängigkeiten, die deren Abfolge regeln.
- Interaktionsmodell
  Zur Abwicklung von Entscheidungen oder Verhandlungen zur Laufzeit, bspw. bezüglich des konkurrierenden Datenzugriffs, ist ein entsprechendes Kommunikationsmodell zu berücksichtigen. Dieses stellt Protokolle zwischen Interaktionspartnern (Aktivitäten, Objekte, Akteure) zur Verfügung.

Die Organisationsstruktur, die i.w. aus den Personen einer Organisation und deren Beziehungen, Rollen usw. besteht, ist für eine systemgestützte Durchführung von Prozessen wichtig. Jedoch werden wir diesen Aspekt im folgenden nur noch gelegentlich erörtern, da wir uns hier nicht von anderen Ansätzen unterscheiden und daher entsprechende Modelle angewendet werden können (z.B. [JABLONSKI UND BUßLER 1996] und [MUCHITSCH 1998]).

Es existiert ein Rollenmodell auf Basis von Akteuren und deren Fähigkeiten, das für die Aktivitäten verwendet wird. Weiterhin werden den Aktivitäten auch Ressourcen und Entwurfsobjekte zugeordnet, die über einen Informationsraum genutzt werden können. Wir werden dieses Modell im nächsten Abschnitt detailliert vorstellen. Im folgenden bezeichnen wir die Entwurfsobjekte und Ressourcen des Informationsraums, soweit keine Unterscheidung nötig oder der Bezug klar ist, einfach als *Objekte*. Aktivitäten, (Entwurfs-)Objekte und Akteure nennen wir *Entitäten.* Die *Aktivitätsstruktur* bezeichnet schließlich den logischen Aufbau von Aktivitäten, d.h. das Enthaltensein und andere Beziehungen, bspw. in Form eines Baumes.

4.4 Das Aktivitätenmodell

In diesem Abschnitt wird das Aktivitätenmodell des ADM beschrieben. Als Überblick wird das Architekturmodell vorgestellt, das die Beziehungen der einzelnen Bestandteile veranschaulicht. In den nachfolgenden Abschnitten, werden diese Bestandteile eingehend besprochen.

Abbildung 44 zeigt das Architekturmodell für das Aktivitäten-Management. Ein zu realisierender Designflow wird durch die *Designflow-Spezifikation* beschrieben (Abschnitt 4.4.1). Diese Spezifikation ist dynamisch und kann zur Laufzeit geändert werden. Die Definition erfolgt durch eine entsprechende Benutzungsschnittstelle (*Designflow-Definitions-GUI*). Änderungen zur Laufzeit finden durch eine *Client-GUI* statt, da der *Aktivitäten-Management-Koordinator* die Gültigkeit der Änderungen prüfen muß. Die Designflow-Spezifikation beinhaltet die Beschreibung der Aktivitäten, genutzten Ressourcen und Constraints, welche die Ausführungsabhängigkeiten zwischen Aktivitäten beschreiben (Abschnitt 4.4.4). Constraints liegen in einer durch eine *Constraints-Definitions-GUI* modifizierbaren *Constraint-Definition* vor. Das *Constraint-Management* baut eine *Regelbasis* auf, die aus allgemeingültigen Ausführungsregeln (z.B.: eine Aktivität muß gestartet sein, bevor sie beendet werden kann) und den Constraints aus der Designflow-Spezifikation besteht (Abschnitt 4.4.4). Aufgrund dieser Regelbasis kann die Ausführungsreihenfolge von Aktivitäten bestimmt werden. Nach dem Aufruf einer Aktivität werden die von der Aktivität ausgelösten Ereignisse durch das Constraint-Management auf ihre Gültigkeit geprüft. Die Aktivitäten werden durch die Benutzer bearbeitet. Gemeinsam genutzte Objekte werden durch den Informationsraum verwaltet (Abschnitt 4.6). Eine solche Nutzung wird durch die Aktivitäten initiiert. Die Aktivitäten bzw. deren Implementierungen und die Objekte basieren auf entsprechenden externen Anwendungen und deren Funktionalitäten.

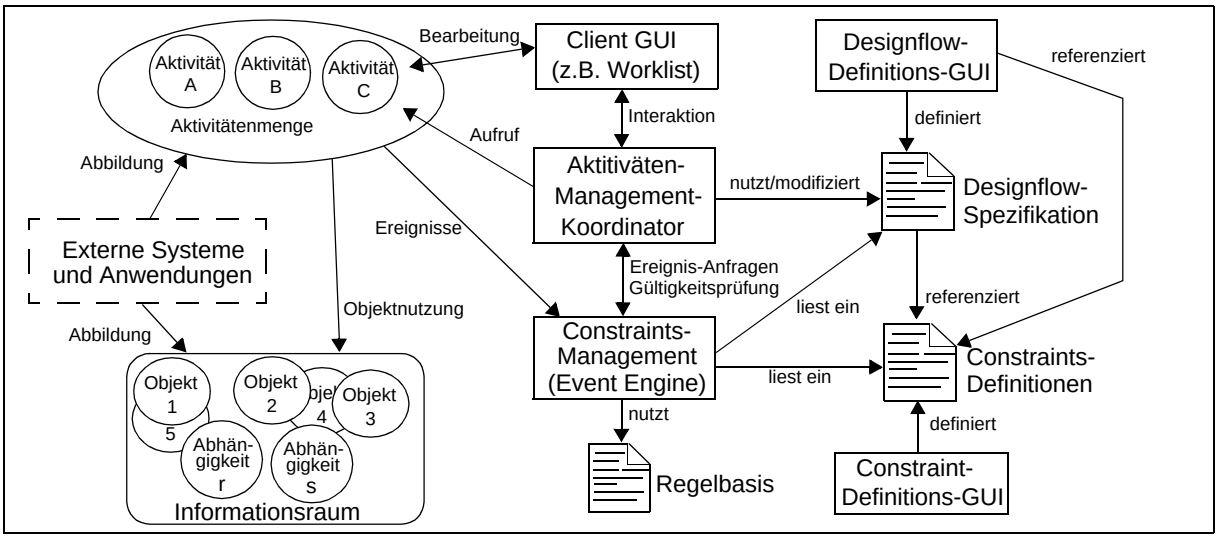


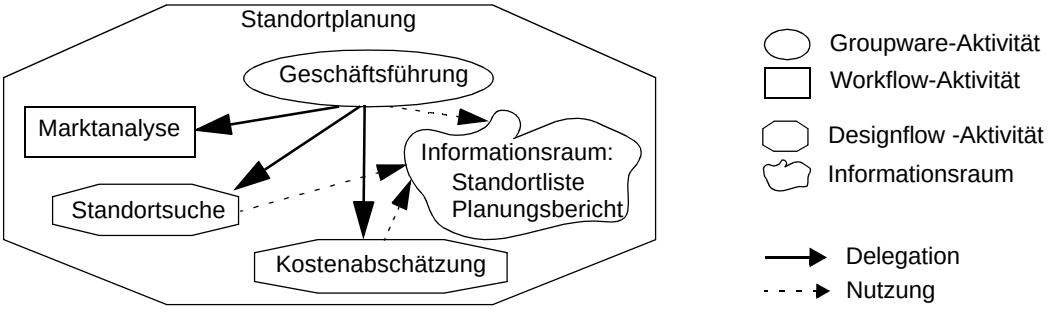
Abbildung 44: Architekturmodell für das ADM-Aktivitäten-Management

Zur Verdeutlichung der Aktivitätsstruktur zeigen wir eine vereinfachte Definition für den rekursiven Aufbau von Designflows. Im Anhang ist eine vollständige Definitionssprache für Designflows zu finden; hier werden die wesentlichen Konzepte zusammengestellt:

primitive-activity	::= design-step | groupware primitive
basic-activity	::= workflow-activity | groupware-activity | designflow-activity | primitive-activity
groupware-activity	::= groupware-primitive
groupware-primitive	::= tele-conference | shared editing | BBS | interview | scheduler | ...
workflow-activity	::= workflow-definition({basic-activity})
workflow-definition	::= workflow-wff ({workflow-primitive}) /* workflow well-formed formula */
workflow-primitive	::= sequential-execution | and | or | ... /*z.B. WfMC-Primitive [WFMC 1995] */
designflow-activity	::= designflow-defintion({designflow-primitive({basic-activity}) })
designflow-primitive	::= negotiation | usage | delegation | constraint

Eine darauf basierende textuelle Designflow-Definition läßt sich weitgehend direkt in eine grafische Darstellung umsetzen, wie es in Abbildung 45 anhand des Planungsszenarios aus Abschnitt 2.3.3 gezeigt wird. Dort wird derselbe Designflow einmal in einer graphischen Repräsentation und einer feiner detaillierten textuellen Definition beschrieben.

Sowohl in graphischer als auch textbasierter Darstellung in Abbildung 45 läßt sich eine Designflow-Aktivität *Standortplanung* erkennen, welche sich aus vier Subaktivitäten und einem gemeinsamen Informationsraum zusammensetzt. Davon sind zwei wiederum Designflow-Aktivitäten (*Standortsuche* und *Kostenabschätzung*). *Marktanalyse* ist eine Workflow-Aktivität und *Geschäftsführung* eine Groupware-Aktivität. Zudem bestehen Abhängigkeiten unterschiedlicher Bedeutung (hier: *delegates*, *uses* und das Constraint *starts_before*) zwischen bestimmten Subaktivitäten. Die Entwurfsobjekte (*Standortliste*, *Planungsbericht*) sind dem Informationsraum zugeordnet. Im folgenden werden die im ADM verwendeten Konstrukte einzeln vorgestellt. In Abschnitt 4.5 werden wir das Beispiel noch weiter aufschlüsseln. Der komplette Designflow ist in Anhang C zu finden.




```
designflow-activity: KOSTENABSCHÄTZUNG(..)

designflow-activity: STANDORTSUCHE(..)

workflow-activity: MARKTANALYSE(..)

groupware-activity: GESCHÄFTSFÜHRUNG(..)

designflow-activity: STANDORTPLANUNG(
   RESOURCES:
      Standortliste
   DESIGN-OBJECTS:
      Planungsbericht
   ACTORS:
      Planungsgruppe, Geschäftsführung

   GOAL:
      Planungsbericht.fertig == TRUE
   ACTIVITIES:
      MARKTANALYSE:          M,
      STANDORTSUCHE:         S,
      KOSTENABSCHÄTZUNG:     K,
      GESCHÄFTSFÜHRUNG:      G,
   DESIGN_PRIMITIVES:
      M starts_before {S, K},
      G delegates {M, S, K},
      G uses Standortliste,
      G uses Planungsbericht,
      K uses Planungsbericht,
      K uses Standortliste,
      S uses Standortliste
)
```

Abbildung 45: Vereinfachte grafische und textuelle Designflow-Spezifikation

4.4.1 Elemente der Designflow-Spezifikation

Wie in der semiformalen Definition zu sehen ist, berücksichtigt das ADM verschiedene Aktivitätstypen, die zur Bildung eines Designflows eingesetzt werden können. Diese lassen sich in Aktivitäten und Primitive unterteilen, die im folgenden beschrieben werden.

4.4.1.1 Primitive Aktivitäten

Die primitive Aktivität ist die einfachste Form einer Aktivität des ADM. Sie ist die grundlegende Einheit für den Aufruf von Funktionen oder Applikationen, die zur Durchführung eines Prozeßschritts notwendig sind. Man kann sie auch atomare Aktivität nennen, da sie eine abgeschlossene Funktionalität repräsentieren, nämlich ein *Primitiv*. Ein Primitiv ist entweder ein einzelner Entwurfsschritt (design step) oder ein Groupware-Primitiv (s.u.). Beide sind nicht weiter zerlegbar. Eine primitive Aktivität wird durch folgende Definition beschrieben:

```
primitive_activity: Activity_Name(
   [RESOURCES: obj_type Obj_Name {,obj_type Obj_Name}*]
   [DESIGN-OBJECTS: obj_type Obj_Name {,obj_type Obj_Name}*]
   ACTORS: automatic | actor_type Actor_Name {,actor_type Actor_Name}*
   GOAL: goal_definition
   [APPLICATION: Program_Call | Facility_Call ]
)
```

`RESOURCES` spezifiziert die verwendbaren Ressourcen (Daten, Maschinen, Programme, etc.) innerhalb der Aktivität. Sie können auch durch den gemeinsamen Informationsraum genutzt werden, der anderen Aktivitäten zugänglich ist. Auf den enthaltenen Daten können Nutzungsbeziehungen definiert werden, um so verschiedene Aktivitäten über diesen kooperieren zu lassen. Die Entwurfsobjekte (`DESIGN-OBJECTS`) werden immer im Informationsraum verwaltet und daher als eigenständiger Bestandteil einer Aktivität betrachtet. `ACTORS` beschreibt die betroffenen Akteure, wie einzelne Personen, Gruppen, Maschinen und automatische Programmausführung. Diese können auch durch Eigenschaften definiert werden, die erst zur Laufzeit aufgelöst werden. Durch `GOAL` werden die Ziele dieser Aktivität beschrieben. Dies kann rein textuell geschehen oder auch mit Bedingungen, die sich bspw. auf das Entwurfsobjekt beziehen (z.B. `document.finished == true`). Die Aktivität kann ein Programm zu deren Durchführung vorschreiben, das mittels `APPLICATION` spezifiziert wird. Wir unterscheiden Applikationen nach einfachen Werkzeug- oder Programmaufrufen (z.B. Schaltungssimulator, Textverarbeitung) und Facility-Aufrufen (z.B. Workflow-System).

4.4.1.2 Groupware-Primitiv und Design-Step

Ein Groupware-Primitiv stellt die Verwendung einer atomaren Anwendung von Groupware dar, wie etwa der Aufruf eines Tele-Konferenz-Systems. Ein Design-Step repräsentiert dagegen eine primitive und nicht weiter zerlegte Aktivität (auf einem Entwurfsobjekt) wie etwa eine spezifische Werkzeuganwendung. Ein Design-Step ist der Aufruf einer Funktion oder Anwendung oder auch nur die textuelle Beschreibung einer einzelnen Tätigkeit. Beispielsweise kann hier das Testen einer elektrischen Schaltung mittels eines Simulator-Programmes festgelegt werden (z.B. `PROGRAM: spice-simulator`). Im Unterschied zum Groupware-Primitiv erlaubt ein Design-Step die Protokollierung dieses Vorgangs und damit auch eine direkte Systemunterstützung, wie etwa das Zurücksetzen des Arbeitsschritts.

4.4.1.3 Workflow- und Designflow-Primitiv

Im Gegensatz zu den obigen Primitiven beschreiben Workflow- und Designflow-Primitive keine aktiven Schritte. Workflow-Primitive sind u.a. Kontrollflußkonstrukte und Datenflußdefinition (siehe Abschnitt 3.4.3.1), die genutzt werden, um Workflows aus einer Menge von Aktivitäten heraus aufzubauen.

Designflow-Primitive stellen die Ablaufabhängigkeiten (z.B. `A starts_before B`) und Kooperationsmuster (Verhandlung, Delegation) dar. Dadurch können Kontroll- und Datenflußaspekte und auch Kooperationen beschrieben werden. Im einzelnen sind dies:

- Delegation
  Die Delegation erlaubt es, abhängige Aktivitäten zu erzeugen, die einen Teilentwurf zu erfüllen haben. Zur Behandlung einer Delegation wird ein Protokoll benötigt, das die Übergabe einer Zielspezifikation erlaubt. Die Spezifikation beschreibt die Aufgabe der delegierten Aktivität. Diese Spezifikation ist durch eine Verhandlung anpaßbar (bspw. bei geänderten Gegebenheiten oder Unerfüllbarkeit). Letztendlich muß auch die Annahme der Entwurfsergebnisse beschlossen werden, besonders auch im Hinblick auf eine erfolgreiche Integration mehrerer Teilentwürfe (siehe Abbildung 4 auf Seite 24).
- Nutzung
  Die Nutzungsbeziehung beschreibt die Verwendung von Objekten. Dadurch können Datenabhängigkeiten einer Aktivität modelliert werden und somit auch explizit durch das System unterstützt werden. Die Durchführung von Operationen auf solchen Objekten ist abhängig von deren Typ (Datenbankdaten, Dokumente, Maschinen, ...) und der synchronen Nutzung durch andere Aktivitäten. Daher sind hier anpaßbare Protokolle notwendig, um verschiedene Szenarien unterstützen zu können.
- Verhandlung
  Die allgemeine Verhandlung wird bereitgestellt, da ein gewisser Abstimmungs- und Interaktionsbedarf in Entwurfsprozessen besteht. Diese müssen verschiedene Arten der Interaktion unterstützen, die abhängig von der Art der Teilnehmer (Ressourcen, Aktivitäten, Akteure), deren Anzahl und deren Inhalt sind.
- Constraints
  Constraints erlauben eine variablere Beschreibung von Ablaufabhängigkeiten als Workflow-Kontrollflußkonstrukte (bspw. Aktivität A oder B muß ausgeführt werden, siehe etwa [ATTIE ET AL. 1993]). Außerdem setzen wir diese auch zur Beschreibung von Datenabhängigkeiten ein (z.B. Ressource R darf allein von Aktivität A genutzt werden, während A aktiv ist).

Wie man sieht, stellen Protokolle einen wesentlichen Aspekt bei der Nutzung der Beziehungen dar. Wir werden diese in Abschnitt 4.7 detailliert besprechen. Durch die Designflow-Beziehungen, Constraints und die Protokolle wird eine große Flexibilität zur Unterstützung von Entwurfsaufgaben realisiert, wie sie in Abschnitt 2.5 gefordert wird.

4.4.1.4 Basis-Aktivität

Alle Aktivitäten unseres Aktivitätenmodells sind auch sog. *basic activities*, um einen rekursiven Aufbau von Designflows zu ermöglichen. Sie können jedoch nicht in einer Groupware-Aktivität verwendet werden, wie in Abschnitt 4.4.1.6 erklärt. Eine Basis-Aktivität wird durch folgende Parameter beschrieben:

```
basic-activity. Activity_Name(
    [RESOURCES: obj_type Obj_Name {,obj_type Obj_Name}*]
    [DESIGN-OBJECTS: obj_type Obj_Name {,obj_type Obj_Name}*]
    ACTORS: automatic | actor_type Actor_Name {,actor_type Actor_Name}*
    GOAL: goal_definition
    [ACTIVITIES: Activity_Name local_Activity_Name {Activity_Name local_Activity_Name}*]
    [APPLICATION: Program_Call | Facility_Call ]
)
```

Typischerweise enthält eine Aktivität, wenn sie keine primitive Aktivität ist, Subaktivitäten, welche durch `ACTIVITIES` referenziert werden. Ansonsten stimmt die Beschreibung mit der primitiver Aktivitäten überein.

4.4.1.5 Workflow-Aktivität

Workflows kapseln Prozesse, die durch die bekannten Workflow-Konstrukte und Workflow-Funktionalitäten unterstützt werden können. Ein Workflow beinhaltet Aktivitäten, die durch Workflow-Primitive (sequentielle Ausführung, bedingte Verzweigung, etc. [WFMC 1995]) verbunden sind. Aktivitäten können auch Aufrufe weiterer Workflow-, Groupware- und Designflow-Aktivitäten sein. Eine Workflow-Aktivität garantiert eine korrekte Ausführung des spezifizierten Workflows. Dieser Workflow wird durch `APPLICATION:` Facility_Call festgelegt. Facility_Call beschreibt die durchführende Workflow-Facility bzw. das WFMS und den zu verwendenden Workflow (siehe Abschnitt 4.5). Die Aktivitäten des Workflows können über `ACTIVITIES` referenziert werden. Die `WF_PRIMITIVES` (Workflow-Primitive, siehe oben) beschreiben bspw. Kontroll- und Datenfluß:

```
workflow_activity: Activity_Name(
   [RESOURCES: obj_type Obj_Name {,obj_type Obj_Name}*]
   [DESIGN-OBJECTS: obj_type Obj_Name {,obj_type Obj_Name}*]
   ACTORS: automatic | actor_type Actor_Name {,actor_type Actor_Name}*
   GOAL: goal_definition
   [ACTIVITIES: Activity_Name local_Activity_Name {Activity_Name local_Activity_Name}*]
   APPLICATION: Facility_Call
   [WF_PRIMITIVES: WF_primitive {, WF_primitive}]
)
```

4.4.1.6 Groupware-Aktivität

Eine Groupware-Aktivität ist eine einzelne kooperative Sitzung, wie z.B. eine Tele-Konferenz oder die Festlegung eines gemeinsamen Termins (*Scheduling*). Da eine solche Aktivität im allgemeinen weniger strukturiert ist als ein Workflow, kann sie nicht rekursiv beschrieben werden und besitzt auch keine Subaktivitäten. Es wird nur ein einzelnes Groupware-Primitiv zur Definition einer solchen Sitzung verwendet, das durch `APPLICATION` bestimmt wird. Somit können beliebige Groupware-Systeme eingebunden werden. Diese können auf die anderen Bestandteile des Modells, wie etwa Entwurfsobjekte, zugreifen.

```
groupware_activity: Activity_Name(
   [RESOURCES: obj_type Obj_Name {,obj_type Obj_Name}*]
   [DESIGN-OBJECTS: obj_type Obj_Name {,obj_type Obj_Name}*]
   ACTORS: automatic | actor_type Actor_Name {,actor_type Actor_Name}*
   GOAL: goal_definition
   APPLICATION: Program_Call | Facility_Call
   [DESIGN_PRIMITIVES: design_primitive {, design_primitive}]
)
```

4.4.1.7 Designflow-Aktivität

Die Designflow-Aktivität kann einen (komplexen) Entwurf beschreiben, der typischerweise aus verschiedenen Aktivitätstypen rekursiv zusammengesetzt ist: Workflow-Aktivitäten, Groupware-Aktivitäten, Primitive und natürlich auch weitere Designflow-Aktivitäten. Innerhalb einer Designflow-Aktivität werden die Designflow-Primitive wie Delegation, Objektnutzung und Constraints unterstützt. Die Syntax zur Definition lautet wie folgt:

```
designflow_activity: Activity_Name(
   [RESOURCES: obj_type Obj_Name {,obj_type Obj_Name}*]
   [DESIGN-OBJECTS: obj_type Obj_Name {,obj_type Obj_Name}*]
   ACTORS: automatic | actor_type Actor_Name {,actor_type Actor_Name}*
   GOAL: goal_definition
   [ACTIVITIES: Activity_Name local_Activity_Name {Activity_Name local_Activity_Name}*]
   [DESIGN_PRIMITIVES: DF_primitive {, DF_primitive}]
)
```

Die Designflow-Primitive werden innerhalb der `DESIGN_PRIMITIVES` spezifiziert und können in Constraints und Beziehungen unterschieden werden. Beispiele für Constraints sind `Standortsuche starts_before Kostenabschätzung` oder `Kostenabschätzung X_needs Standortliste`. Die Benutzung von Constraints erlaubt einen höheren Automatisierungs-

grad, da sie durch das System überprüft werden können (siehe Abschnitt 4.4.4). Die Beziehungen beschreiben die Delegation von Aktivitäten und die Nutzung von Objekten des Informationsraums.

4.4.2 Aktivitätenstrukturierung

Die ADM-Aktivitäten lassen sich rekursiv zu komplexen Designflows aufbauen wie auch Workflow-Aktivitäten. So erhält man Sub-Designflows in denen weitere Aktivitäten enthalten sind. Dies fördert die Übersichtlichkeit und den strukturierten Aufbau von Designflows. Zudem können Wiederverwendbarkeitsvorteile erzielt werden, etwa durch das Benutzen von Aktivitäten als wiederverwendbare Schablonen.

Bei dem Aufbau eines Designflows ist dabei zu beachten, daß die Delegation auch ein Strukturierungselement darstellt und sehr ähnlich der hierarchischen Schachtelung von Aktivitäten ist. Die Delegation bietet jedoch weitergehende Merkmale, wie die Verhandlung über Entwurfsziele (siehe Abschnitt 4.7), die eine enge Kopplung der betreffenden Aktivitäten als vorgeplantes Kooperationsmuster erreicht.

4.4.3 Starten einer Aktivität

Die Laufzeitphase kann in das Abarbeiten von Designflows und das Überwachen dieser Abarbeitung unterteilt werden, wie dies auch bei Workflows der Fall ist [JABLONSKI 1995]. Die Abarbeitung selbst erfolgt durch die Ausführung der Aktivitäten. Die Überwachung beinhaltet das Starten und Beenden von Aktivitäten, die Durchsetzung von Ablaufabhängigkeiten, die Unterstützung von Kooperationsprotokollen und das Bereitstellen von Ressourcen. Diese Überwachung ist durch einen sogenannten *Aktivitäten-Management-Koordinator* zu leisten.

Wir verwenden ein Modell zur Laufzeitsteuerung, das zunächst sehr ähnlich einem Workflow-System ist. Dies bezieht sich vor allem auf Aktivitäten, deren Startbedingungen evaluiert werden, um diese gegebenenfalls zu starten. Dazu gehört auch die Versorgung der Aktivitäten mit Akteuren und Ressourcen. Es werden sogenannte Worklisten bereitgestellt, die einem Benutzer die zu bearbeitenden Aktivitäten anbieten.

Durch den stärkeren Einfluß eines Akteurs bezüglich der Durchführung und durch die gewünschte Awareness bzw. Kooperation und Interaktion müssen weitere Kontextinformationen für einen Benutzer verfügbar und änderbar sein. So muß ein Akteur wissen, an welchen Aufgaben und mit welchen Ressourcen seine “Kollegen” arbeiten. Dies ist durch eine gemeinsame Sicht auf den Designflow und die Etablierung des Informationsraums zu erreichen.

Eine Aktivität ist initialisiert, wenn die Bedingungen für einen Start erfüllt werden können, wenigstens ein potentieller Akteur gefunden wurde und die Aktivität auf einer Worklist erscheint. Gestartet ist eine Aktivität sobald der erste Akteur die Aktivität in der Worklist zum Start auswählt, oder diese (etwa durch ein Programm) automatisch durchgeführt werden kann (Abbildung 46). Wir unterscheiden folgende drei Kategorien von Startbedingungen:

1. Verfügbarkeit der benötigten Ressourcen für die Ausführung (z.B. eine Maschine muß frei sein, damit man mit ihr arbeiten kann),
2. andere Abhängigkeiten, die etwa durch Design-Constraints beschrieben wurden (z.B. Aktivität A darf erst nach Start von Aktivitäten B und C gestartet werden) und
3. Delegationsbeziehungen.

In der Abbildung 46 wurde die Überprüfung von Constraints und Delegationen zusammengezogen, da man die Delegationsbedingungen als eine Untermenge der Constraints verstehen kann.

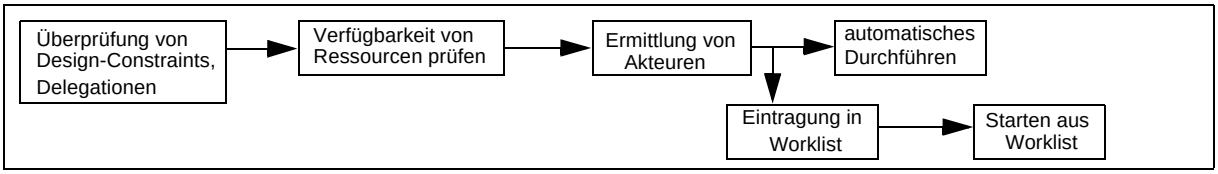


Abbildung 46: Ablauf zum Start einer Aktivität

4.4.4 Ablaufabhängigkeitsbeschreibung durch Constraints

Wir betrachten in diesem Abschnitt den Ablauf eines Designflows als eine Menge von Ausführungen von Aktivitäten. Im Rahmen der vorgegebenen (oder auch fortgeschriebenen) Designflow-Spezifikation, muß das System in der Lage sein, zu ermitteln, welche Aktivitäten ausgeführt werden können. Wir werden dazu im folgenden Constraints einführen, die ein flexible Durchführung der Ausführungsreihenfolge von Aktivitäten gestatten.

Es existiert eine *Reihenfolgenabhängigkeit* zwischen A und B, wenn B in seiner Ausführung abhängig von A ist und erst nach dessen Beendigung gestartet wird. Die Reihenfolgeabhängigkeit ist eine wesentliche Basis zur Beschreibung von Kontrollfluß. Beim Workflow Management wird diese Beziehung durch das serielle Ausführungskonstrukt realisiert. Darauf aufbauend gibt es eine Menge weiterer Ablaufkonstrukte, wie bereits in Abbildung 29 auf Seite 55 dargestellt. Im Rahmen der Integration von Workflow-Management in das ADM übernehmen wir diese Konstrukte in Workflow-Aktivitäten.

Die Delegation kann auch als Kontrollflußkonstrukt betrachtet werden, da die Ausführung einer delegierten Aktivität veranlaßt wird. Jedoch kommen hier zusätzlich die Funktionalität der Spezifikation des Ergebnisses und entsprechende Abhängigkeiten hinzu. Dadurch wird eine andere Bedingung bezüglich der Beendigung der delegierten Aktivität erreicht, da die Spezifikation, die bei einer Delegation übergeben wird, erfüllt werden muß. Die Spezifikation ist verhandelbar, weswegen die beteiligten Aktivitäten zeitgleich ablaufen müssen. Es findet also eine Interaktion zwischen delegierender und delegierter Aktivität statt. Damit ist die Delegation keine Kontrollflußübergabe, sondern eher mit der Erzeugung von *Threads* oder Kindprozessen zu vergleichen.

Aus der Struktur einer Designflow-Definition ergeben sich bereits die ersten Bedingungen für startbare Aktivitäten. Betrachten wir für die Definition einen Graphen mit den Aktivitäten als Knoten und den Kanten *enthalten* und *Delegation* (Abbildung 47). Enthalten-Kanten beschreiben die hierarchische Schachtelung von Aktivitäten. Daraus ergibt sich, daß Kindaktivitäten frühestens gestartet werden können, wenn bereits ihre Elternaktivitäten gestartet wurden. Hier wird auch die Verwandschaft zwischen dem Enthaltensein von Aktivitäten und der Delegation offensichtlich. Die Beendigung einer Elternaktivität bedingt auch die Beendigung ihrer Kindaktivitäten. Die Strukturierungsbeziehungen ’enthalten’ und die Delegation besitzen bereits implizit Bedingungen für die Ausführungsreihenfolge. In Abbildung 47 ergeben sich damit u.a. folgende gültigen Startreihenfolgen: ABCDEFGH, AEBCDEFGH, ABDCFGHE, ABDCFHGE, AEBDFCH usw.

Diese potentiellen Ausführungsreihenfolgen sind uns zu ungenau für die Modellierung von Prozessen. Es fehlen hier Möglichkeiten um Abhängigkeiten zwischen Aktivitäten darzustellen. Daher benötigen wir weitere Bedingungen zur Beschreibung der Abhängigkeiten, wie sie durch Constraints realisiert werden können.

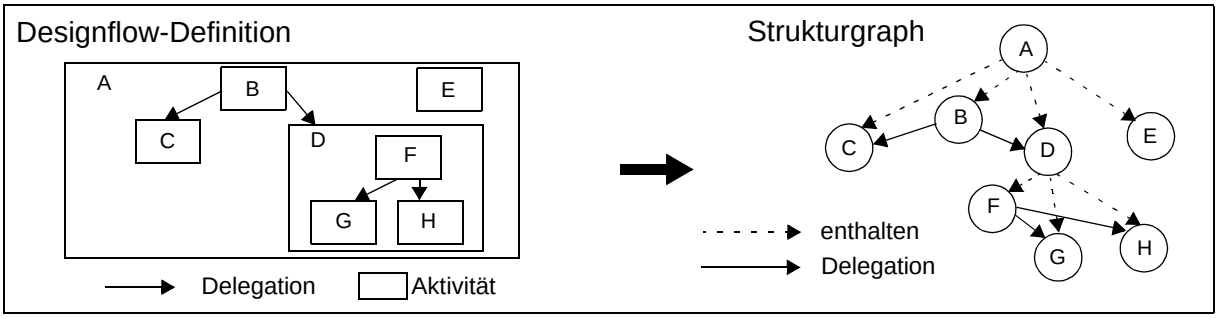


Abbildung 47: Überführung einer Designflow-Definition in einen Strukturgraphen

4.4.4.1 Constraints zur Modellierung von Prozeßabhängigkeiten

Durch Constraints schränken wir die möglichen Ausführungsreihenfolgen von Aktivitäten durch Abhängigkeiten zwischen den Aktivitäten ein. Dadurch werden entwurfsbedingte Abhängigkeiten (z.B. Layout einer Schaltung vor deren Simulation) unterstützt. Weiterhin sind diese Einschränkungen flexibel genug um den Teilnehmern eines Entwurfsprozesses eine geeignete Wahl der nächsten Arbeitsschritte zu überlassen. Die vorgestellten Constraints können ’restriktiv’ angewendet werden und streng sequentialisierte Prozesse nachbilden; es können aber auch große Freiheiten gestattet werden.

Mit Constraints lassen sich Ablaufabhängigkeiten deskriptiv beschreiben. Da verschiedene Ausführungsreihenfolgen bestehen, kann ein Benutzer seinen nächsten Schritt bestimmen, wobei das Steuerungssystem nur dessen Gültigkeit aufgrund der vorgegebenen Bedingungen überprüft. Man kann folgende Arten von deskriptiven Konstrukten unterscheiden:

- Zeitliche Limitierung
  Bsp.: A darf nicht mehr begonnen werden, sobald B angefangen hat.
- Verzögerung
  A darf nur ausgeführt werden, wenn B bereits ausgeführt wurde oder feststeht, daß B ausgeführt werden wird.
- Existenzabhängigkeit
  A darf nur ausgeführt werden, wenn B bereits ausgeführt wurde oder feststeht, daß B ausgeführt werden wird.

Für das ADM stellen die Constraints aus Tabelle 3 eine Basis für eine sinnvolle Modellierung von Designflows dar, jedoch soll dies keine abgeschlossene Menge darstellen. Vielmehr kann diese Menge für bestimmte Anwendungsszenarien angepaßt und erweitert werden (siehe nächster Abschnitt). Beispielsweise können bei Bedarf konkrete zeitliche Beschränkungen hinzugefügt werden (A muß um 9:00 Uhr gestartet werden, B darf maximal 30 Minuten durchgeführt werden) oder die Verwendung von Ressourcen spezifiziert werden.

Constraint	Beschreibung
A LIMITS B	entspricht der zeitlichen Limitierung
A ENABLES B	entspricht der Verzögerung
A IMPLIES B	entspricht der Existenzabhängigkeit
A DIRECTLYAFTER B	Unmittelbar nach der Ausführung von B muß A ausgeführt werden. Dies entspricht im wesentlichen der sequentiellen Ausführung.
A AFTER B	Irgendwann nach der Beendigung von B muß auch A ausgeführt.
A OR B	Mindestens eine der beiden Aktivitäten muß ausgeführt werden.
A XOR B	Entweder A oder B muß ausgeführt werden, aber nicht beide.
A MUST_APPEAR	A muß ausgeführt werden.
A EXCLUDES B	Die Ausführung von A verhindert eine Ausführung von B.
A X-NEEDS R	A benötigt zur Durchführung exklusiven Zugriff auf die Ressource R.

Tabelle 3: ADM-Constraints

4.4.4.2 Ereignisorientierte Beschreibung von Ablaufabhängigkeiten

Im ADM sind zwei wichtige Aspekte bei den Ablaufabhängigkeiten zu berücksichtigen. Zum einen ist das Granulat für Abhängigkeiten nicht eine atomare Aktivität, da die Aktivitäten des ADM verschränkt ablaufen und währendessen gegenseitige Interaktionen beachten werden müssen (beispielsweise innerhalb einer Delegation oder Objektzugriffe). Daher ist es wichtig, im Gegensatz zu den herkömmlichen Verfahren zur Beschreibung von Abhängigkeiten, diese Möglichkeiten von ADM-Aktivitäten zu berücksichtigen. Zum anderen setzt der Wunsch nach anwendungsspezifischen Constraints auch eine entsprechende Definierbarkeit voraus. Beides kann durch die Anwendung feingranularer Ereignisse erreicht werden, aus denen umfassende Constraint-Ausdrücke gebildet werden.

Die Aktivitäten sind im ADM nicht atomar, daher ist die Semantik von bspw. *A AFTER B* genauer festzulegen. In diesem Falle: entweder A kann nur nach dem Start von B gestartet werden oder nur nach der Beendigung von B. Dies ist etwa im Rahmen einer Delegation wichtig, da hier parallel Aktivitäten delegiert werden können, und es vielleicht notwendig ist, daß eine dieser Aktivitäten vor einer anderen (delegierten) Aktivität beginnen muß. Jedoch können beide parallel ablaufen (dann z.B. durch ein zusätzliches Constraint *A WHILE B*). Daher folgt die Überlegung das Verhalten von Aktivitäten etwas feiner zu modellieren und aufgrund der so entstehenden Primitive die Constraints zu definieren. Diese Primitive ermöglichen zudem die anwendungsspezifische Erstellung von Constraints.

Zur Definition von Constraints bietet sich die Anwendung von Aktivitätenereignissen an, wie *start(A)* im Falle des Startens einer Aktivität A. Diese Ereignisse werden ausgelöst, wenn entsprechende Zustandsübergänge stattfinden. Eine zentrale Instanz kann die aufgetretenen Ereignisse protokollieren und daher entscheiden, welche aktuellen Zustände existieren. Durch die Bildung von Ausdrücken auf Basis der Ereignisse können Constraints gebildet werden, wie etwa eine einfache Reihenfolgebeziehung von A und B: *start(B) AND terminate(A)*. Solche Ausdrücke können im Sinne eines reaktiven Aktivitätenmanagements eingesetzt werden, das seine Regelbasis aus Constraints vorhält und diese gegen eintreffende Ereignisse abgleicht und nur gültige Ereignisse eintreten läßt. Durch die Basis-Ereignisse in Tabelle 4 können die von uns gewünschten Constraints definiert werden:

Ereignis	Beschreibung
s(A)	Aktivität A wurde gestartet (start)
t(A)	Aktivität A wurde terminiert (terminate)
c(A)	Aktivität A wurde abgebrochen (cancel)
p(A)	Aktivität A pausiert (pause)
u(A)	Aktivität A wurde wiederaufgenommen (un-pause)
r(A)	Aktivität A wurde zurückgesetzt (reset)
i(A)	Aktivität A wurde initialisiert (initialize)
b(R,A)	Ressource R steht derzeit nur Aktivität A zur Verfügung (busy)
r(R,A)	Die Ressource R wurde von Aktivität A freigegeben (ready)

Tabelle 4: Ereignisse für Constraints

Tritt ein solches Ereignis ein, so wird es in eine Historie aufgenommen, und wir nennen es dann ein *Fakt*. Aus diesen Fakten lassen sich weitere Fakten ableiten, wie etwa Aktivität A wird derzeit ausgeführt: *s(A) AND NOT(t(A) OR c(A))*. Um daraus nun Constraints zu bilden, existieren verschiedene Ansätze. [KLEIN 1991] nutzt folgende Primitive zu Definition zwischen Ereignissen:

- $e < f$
  Wenn die Ereignisse *e* und *f* eintreten, dann muß *e* vor *f* eintreten.
- $e \rightarrow f$
  Wenn Ereignis *e* eintritt, dann muß auch *f* eintreten. Die Reihenfolge ist damit jedoch nicht festgelegt (es muß nur zugesichert werde, daß dann beide eintreten).

Diese Primitive sind auch die Grundlage der Abhängigkeitsverwaltung in [GÜNTHÖR 1996]. Dort werden zusätzlich sogenannte Pseudoereignisse genutzt um Garantien für das (zukünftige) Eintreten bzw. Nicht-Eintreten von Ereignissen zu beschreiben, die i.w. den Wächtern (*guards*) in [SINGH 1997A] und [SINGH 1997B] entsprechen. In diesen letzten beiden Publikationen wird ein ähnliches Modell beschrieben, das zusätzlich Variablen in den Ereignissen behandelt. Diese Variablen erlauben es weitergehende Merkmale für Ereignisse auf einfache Weise zu spezifizieren, wenn auch die Mächtigkeit gegenüber [GÜNTHÖR 1996] nicht direkt erweitert wird.

Aufgrund der Komplexität zur Behandlung der Wächter und der anderen Nachteile [WAGNER 2001], setzen wir lediglich Ansätze aus [SINGH 1997A] ein. Wir verwenden zur Definition von Constraints die folgende Darstellung um Ausdrücke zu formen (eine umfassende Herleitung und Definition findet sich in [WAGNER 2001]):

- *e*(*params*)
  Ein Ereignis *e* mit den Parametern *params*
- *params*: {id = *var*}*
  Die Parameter sind eine Menge von Identifikatoren, denen eine Variable zugewiesen werden kann. Diese Variablen können bei der Abhängigkeitsabarbeitung gebunden werden. Ein Beispiel ist *b(aktivität* = Standortplanung, *ressource* = Standortliste*)*, das die Verwendung der Ressource Standortliste durch die Aktivität Standortplanung signalisiert.
- $e \text{—} f$
  Dies beschreibt die sequentielle Folge der Ereignisse *e* und *f*. D.h., nach Ereignis *e* tritt Ereignis *f* ein.
- $e \wedge f$
  Die Ereignisse *e* und *f* treten beide ein (logisches Und).

- $e \vee f$
  Wenigstens eines der beiden Ereignisse *e* oder *f* tritt ein (logisches Oder).
- $!f$
  Das Ereignis *f* tritt nicht ein.

Diese Grundelemente können zu komplexen Ausdrücken kombiniert werden. Die Operatoren von Klein (siehe oben) können damit auch beschrieben werden: $e < f$ entspricht $!e \vee !f \vee e — f$ und $e \rightarrow f$ entspricht $!e \vee f$. Wir können nun parametrisierte Constraints (siehe Tabelle 5) beschreiben um sie für Designflow-Definitionen zu verwenden (z.B. `starts_before` in Abbildung 45). Dazu sind einige Beispiele in Tabelle 5 aufgeführt. Eine Liste der in dem Prototyp CASSY verwendeten Constraints befindet sich in Anhang D.

Constraint	Beschreibung	Definition
A AFTER B	A darf erst nach der Ausführung von B (auch im Falle des Abbruchs) gestartet werden.	(s(B) - t(B) -s(A)) v (s(B) - c(B) - s(A) v !s(A)
A WHILE B	A darf nur während der Laufzeit von B ausgeführt werden.	(s(B) - s(A) - c(A) - c(B)) v (s(B) - s(A) - t(A) - c(B)) v (s(B) - s(A) - t(A) - t(B)) v (s(B) - s(A) - c(A) - t(B)) v !s(A)
A IMPLIES B	Wenn die Aktivität A terminiert, dann muß auch die Aktivität B terminieren.	t(B) v !t(A)
A DISABLES B	Sobald die erste Aktivität erfolgreich beendet wurde, darf die zweite Aktivität nicht mehr gestartet werden.	(s(B) -t(B) - s(A)) v (s(A) - c(A)-s(B)) v !s(A) v !s(B)
A X_NEEDS R	A benötigt zum Start exklusiven Zugriff auf die Ressource R.	(s(A) - b(R,$X)) v (r(R,$X) - s(A)) v !s(A)
A STARTS_BEFORE B	Wenn A ausgeführt werden soll, so muß A vor B gestartet werden.	(s(A) - s(B)) v !s(A) v !s(B)

Tabelle 5: Beispiele für Constraint-Definitionen

Die Benutzung der Variablen für Ereignisse erlaubt auch die Einführung von zeitlichen Bedingungen. Wenn die Ereignisse mit Zeitstempeln versehen werden, können somit entsprechende Einschränkungen ermöglicht werden, wie zum Beispiel, daß die Aktivität A maximal 30 Minuten ausgeführt werden darf (vereinfacht: $s(A,time) \wedge t(A,time+30)$). Als Folge lassen sich damit auch Schätzungen der Dauer von Prozessen berechnen, die im Rahmen von bspw. Netzplänen wichtige Hinweise auf die Durchführung und Planung einer Arbeit liefern. Wir führen diese hier jedoch nicht aus.

4.4.4.3 Abarbeitung von Constraints

Im Falle einer Designflow-Ausführung werden die Constraints geladen und die Variablen, soweit bekannt (s. u.), mit den entsprechenden Instanzen (Aktivitäten und Ressourcen) gebunden. Somit entsteht ein System von erlaubten Ereignisfolgen, die durch das Aktivitätenmanagement abgearbeitet wird. Das Starten einer Aktivität kann nun aufgrund der oben angegebenen Kontrollflußbeschreibungen geregelt werden. Das heißt das System kann ermitteln, welche Aktivitäten ausgeführt werden dürften und für welche die Bedingungen (noch) nicht genügen.

Betrachten wir beispielsweise eine allgemeingültige Regel zur korrekten Durchführung einer Aktivität, wobei wir zur Vereinfachung nur die Ereignisse s(), c() und t() in Betracht ziehen:

$$(i(a) - s(a) - c(a) \vee i(a) - s(a) - t(a)) \wedge (!c(a) \vee !t(a))$$

Diese Regel bestimmt, daß Aktivitäten zuerst initialisiert und gestartet werden und danach terminiert oder abgebrochen werden können. Der letzte Teilausdruck legt zusätzlich fest, daß entweder Abbruch oder Terminierung eintreten muß. Alleine mit den beiden vorne stehenden Sequenzen dürften sonst beide eintreten, da die Durchführungssequenzen durch ein (inklusives) Oder verknüpft sind.

Wir nehmen dieses allgemeine Constraint in dem Beispiel der Tabelle 6 auf (Zeile eins bis vier). Zusätzlich sei im System das Constraint *t(A) - s(B)* vorgegeben (Zeile fünf), also eine strenge Sequenz von A und B. In der Tabelle stehen die Ereignisse in der Titelzeile von links nach rechts in der Reihenfolge, in der sie in diesem Beispiel eintreten sollen. In den Spalten darunter sind jeweils nur die durch das Ereignis veränderten Zeilen dargestellt. Die ganz durchgezogenen Linien trennen die Constraints. Alle Zeilen, die nicht durch eine Linie getrennt sind, stellen Alternativen in einer Oder-Verknüpfung dar. Die Zeilen 1 bis 5 enthalten die ursprünglich definierten Constraints. Von diesen ändern sich die ersten vier nie, da sie eine allgemeine Regel darstellen, jedoch werden von ihnen Instanzen durch Anwendung von Ereignissen auf die Variable $a angelegt (ab Zeile sechs).

#		**i(B), i(A)**	**s(A)**	**t(A)**	**s(B)**	**c(B)**	**!t(B)**	**!c(A)**
1		i($a) - s($a) - c($a)						
2	v	i($a) - s($a) - t($a)						
3		!c($a)						
4	v	!t($a)						
5		t(A) -s(B)		s(B)	true			
6		s(B) - c(B)			c(B)	true		
7	v	s(B) - t(B)			t(B)			
8		!c(B)				false		
9	v	!t(B)					true	
10		s(A) - c(A)	c(A)					
11	v	s(A) - t(A)	t(A)	true				
12		!c(A)						true
13	v	!t(A)		false				

Tabelle 6: Beispiel für die Abarbeitung von Constraints

Zunächst kann nur ein *i()*-Ereignis eintreten, da alle anderen Ereignisse an einer späteren Stelle in einer Sequenz auftreten und somit blockiert sind (nicht aufgeführte Ereignisse wie z.B. *b(G,R)* oder *p(A)* könnten aber eintreten). Im Beispiel treten nun die beiden Ereignisse *i(A)* und *i(B)* ein. Danach kann nur *s(A)* akzeptiert werden, da wegen des Constraints in Zeile 5 *s(B)* erst nach *t(A)* erlaubt ist. *c(A)* darf jetzt nicht eintreten, weil dann sowohl *t(A)* (Zeile 5) als auch *!t(A)* (Zeile 13) eintreten müßten, was zu einem Widerspruch führt. Also wird mit *t(A)* fortgefahren, worauf nun die Aktivität B starten kann: *s(B)*. Letztlich beendet sich im Beispiel die Aktivität B durch den Abbruch *c(B)*.

Die Anwendung von Variablen in Ereignissen erlaubt uns zudem die Realisierung von Schleifen und Template-Instantiierungen. Im Planungsbeispiel aus Abschnitt 2.3.3.2 ist die Begutachtung einer vorher unbekannten Anzahl von möglichen Standorten notwendig. Ein Ereignis mit einer Variablen als Parameter kann dann für jede Bindung der Variablen an eine Konstante einmal eintreten. Ein weiteres Beispiel für den effektiven Einsatz von Variablen ist die Anwendung

des *X_NEEDS*-Constraints (siehe Tabelle 5), das *b(R,X)* bzw. *r(R,X)* verwendet, um die exklusive Nutzung durch irgendwelche Aktivitäten X zu modellieren. Dadurch kann bei einer Abarbeitung die dynamische Nutzungsänderung berücksichtigt werden [WAGNER 2001].

Somit haben wir durch den Aufbau von Constraints anhand parametrisierbarer und mit Variablen versehener Ereignisse eine flexible Durchführungsbeschreibung ermöglicht. Durch diese Constraints sind Designflow-Abhängigkeiten auf eine feingranulare Weise definierbar. Zudem können neue Constraints hinzugenommen werden, ohne den Abarbeitungsalgorithmus ändern zu müssen. Und nicht zuletzt kann ein auf Constraints basierendes Regelsystem dynamisch Constraints aufnehmen, wodurch die Änderung eines Designflows ermöglicht wird.

Ein Nachteil dieser Art Abarbeitung ist das Auftreten sich blockierender (Rest-)Sequenzen, wie wir es in Abschnitt 5.4 beschreiben.

4.4.5 Versorgung mit Entwurfsobjekten und Ressourcen

Im ADM ordnen wir die Entwurfsobjekte den Ressourcen zu. Wir unterscheiden diese nur sofern von Bedeutung (z.B. bei der Delegation), da die Entwurfsobjekte spezialisierte Ressourcen sind. Ressourcen sind eine weitere Voraussetzung für die korrekte bzw. vollständige Durchführung einer Aktivität. Ein Objekt wird wie folgt definiert:

```
OBJECT_TYPE: Obj_type(
    Obj_type | Primitive_type Attribute_Name [= value ] {, [Obj_type | Primitive_type] Attribute_Name [= value ] }*
)
OBJECT:  Obj_type Obj_Name(
    [Attribute_Name = value] {, Attribute_Name = value}*
)
```

Ein Objekt ist von einem spezifischen Objekttyp (*Obj_type*) und besitzt einen eindeutigen Namen (*Obj_Name*). Der Objekttyp wird durch Objekte und primitive Typen (*Primitive_type*, z.B. `integer`, `string`) definiert. Die Attribute können vorbelegt werden (*value*).

Wir haben im vorherigen Abschnitt bereits das *X_NEEDS*-Constraint eingeführt. Das Designflow-System sorgt dann für die exklusive Bereitstellung von Ressourcen. So ist beispielsweise für die Erstellung eines Werkstücks eine Fräse notwendig, die dann auch zur Verfügung stehen muß, was durch *X_NEEDS* bereits modelliert werden kann. Dagegen können sich aber auch Ressourcenbedürfnisse erst nach dem Start einer Aktivität ergeben; oder es ist vielleicht zunächst für den Start einer Aktivität nicht notwendig, daß eine bestimmte Ressource zur Verfügung steht, man aber davon ausgeht, daß sie zu einem späteren Zeitpunkt nutzbar wird. Daher ist eine exklusive Zuteilung zum Start nicht immer notwendig. Aktivitäten können frühzeitiger gestartet werden, da nicht alle Ressourcen exklusiv zur Verfügung stehen müssen. Andererseits muß aber dann auch der Zugriff verschiedener Aktivitäten auf die Ressourcen geregelt werden.

Die Standortliste aus dem Planungsbeispiel wird bspw. von einigen Aktivitäten nur lesend genutzt. Dies ist prinzipiell durch einen Constraint ausdrückbar, wenn entsprechende Ereignisse eingeführt werden. Jedoch ist aufgrund der großen Anzahl von Zugriffen und der verschiedenen Zugriffsarten auf Ressourcen (z.B. lesend, verschiedene Sperren einer Sperrenmatrix usw.) eine Realisierung aufwendig. Daher soll der Aufbau einer solchen Nutzungsbeziehung durch die initialisierte Aktivität selbst bestimmt werden und somit auch, ob der Start durchgeführt wird.

4.4.6 Akteure

Zum Starten einer Aktivität ist die Ermittlung des ausführenden Akteurs notwendig. Im ADM sind Akteure tatsächlich eine Spezialisierung von Ressourcen. Das leitet sich aus der Überlegung ab, daß Akteure aus dem Blickpunkt der Aktivität auch Ressourcen sind. Dies ist am Beispiel maschineller Akteure, wie etwa ein automatisches Computerprogramm oder eine Maschine, offensichtlich. Akteure zeichnen sich durch Rollen, Eigenschaften und Gruppenzu-

gehörigkeit aus. Natürlich ist es auch möglich, daß nur ein ganz bestimmter Akteur (z.B. “Erwin Engel”) diese Aktivität durchführen kann. Dies bedeutet, daß als erster Schritt die Menge möglicher Akteure berechnet und die davon verfügbaren ausgewählt werden. Ist wenigstens ein Akteur zu finden, so wird die Aktivität entweder in die entsprechende Worklist eingetragen oder im Falle einer automatischen Aktivität direkt ausgeführt. Wird die Aktivität nicht sofort gestartet, so kann durch andere Ereignisse eine Situation eintreten, die eine Ausführung nicht mehr erlaubt. Daher muß das Aktivitäten-Management ein Starten verhindern bzw. die Aktivität aus den betroffenen Worklisten entfernen.

4.4.7 Durchführung einer Aktivität

Wir haben nun die Überprüfung und Vorbereitungen für den Start einer Aktivität beschrieben. Danach kann im Falle einer automatischen Aktivität diese vom System gestartet werden (siehe auch Abschnitt 3.4). Bei interaktiven Aktivitäten geschieht die Durchführung durch den Akteur, der das von einer Worklist aus veranlaßt.

Worklist

Eine Worklist wird für einen Benutzer angelegt und beinhaltet die Aktivitäten, deren Startbedingungen erfüllt sind und die durch den Akteur ausgeführt werden dürfen. Eine Aktivität kann dabei in mehreren Worklisten eingetragen werden, jedoch nur einmal gestartet werden. Sie ist dann nicht mehr von den anderen Worklisten aus startbar.

Start einer Aktivität

Der Start führt nun die letztlich notwendigen Aktionen zum Ausführen der Aktivität durch. Dazu wird die entsprechende Aktivität benachrichtigt, die Objektnutzungsbeziehungen aufbaut und das zu verwendende Programm startet. Zur Beschreibung von Programmen existiert folgendes Konstrukt:

```
PROGRAM: Prog_Name (Start_Name, string)
```

Der Name ist der Identifikator für das zu startende Programm, wie zum Beispiel *Textverarbeitung* oder *spice simulator*. Der Platzhalter *Start_Name* benennt das Programm direkt, wie z.B: *word.exe*. Die Parameter für den Aufruf werden in Form einer Zeichenkette definiert (`string`). Sie erlauben eine Vorkonfiguration der Anwendung. Bspw. kann für eine Rechtschreibprüfung die Sprache vorbelegt werden (“`-spellcheck english`”). Die Parameterliste, kann auch Objekte referenzieren, die der Aktivität zur Verfügung stehen, z.B. “`$Standortliste.entry_7$`”.

Aktivitätszustände

Eine Aktivität kann zur Laufzeit verschiedene Zustände erreichen. Diese werden dem System durch die oben besprochenen Ereignisse bekanntgemacht. Das Zustandsübergangsdiagramm, das Aktivitäten zumindest erfüllen müssen ist in Abbildung 48 dargestellt und entspricht dem Diagramm aus Abschnitt 3.4.3.2. Lediglich der Zustand *bereit* ist besonders markiert, da er nicht von der Designflow-Engine zu berücksichtigen ist, weil eine Aktivität bereits “im System” sein muß, wenn sie gestartet werden soll. Es bleibt einer Implementierung überlassen weitere Zustände einzuführen. Sogar die Erzeugung entsprechender Ereignisse aufgrund einer erweiterten Ereignisabarbeitung ist durchführbar. Dafür ist jedoch die Event Engine um diese Ereignisse zu erweitern.

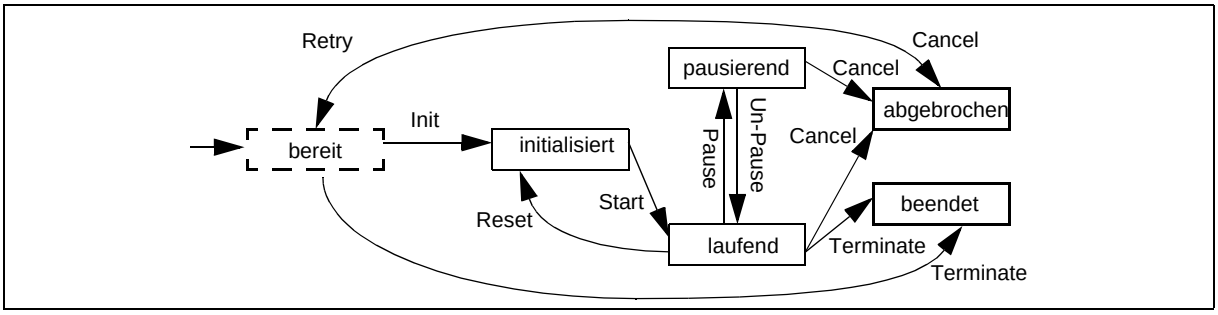


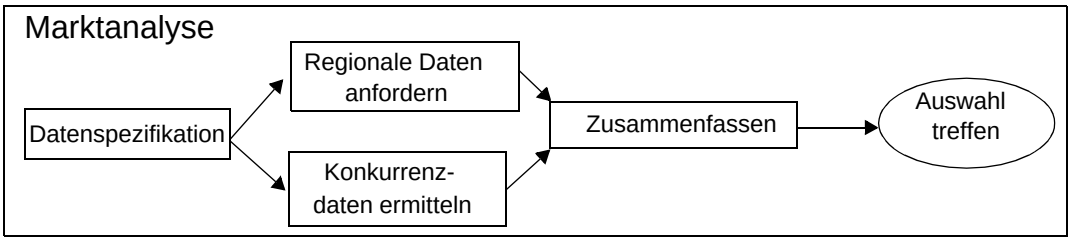
Abbildung 48: Zustandsübergangsdiagramm für ADM-Aktivitäten

4.5 Anwendung auf ein Planungs-Szenario

In diesem Abschnitt modellieren wir als Beispiel das Planungsszenario aus Abschnitt 2.3.3 um die Anwendung des Aktivitätenmodells zu erläutern.

Die Hauptaktivität *Standortplanung* wurde bereits in Abbildung 45 auf Seite 84 gezeigt. Wir beschreiben nun einige deren Subaktivitäten grafisch und textuell. Die vollständige Definition findet sich in Anhang C. Die *Marktanalyse* wurde als gut strukturiert erkannt. Daher beschreiben wir diese als eine Workflow-Aktivität (Abbildung 49). Lediglich die Aktivität *Auswahl treffen* wird als Groupware-Aktivität dem Designflow-System bekannt gemacht, da sie den Aufruf eines Konferenzwerkzeugs erfordert.

Der Workflow *Marktanalyse* soll mittels des WFMS FlowMark ausgeführt werden, das über die CORBA-basierte JFlow-Facility gekapselt wurde (siehe Abschnitt 5.5). Dies spezifiziert `APPLICATION: JFLOW_FLOWMARK( <ausführende Objektmethode>, <Übergabeparameter> )`. Die Subaktivitäten wurden nicht weiter beschrieben, da sie durch das Workflow-System zu verwalten sind. Für uns sind lediglich die gemeinsamen Daten von Bedeutung, weswegen hier das *uses*-Konstrukt (Nutzungsbeziehung) angewandt wurde um dem System mitzuteilen, wo die *Standortliste* durch Workflow-eigene Aktivitäten genutzt wird.

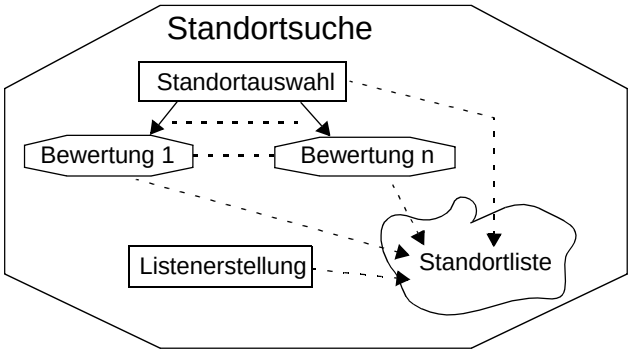



```
workflow-activity: MARKTANALYSE(
   DESIGN-OBJECTS:
      Standortliste
   ACTORS:
      Marktforschungsleiter
   GOAL:
      Standortliste_fertig=TRUE
   ACTIVITIES:
      DATENSPEZIFIKATION:   D,
      REGIONALDATEN:        R,
      KONKURRENZDATEN:      K,
      ZUSAMMENFASSEN:       Z,
      AUSWAHL:              A
   WF_PRIMITIVES:
      Z uses Standortliste,
      A uses Standortliste
   APPLICATION:
   JFLOW_FLOWMARK( MARKTANALYSE,
   Standortliste)
                                    )
```

Abbildung 49: Workflow-Aktivität Marktanalyse

Wir betrachten nun noch die Designflow-Aktivität *Standortsuche*, welche die Arealbewertungen durchführt (Abbildung 50). Hier wird in *Standortauswahl* die durch die *Marktanalyse* produzierte Datenbanktabelle *Standortliste*, die bisher nur Orte und Regionen enthält, an einzelne Gutachter verteilt, deren Aufgabe es ist, an jeweils einem Ort geeignete Bauplätze oder Immo-

bilien für neue Filialen zu finden. Diese führen die Suche und Bewertung vielversprechender Areale durch und tragen deren Eigenschaften in die dadurch erweiterte *Standortliste* ein. Die Liste steht allen Gutachtern zur Verfügung (aber auch der *Geschäftsführung* und der *Kostenabschätzung*), um deren Ergebisse etwa für Verhandlungen zu nutzen. Diese Liste wird durch eine automatische Aktivität *Listenerstellung* immer neu erstellt, d.h. es werden die einzelnen Standortdaten in einem lesbaren Dokument zusammengefaßt.




```
designflow-activity: STANDORTSUCHE(
   DESIGN-OBJECTS:
      Standortliste
   ACTORS:
      Leiter_Filialprojekt
   GOAL:
      "Begutachtung der Immobilien und
      Grundstücke."
   ACTIVITIES:
      STANDORTAUSWAHL        S,
      BEWERTUNG[i]:          B[i],
      LISTENERSTELLG:        L,
   DESIGN_PRIMITIVES:
      {S,L,B[i]} uses Standortliste,
      S delegates B[i],
      L starts_after S
)
```

Abbildung 50: Designflow-Aktivität Standortsuche

Wir gehen nun auf die Besonderheiten des *Bewertungs*-Designflows ein. Dies ist eine Aktivität, die mehrfach instantiiert werden kann. In der Definition ist das durch *B[i]* ausgedrückt und kann entsprechend durch das Aktivitätenmanagement erkannt werden[1]. In der unten aufgeführten Definition, ist zu sehen, daß diese Aktivität mit *Standorteintrag* als Entwurfsobjekt versorgt wird (ein Datensatz aus der Standortliste). Dies stellt den Bezug auf die *Standortliste* dar. Akteur dieser Aktivität ist eine Person, welche die Rolle *Gutachter* erfüllt. Zur Durchführung der Aktivität, d.h. für die Eingabe und Modifikation des Datensatzes, spezifiziert ***APPLICATION*** ein Dateneingabeprogramm *StandortEingabe* mit *Standorteintrag* als Übergabeparameter. Diese Applikation ist über das Internet verfügbar und wird daher mit einem HTML-Client (*netscape*) und der entsprechenden URL aufgerufen.

```
designflow-activity: BEWERTUNG(
   DESIGN-OBJECTS: Standorteintrag
   ACTORS: Gutachter
   GOAL: Standorteintrag.ERLEDIGT == true
   APPLICATION: StandortEingabe(Standorteintrag)
)
PROGRAM: StandortEingabe( netscape, "app_serv.uni.edu/app.html?standortmaske%20$Standorteintrag$")
```

Aufgrund der Struktur dieser Aktivität würde auch eine primitive Aktivität zur Beschreibung ausreichen. Es wurde jedoch eine Designflow-Aktivität gewählt, da es wahrscheinlich ist, daß hier zur Laufzeit weitere Aktivitäten delegiert werden können.

Das Constraint `L starts_after S` aus Abbildung 50 unterstützt die Durchführung einer ständig laufenden Aktivität Listenerstellung, deren Definition unten dargestellt ist. Diese ist durch ein dafür erstelltes Programm *Berichtgenerator* realisiert, das automatisch eine lesbare Form der Datentabelle *Standortliste* erzeugt, nämlich das Dokument *Standortbericht*. Der Generator ist ADM-aware, d.h. er registriert sich auf die Tabelle und läßt sich bei Änderungen notifizieren um automatisch einen aktuellen Bericht zu erstellen. Dieser Bericht wird entsprechend der Ressourcendefinition in dem angegebenen Pfad gespeichert.

1. Durch Anwendung der Variablen in Constraint-Definitionen können hier die multiplen Durchführungen und deren Beziehungen umgesetzt werden.

```
primitive-activity: LISTENERSTELLUNG(
    RESOURCES:Standortliste, Standortbericht
    ACTORS: automatic
    APPLICATION: Berichtgenerator( Standortliste, Standortbericht)
)
OBJECT: Standortbericht( Report = “reportserver.uni.edu/projects/new_branch/report.txt” )
PROGRAM:  Berichtgenarator( REPGEN.EXE, $Standortliste.table_name$ + “-path ” + Standortbericht.Report )
```

Wir haben nun die Anwendung der wesentlichen Beschreibungselemente von Designflows an einem Beispiel vorgestellt. Anhand der bereits dokumentierten Aktivitäten wurde gezeigt, wie ein Designflow erstellt wird und, daß diese Definition die Basis für die Nutzung des Informationsraums und der Objektnutzung darstellen. Auf die definierten Entwurfsobjekte und Ressourcen können Protokolle wie Delegation, Verhandlung und Nutzung bezüglich der betroffenen Aktivitäten und Objekte angewendet werden, wie sie im folgenden erläutert werden.

4.6 Informationsraum

Die gemeinsame Datennutzung ist ein wesentlicher Bestandteil unseres Kooperationsmodells, da sie zur Kooperation und damit auch Integration von Aktivitäten beiträgt. Weil die gemeinsame und regelbare Nutzung von Objekten und die Interaktion zwischen Aktivitäten ermöglicht wird, geht unser Ansatz über eine reine Ablaufunterstützung hinaus. Nachfolgend werden die dafür notwendigen Konzepte erläutert, die auf einem gemeinsamen Informationsraum und der Strukturierung von Objekten basieren.

Die Grundlage für die Kooperation stellt der Informationsraum dar, den wir in Abschnitt 4.6.1 motivieren. Durch diesen sind die gemeinsamen Objekte den Aktivitäten verfügbar (Abbildung 51). Wir beschreiben die Struktur des Informationsraums in Abschnitt 4.6.2. Daraufhin führen wir in Abschnitt 4.6.3 die Strukturierung von Objekten durch Objektabhängigkeiten ein, die eine anwendungsspezifische Modellierung von komplexen Objekten erlauben und zudem direkt die Idee des Zerlegen eines Entwurfsartefakts unterstützen (Abschnitt 4.6.4). Zum Schluß fassen wir dies in Abschnitt 4.6.4 übersichtlich zusammen, um zu den Interaktionsprotokollen in Abschnitt 4.7 überzuleiten, die auch für Objektzugriffe angewendet werden.

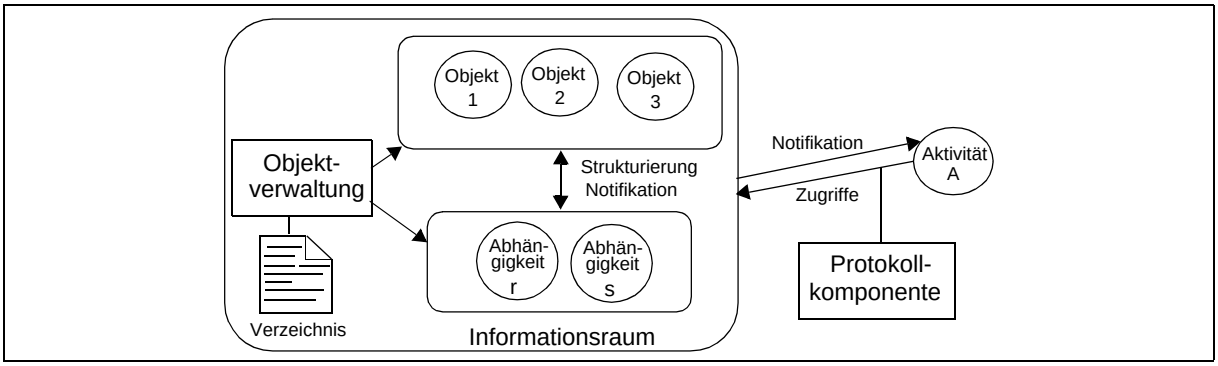


Abbildung 51: Komponenten des Informationsraums

4.6.1 Allgemeine Kooperationsmöglichkeiten

Die Abbildung 52 stellt den Informationsraum als integrierendes Element für die verschiedenen Technologien im ADM dar. Der Informationsraum dient zur gemeinsamen Nutzung von Objekten und fördert dadurch die Kooperation zwischen den Akteuren in einem ADM-System. Die Nutzer von Objekten sind die Aktivitäten, d.h. die Objekte sind nicht an Akteure gebunden. Dadurch wird eine enge Kopplung der verschiedenen Aktivitätsarten erreicht.

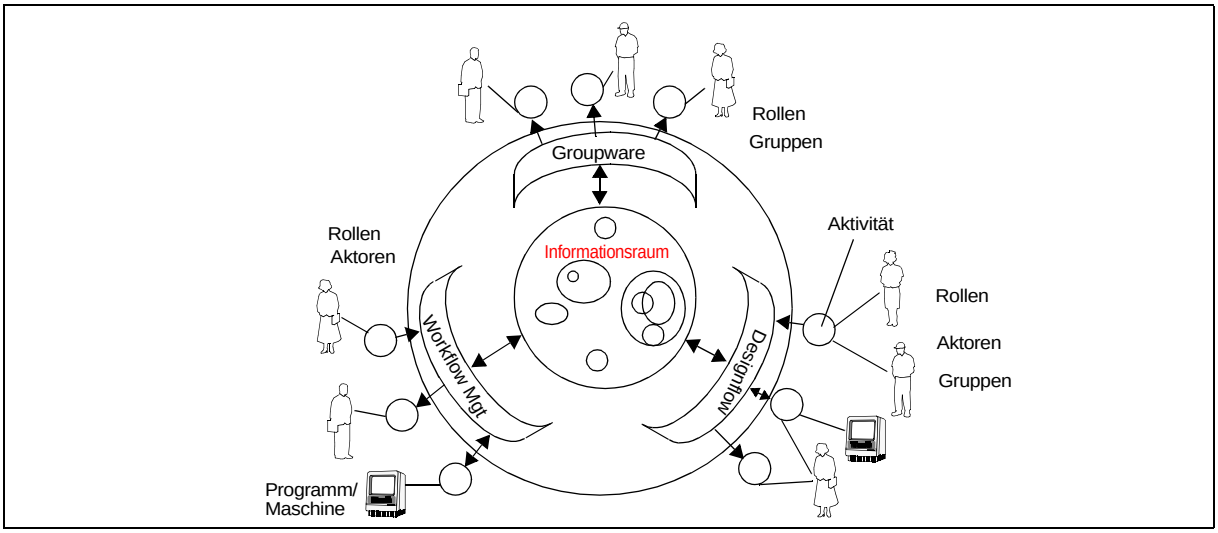


Abbildung 52: Kooperation über einen gemeinsamen Informationsraum

Durch diesen Ansatz und die zusätzliche Einbindung von Interaktionsprotokollen wird eine weitreichende Unterstützung der in Abschnitt 3.3 vorgestellten CSCW-Mechanismen erreicht. Die Interaktionsprotokolle werden nicht nur zur Benutzung des Informationsraumes, sondern auch zur direkten Kooperation und Kommunikation zwischen allen Entitäten des ADM eingesetzt, wie schon bei der kurzen Beschreibung von Delegation und Verhandlung angedeutet (Abschnitt 4.4.1.3). Die wichtigsten Mechanismen unseres Kooperationsmodells sind:

- *Der Informationsraum*
  Er ist die Basis für die gemeinsame Objektnutzung.
- *Bereitstellung von Objekten im Informationsraum*
  Im Informationsraum werden die Objekte bereitgestellt, die von Aktivitäten gemeinsam genutzt werden können.
- *Nutzung von Objekten durch Aktivitäten*
  Aktivitäten können sich auf Objekte registrieren, um diese zu nutzen.
- *Strukturierte Objekte*
  Durch die Einführung von Abhängigkeiten können komplexe Objekte modelliert werden, welche die Zerlegung von Entwurfsartefakten unterstützen.
- *Geregelte Objektzugriffe*
  Registrierte Aktivitäten können auf Objekte mittels Interaktionsprotokollen zugreifen. Diese Protokolle berücksichtigen objekttypische Zugriffsmöglichkeiten.
- *Kooperation über Objekte*
  Die gemeinsame Nutzung von Objekten erfordert eine Regulierung der Zugriffe. Diese kann auf verschiedene Arten erreicht werden:

  a) Durch die Vergabe von Zugriffsrechten und deren Durchsetzung wird eine gewisse Group Awareness und transaktionsähnliche Koordination erreicht. Beispielsweise besitzt Aktivität B das exklusive Schreibrecht für ein Objekt. Deswegen erhält A bei einem Schreibversuch lediglich die Meldung, daß B die Rechte zum Schreiben besitzt.
  b) Über Notifikationsmechanismen werden registrierte Aktivitäten über Objektänderungen informiert. Hier ist zwischen Pull und Push zu unterscheiden, wobei der Pull-Mechanismus eine weniger starke Awareness-Realisierung darstellt (asynchrone Benachrichtigung).
- *Explizite Kooperation*
  Direkte Kommunikation wird durch die Verwendung von Interaktionsprotokollen realisiert, die auch durch den Akteur geleitet werden können (z.B. eine Anfrage an Aktivität B,

bzw. deren Akteur, zur Abgabe ihres Schreibrechts auf ein gemeinsames Objekt).

Durch die oben aufgeführten Mechanismen qualifiziert sich das ADM als CSCW-Modell. Es nutzt direkte Koordinationsmechanismen, die typischerweise dem Bereich Workflow zuzuordnen sind, und indirekte Koordinationsmechanismen, die eher dem Group Authoring entsprechen. Darüber hinaus werden synchrone Kommunikation und Kooperation angeboten.

4.6.2 Strukturierung des Informationsraums

Der Informationsraum verwaltet die Objekte, die für Kooperationen bereitstehen. Die Strukturierung des Informationsraums kann grundsätzlich auf zwei Arten erfolgen. Einerseits können den Aktivitäten die Objekte, und andererseits können den Objekten die Aktivitäten zugeordnet werden. Im folgenden werden wir die Vor- und Nachteile dieser beiden Ansätze diskutieren und die Entscheidung für die Zuordnung von Objekten zu Aktivitäten darlegen.

Ein hierarchischer Informationsraum, der auch in der Implementierung der CSCW-Facility eingesetzt wurde (Abschnitt 5.6), betrachtet die Zuordnung von Objekten zu Aktivitäten. Dies hat den Vorteil, daß ein Informationsraum die Zugriffsregelung auf die enthaltenen Objekte umfassend regeln kann. Das von uns gewählte Konzept des Informationsraums basiert auf sogenannten Kooperationsmengen, die sich aus der Zuordnung von Aktivitäten zu Objekten ergeben. Daher erfolgt hier die Zugriffsregelung durch die Objekte selbst. Wir diskutieren im folgenden die Gründe für diese Entscheidung.

4.6.2.1 Zuordnung von Objekten zu Aktivitäten

Die Zuordnung von Objekten zu Aktivitäten erfolgt über Informationsräume, die jeweils zu den entsprechenden Aktivitäten gehören. Informationsräume enthalten die Objekte, die von den Aktivitäten benötigt werden. Diese Informationsräume können auf verschiedene Arten miteinander verbunden sein (siehe Abbildung 53). Dies reicht von einer hierarchischen Schachtelung (die Kanten bedeuten hier eine Vater-Kind-Beziehung) bis hin zu einem gemeinsamen, globalen Informationsraum, der allen Aktivitäten zugeordnet ist.

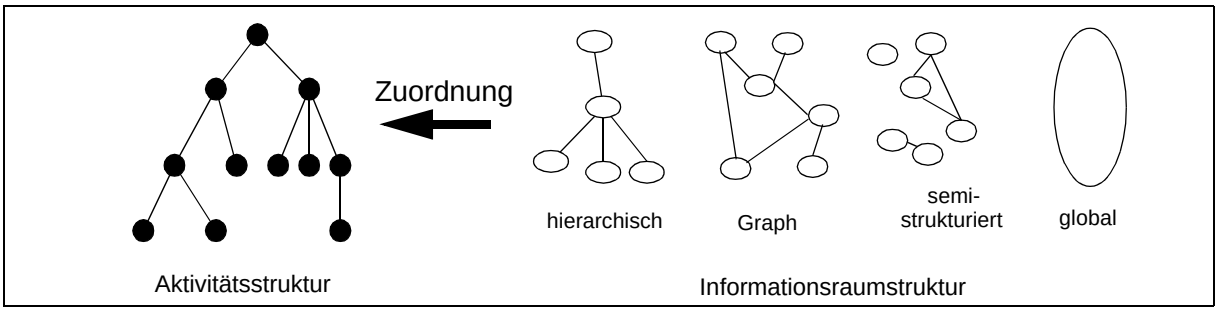


Abbildung 53: Verschiedene Strukturen für den Informationsraum

Ein globaler Informationsraum erlaubt es nicht, die einzelnen Objekte den Aktivitäten strukturiert zuzuordnen, da sie alle im selben Informationsraum verwaltet werden und sich somit nicht an der Aktivitätsstruktur orientieren. Hier ergibt sich der Fall, den wir im folgenden Abschnitt besprechen. Wir untersuchen nun den hierarchischen Informationsraum, der die Schwächen von strukturierten Informationsräumen offenbart, in welchen die Objekte den Aktivitäten zugeordnet werden.

Ein hierarchischer Aufbau des Informationsraums wird z.B. in der von uns implementierten CSCW-Facility in Form hierarchischer Workspaces realisiert (Abschnitt 5.6). Ein solcher Workspace enthält für jeweils eine Gruppen-Aktivität einen Sub-Workspace, der die benötigten Objekte verwaltet. Diese Zuordnung entspricht dem hierarchischen 1:1-Informationsraum in

Abbildung 54, als Spezialisierung eines allgemeinen hierarchischen Informationsraums (Abbildung 53). Es entsteht ein hierarchischer Raum, welcher der Schachtelung der Aktivitäten entspricht. Das Workspace-Konzept hat den Vorteil, daß jeder Aktivität auch ihre Ressourcen zugeordnet sind, die sie nutzen kann. Ein solcher Workspace verwaltet die externen Zugriffe auf enthaltene Objekte [MUCHITSCH 1998]. Bis auf die Workspace-Wurzel kann ein Zugriff von außen auf einen Workspace nur über dessen Super-Workspace erfolgen. Durch die Hierarchisierung wird sowohl die Abhängigkeit der Aktivitäten unterstützt, als auch ein hierarchischer Zugriff auf Ressourcen ermöglicht. Dieser Ansatz ist interessant, da die Entwurfsobjekte einer Verfeinerung durch die Delegation unterliegen. Somit kann durch einen hierarchischen Informationsraum die Zerlegung von Entwurfsartefakten strukturell unterstützt werden.

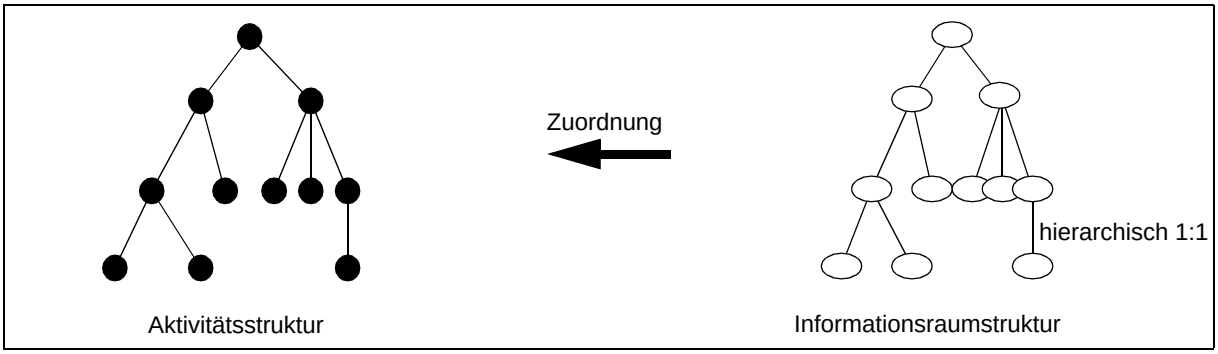


Abbildung 54: 1:1-Zuordnung von Informationsräumen zu Aktivitäten

Der Vorteil von geschachtelten Informationsräumen dieser Art ist die Sichtbarkeit bzw. die einfache Nutzbarkeit von Objekten. So kann eine Subaktivität auf Objekte im Informationsraum ihrer Superaktivität zugreifen. Dies unterstützt auch die Delegation von Subaufträgen, in denen Teilobjekte bearbeitet werden müssen. Jedoch birgt dieses Konzept große Nachteile für unsere Anwendungsumgebung. Einerseits wird hier lediglich die Hierarchie von Entwurfsartefakten betrachtet und nicht die Verwaltung aller zu bearbeitenden Objekte, die in einem Informationsraum liegen können. Andererseits erschwert diese Struktur die gemeinsame Nutzung von Objekten in verschiedenen Ästen der Aktivitätsstruktur, also die von uns gewünschte gemeinsame Nutzung und Kooperation auf Objekten. Dies wird offensichtlich, wenn man die Nutzung eines Objekts aus unterschiedlichen Teilbäumen betrachtet, wie es in Abbildung 55 angedeutet ist. Hier ist jeder Aktivität ein entsprechender Informationsraum zugeordnet, und die Struktur des Informationsraums entspricht der Aktivitätsstruktur.

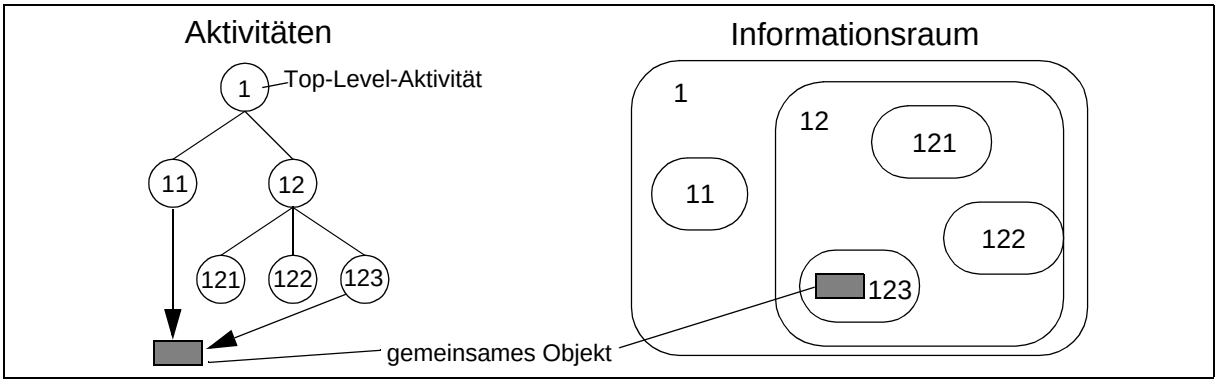


Abbildung 55: Nutzung eines gemeinsamen Objekts bei hierarchischen Informationsräumen

Gemeinsame Objekte sollen in verschiedenen Teilbäumen der Aktivitätsstruktur bearbeitet werden können. Im dargestellten Szenario bezeichnen wir die Aktivität *123* als den primären Nutzer oder auch *owner* des gemeinsamen Objekts, da sich das Objekt in deren Informationsraum befindet. Die Aktivität *11* nennen wir sekundär oder *user*, da diese das Objekt zwar nutzt,

es sich aber in einem anderem Informationsraum befindet. Für die Nutzung von Objekten durch Aktivitäten aus verschiedenen Ästen der Aktivitätsstruktur können die folgenden Mechanismen angewandt werden:

- *Das gemeinsame Objekt wird in den kleinsten gemeinsamen Informationsraum migriert (im Beispiel ist dies Informationsraum* 1*)*
  Dieses Vorgehen wird von der CSCW-Facility umgesetzt, hat aber den Nachteil, daß bei einem dynamischen Aktivitätsmodell mit dynamischen Objektnutzungen häufig entsprechende Migrationen durchgeführt werden müßten. Entsprechend werden viele Objekte in die obersten Informationsräume migriert und daher der hierarchische Informationsraum degeneriert. Im extremsten Fall entwickelt er sich zum globalen Informationsraum. Dies wirkt der Verfeinerung von Entwurfsobjekten entgegen und steht somit im Widerspruch zu einer unserer wichtigsten Intentionen für den Informationsraum, nämlich, daß die Objektzerlegung durch die Informationsraumhierarchie unterstützt wird.
- *Belassung des gemeinsamen Objekts bei der primär nutzenden Aktivität bzw. in deren Informationsraum mit hierarchischem Zugriff für sekundäre Aktivitäten.*
  In diesem Fall gibt es eine owner-Aktivität des Objekts, das entsprechend der Zerlegung und Delegation in dem korrespondierenden Informationsraum abgelegt wurde. Der Zugriff von sekundären Aktivitäten erfolgt über die Struktur des Informationsraums. In unserem Beispiel, fragt Aktivität *11* in Informationsraum *1* nach dem Objekt und wird über *12* nach *123* verwiesen. Dieses Vorgehen erhält zwar die Struktur des Informationsraums, entspricht aber nicht der Idee der einfachen gemeinsamen Nutzung. Nach der Beendigung einer Aktivität ist das Objekt in einen anderen Informationsraum einzubringen. Es bietet sich an den übergeordneten Informationsraum *1* zu wählen, was auch dem Prinzip der Integration entspricht. Dieser Informationsraum kann hingegen andere Zugriffsbeschränkungen besitzen, als der vorherige, was die Nutzung durch die bisher registrierten Aktivitäten erschwert.
- *Belassung des gemeinsamen Objekts bei der primär nutzenden Aktivität bzw. in deren Informationsraum mit direktem Zugriff für sekundäre Aktivitäten.*
  Im Gegensatz zur obigen Alternative kann eine sekundär nutzende Aktivität auf das gemeinsame Objekt direkt zugreifen. Hier ist nun fraglich, warum überhaupt noch die hierarchische Strukturierung beibehalten werden sollte, wenn man insbesondere die Nutzungs- bzw. Zugriffsrechte auf Objekte dynamisch ändern können will.
- *Einführung von Stellvertreterobjekten*
  Um der konzeptionellen Anforderung nachzukommen, daß genutzte Objekte in einem zugreifbaren Informationsraum vorliegen müssen, können Stellvertreterobjekte (*proxies*) in den entsprechenden Informationsräumen eingeführt werden, die den Zugriff auf das tatsächliche Objekt weiterleiten. Letztendlich wird dadurch die Problematik lediglich verschleiert, da weiterhin das gleiche Problem besteht, das nur mit einer neuen Schicht (d.h. den Stellvertreterobjekten) verdeckt wird.

Trotz der aufgeführten Nachteile ist zu beachten, daß ein Vorteil des beschriebenen Informationsraumsmodell die einfache Verwaltung der Objektzugriffe ist, falls nur Aktivitäten innerhalb eines gemeinsamen Informationsraums auf ein Objekt zugreifen dürfen und dort die Rechte verwaltet werden (daher auch oben die zweite Alternative mit dem hierarchischen Zugriff). Sehr nachteilig ist das rigide Definitionsmodell, das sich durch die Zerlegung von Objekten und Aktivitäten ergibt. So sollten Objekte entsprechend der Aufgaben, d.h. letztendlich Aktivitäten, zerlegt und zugeteilt werden. Dies läßt zur Laufzeit jedoch nur schwer Änderungen zu, da möglicherweise die Objekte auf andere Weise zerlegt werden müssen, wenn die Aktivitätenstruktur verändert wird.

Die anderen Strukturierungen (Graph oder semi-strukturiert) aus Abbildung 53 besitzen die entsprechenden Nachteile, die aus der Zuordnung zu den Aktivitäten resultieren. Darüberhinaus berücksichtigen sie nicht die Abhängigkeiten zwischen Aktivitäten, ansonsten wären sie im wesentlichen hierarchisch und eignen sich daher ebensowenig für unsere Anwendungsumgebung. Daher betrachten wir im folgenden die zweite Möglichkeit, die auf der Zuordnung von Aktivitäten zu Objekten beruht. Dadurch soll die Objektstruktur erhalten bleiben und die gemeinsame Nutzung von Objekten erleichtert werden.

4.6.2.2 Zuordnung von Aktivitäten zu Objekten durch Kooperationsmengen

Aufgrund der zuvor erläuterten Nachteile sind wir nicht mit dem Konzept des hierarchischen Informationsraums, der sich an der Aktivitätsstruktur orientiert, zufrieden und bedienen uns einer anderen Sichtweise. Wir gehen von den Objekten aus und binden die Aktivitäten an diese. Dadurch geht zunächst die Strukturierung der Objekte über die Hierarchie des Informationsraums verloren, weswegen wir in Abschnitt 4.6.3 die Modellierung der Objektstruktur einführen. Außerdem ist die Zugriffsverwaltung in die Objekte zu verlagern. Die Aktivitäten registrieren sich bei diesem Ansatz direkt bei den Objekten, wodurch implizit eine *Kooperationsmenge* für jeweils ein Objekt entsteht, die alle beteiligten Aktivitäten beinhaltet (siehe Abbildung 56). Die Zugriffe auf das Objekt werden in der Kooperationsmenge geregelt.

In Abbildung 56 sind komplexe Objekte dargestellt, die nicht durch die Struktur eines entsprechenden Informationsraums aufgebaut sind. Dies kann man durch die Verwendung von Abhängigkeiten erreichen, wie im folgenden Abschnitt erklärt. Dadurch können wir die Zerlegung von Entwurfsartefakten realisieren und zwar nicht durch die Struktur des Informationsraums, sondern durch die komplexen Objektstrukturen selbst.

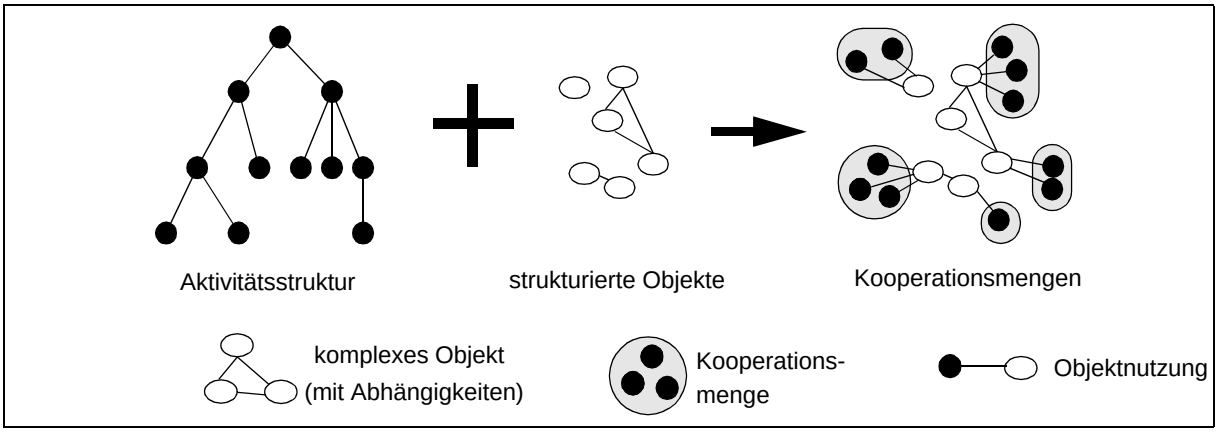


Abbildung 56: Kooperationsmengen

4.6.3 Objektabhängigkeiten

Wir führen Abhängigkeiten ein, durch die komplexe Objekte modelliert werden können. Beispielsweise kann eine Ausprägung einer Abhängigkeit die part-of-Beziehung sein, wie es anhand eines Fahrwerks, das aus Vorderachse, Hinterachse und Bremssystem besteht, in Abbildung 57 dargestellt ist:

Die grundlegenden Eigenschaften einer Abhängigkeit sind:

- Ein Superobjekt, das von weiteren Objekten abhängig ist:
  Wir beschränken uns auf ein einzelnes Superobjekt, dem die Subobjekte zugeordnet sind.
- Eine Menge von Subobjekten.
- Methoden zur Strukturierung und Abfrage der beteiligten Objekte:

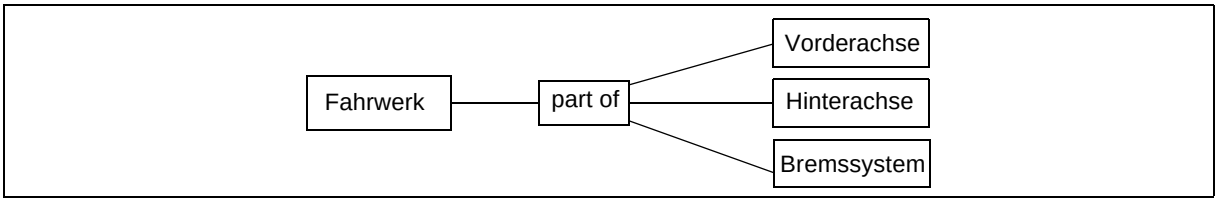


Abbildung 57: Die Part-Of-Beziehung als konkrete Objektabhängigkeit

Objekte können über Abhängigkeiten strukturiert werden, um ein komplexes Objekt aufzubauen. Die an einer Abhängigkeit beteiligten Objekte können erfragt werden.

- Methoden zur Notifikation beteiligter Objekte über neue Objektzustände:
  Bei einer Zustandsänderung eines beteiligten Objekts kann die Abhängigkeit über diese Änderung benachrichtigt werden. Je nach Art der konkreten Abhängigkeit kann eine Notifikation der anderen beteiligten Objekte erfolgen, evtl. sogar ein Abstimmungsverfahren (wie z.B. mit TOGA realisiert [FRANK ET AL. 2000]).
- Änderungen zur Laufzeit:
  Dadurch kann beispielsweise ein komplexes Objekt zur Laufzeit verfeinert werden.
- Anpaßbarkeit:
  Als konkrete Ausprägungen sind *part-of* und *is-a* für uns wichtig, da part-of die Bildung zusammengesetzter Objekte erlaubt und is-a eine Möglichkeit darstellt 'typsichere' Objektstrukturen zu erstellen (siehe Abschnitt 4.6.3.2). Diese können um weitere ergänzt werden, wenn dies sich für ein Anwendungsszenario anbietet (z.B.: *manufacturer-of, located-at*).

4.6.3.1 Spezialisierung von Abhängigkeiten

Eine abstrakte *Dependency* stellt die beschriebenen Eigenschaften zur Verfügung und ist eine generische Beziehung zwischen Objekten. Es werden nur die oben aufgeführten abstrakten Mechanismen angeboten. Für eine Realisierung ist eine Spezialisierung dieser Klasse notwendig. Wie man in Abbildung 58 sieht, kann *Dependency* bspw. zu einer part-of-Beziehung oder zu einer is-a-Beziehung spezialisiert werden. Dadurch werden typisierte Abhängigkeiten erreicht. Diese Beziehungen können nun weiter spezialisiert werden, um anwendungsspezifische Eigenschaften zu realisieren, die weitergehende Abhängigkeiten berücksichtigen. So kann die in der Abbildung 57 dargestellte part-of-Beziehung die spezifischen Entwurfsvorgaben für einen technischen Entwurf berücksichtigen, wie zum Beispiel die Erzwingung eines Abstimmungsverfahrens für Änderungen, oder technische Bedingungen für Unterobjekte verwalten (z.B. Durchmesser des Bauteils darf maximal 10 cm betragen). Diese Abhängigkeit entspricht damit der *technical-part-of*-Klasse aus Abbildung 58. Diese Spezialisierung kann weitergeführt werden bis hinzu speziellen komplexen Objekten wie einer ALU, die durch CPU-ALU-part-of erstellt werden kann. Ein generisches Beispiel ist die *rule-based-part-of-Beziehung*, in der bestimmte Regeln zur Erfüllung der Beziehung bereitgestellt werden können. Dadurch sind dann innerhalb einer spezifischen Anwendung verschieden konfigurierte rule-based-part-of-Beziehungen möglich. Die konkrete Implementierung einer Abhängigkeit wird dadurch offen gelassen. Diese kann beispielsweise in starren Code gegossen sein oder auch durch eine Regelinterpreter realisiert sein.

Durch diese Flexibilität aufgrund der Spezialisierung und Anpaßbarkeit von Abhängigkeiten erhalten wir weitgehende Möglichkeiten Entwurfsartefakte verschiedener Anwendungsfälle zu modellieren. In einer konkreten ADM-Implementierung können nun verschiedene Dependency-Klassen eingesetzt werden.

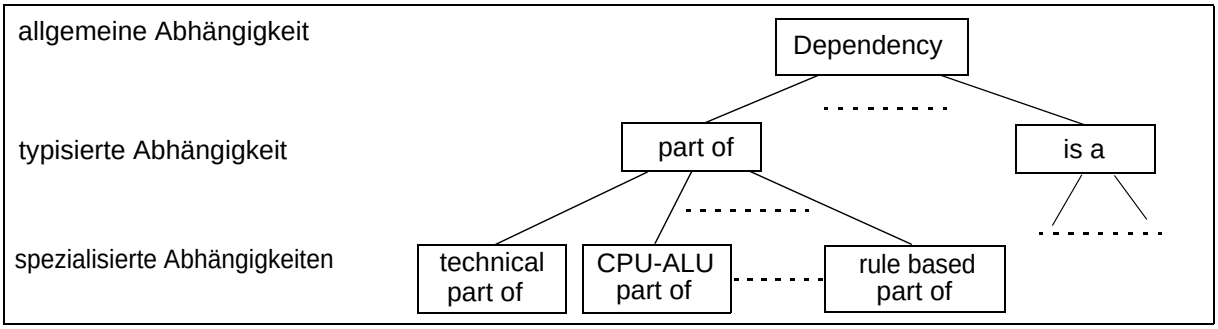


Abbildung 58: Spezialisierung der Dependency-Klasse

4.6.3.2 Beispiel für ein komplexes Objekt

Wir zeigen hier ein vereinfachtes Beispiel für ein komplexes Entwurfsartefakt, um zu zeigen wie Abhängigkeiten für den Entwurf angewendet werden. In der Abbildung 59 wird ein Ausschnitt der Struktur eines Automobilfahrwerks dargestellt. Diese Struktur wird durch die generische part-of-Beziehung aufgebaut. Zusätzlich wird für die Befestigungsschrauben der Einsatz der is-a-Beziehung gezeigt. Die is-a-Beziehung kann hier ausgenutzt werden, falls etwa eine technische Änderung am Bremssystem der Vorderachse eine andere Befestigungsschraube erfordert, so muß entweder der Schraubentyp für vorne geändert werden, d.h. es wird eine weitere is-a-Beziehung mitaufgenommen, oder es wird (z.B. aus Gründen der Kostenersparnis) die Hinterachse ebenfalls mit diesem Schraubentyp ausgestattet. Mit Hilfe des Notifikationsmechanismus aufgrund bestehender Nutzungsbeziehungen können diese Entwurfsentscheidungen dann an die betroffenen Aktivitäten propagiert werden.

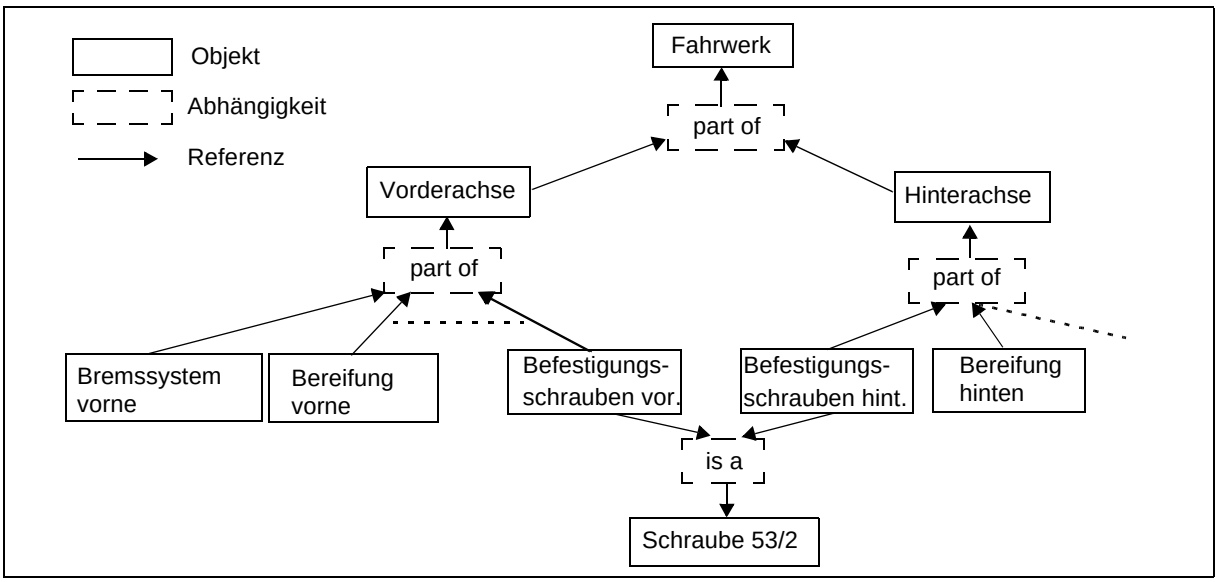


Abbildung 59: Darstellung des komplexen Objekts Fahrwerk durch Abhängigkeiten

4.6.4 Anwendung von Objektstruktur, Notifikation und Kooperationsmengen

Durch Abhängigkeiten werden komplexe Objekte innerhalb des Informationsraums realisiert. Anstatt der Untergliederung des Informationsraums werden Kooperationsmengen in den Objekten aufgebaut, die alle registrierten Aktivitäten eines Objekts enthalten. Zudem bleibt die Strukturierung eines komplexen Objekts durch Abhängigkeiten erhalten.

Diese Abhängigkeiten können, je nach konkreter Implementierung, die Kooperationsmengen implizit erweitern. Dies geschieht beispielsweise, wenn ein Objekt verändert werden soll, dieses jedoch vorher sein Superobjekt “um Erlaubnis fragen” muß. Damit ist natürlich die Kooperati-

onsmenge des Superobjekts indirekt an der Kooperationsmenge des Subobjekts beteiligt, denn es hängt von dieser Super-Kooperationsmenge ab, ob die Änderung letztlich durchgeführt werden darf. Dies kann zu einer Kaskadierung von Nachfragen über mehrere Abhängigkeiten hinweg führen und ist entsprechend ein *worst case scenario*.

Wir wollen dies am Beispiel des Planungs-Szenarios zeigen. Dort wird die Standortliste genutzt, um die Daten über die verschiedenen in Frage kommenden Standorte zu verwalten. In den Arbeitsschritten zur Bewertung der Standorte, werden diese evaluiert und über deren Rahmenbedingungen (Preis, Vertragsinhalte usw.) verhandelt. Die Ergebnisse werden in einer Liste zusammengestellt. Diese Standortliste wird auch von der Geschäftsführung genutzt, um vielversprechende Standorte zu erkennen und an eine Kostenabschätzung zu delegieren.

In Abbildung 60 ist ein solches Szenario dargestellt, in dem die Bewertungsaktivität *Bewertung 5* das Löschen ihres Eintrags (*Standort 5*-Objekt) auf der Liste verlangt (1). Diese Operation hat wesentliche Auswirkungen auf das Superobjekt *Standortliste* und daher wird die Anfrage zum Superobjekt eskaliert (2). Da die Liste aktiv von den Aktivitäten *Geschäftsführung*, *Kostenabschätzung* und *Listenerstellung* genutzt wird, werden diese um eine gemeinsame Entscheidung zur Durchführung der Löschung gebeten (3). Diese gemeinsame Entscheidung wird durch den Kommunikationspfeil (3) und die Kooperationsmenge angedeutet. Die Entscheidungsfindung kann nun aufgrund existierender Protokolle (z.B. TOGA) oder interaktiver Verhandlung getroffen werden (nicht dargestellt). Das Ergebnis wird an das Objekt *Standort 5* propagiert (4). Das Objekt kann nach erfolgter Zustimmung registrierte Aktivitäten (hier: *Bewertung 1*) über die Löschung benachrichtigen (5).

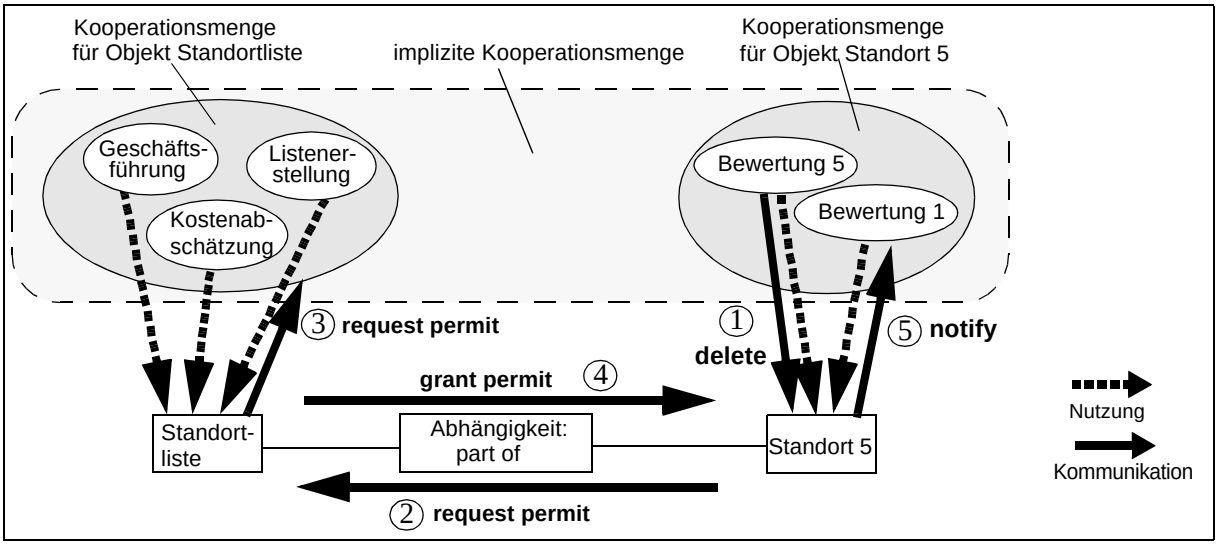


Abbildung 60: Implizite Erweiterung von Kooperationsmengen durch Objektabhängigkeiten

Je nach gewählter Implementierung der Objekte bzw. Abhängigkeit kann nach Schritt (1) oder (2) das Protokoll beendet sein; wenn etwa die Abhängigkeit nur passiv die Änderung annimmt, aber nicht um eine Änderungserlaubnis nachfragen muß. Respektive braucht Schritt (3) nicht zu erfolgen, wenn das Objekt *Standortliste* keine Erlaubnis durch dessen Kooperationsmenge benötigt. Diese Entscheidungen sind anwendungsspezifisch bzw. objektspezifisch und müssen entsprechend in den Objekten und Abhängigkeiten implementiert bzw. konfiguriert sein.

Wie in dem Beispielszenario angedeutet, ist die Notifikation ein wichtiges Mittel zur Konsistenzerhaltung in komplexen Objekten. Zudem kann ein Notifikationsmechanismus die Awareness von am Designflow beteiligten Personen erhöhen und damit die Zusammenarbeit vereinfachen.

4.7 Interaktionsmodell

Im vorherigen Abschnitt wurde die Struktur des Informationsraums festgelegt. Erst im Zusammenhang mit den entsprechenden Interaktionsprotokollen können die kooperativen Aspekte verwirklicht werden. Hier werden diese Interaktionen von Aktivitäten und Objekten beschrieben. Zur Benutzung von Objekten definieren wir flexible Interaktions- und Verhandlungsprotokolle auf Basis von Agententechnologie (siehe auch Abschnitt 3.5). Dadurch ist es auch auf dieser Ebene möglich flexible und der Anwendung angepaßte Kommunikations- und Kooperationsverfahren zur Verfügung zu stellen [FRANK 2001]. Diese Verfahren werden durch die Definition entsprechender Nachrichten und darauf aufbauender Protokolle erreicht. Diese Protokolle werden auch in anderen Bereichen des ADM angewendet, wie der Durchführung der Delegation oder direkter Interaktion und Verhandlung zwischen Aktivitäten und Akteuren.

Durch diese Kombination können wir Koordination und Kooperation auf verschiedenen Ebenen behandeln (wie in Abbildung 61 gezeigt). In diesem Beispiel werden die Spezifikation einer ALU und der MMU eines Mikroprozessors durch die Aktivität *Design Chip* an die Aktivitäten *Design ALU* bzw. *Design MMU* delegiert. Da diese Bestandteile des Prozessors nicht vollkommen unabhängig sind, also auch von weiteren Teilen abhängen, wie der dargestellte *Level 2 Cache*, werden entsprechende Nutzungsbeziehungen eingegangen, um über Änderungen in deren Entwurf benachrichtigt zu werden. Konflikte können bereits weitgehend durch eine geeignete Zerlegung der Entwurfsaufträge im Rahmen der Delegation vermieden werden. Im Falle von ungenügender oder sich ändernder Spezifikation von Aufträgen können diese nachverhandelt werden (siehe Abschnitt 4.7.2.1). Die Nutzung von gemeinsamen Objekten des Informationsraums wird durch geeignete Nutzungsprotokolle realisiert (siehe Abschnitt 4.7.2.2), wodurch indirekt kooperiert werden kann und Konflikte vermieden werden können. Schließlich ist das letzte Mittel die direkte Kooperation zwischen Entitäten (Abschnitt 4.7.1), die eine Verhandlung zwischen den Aktivitäten ermöglicht.

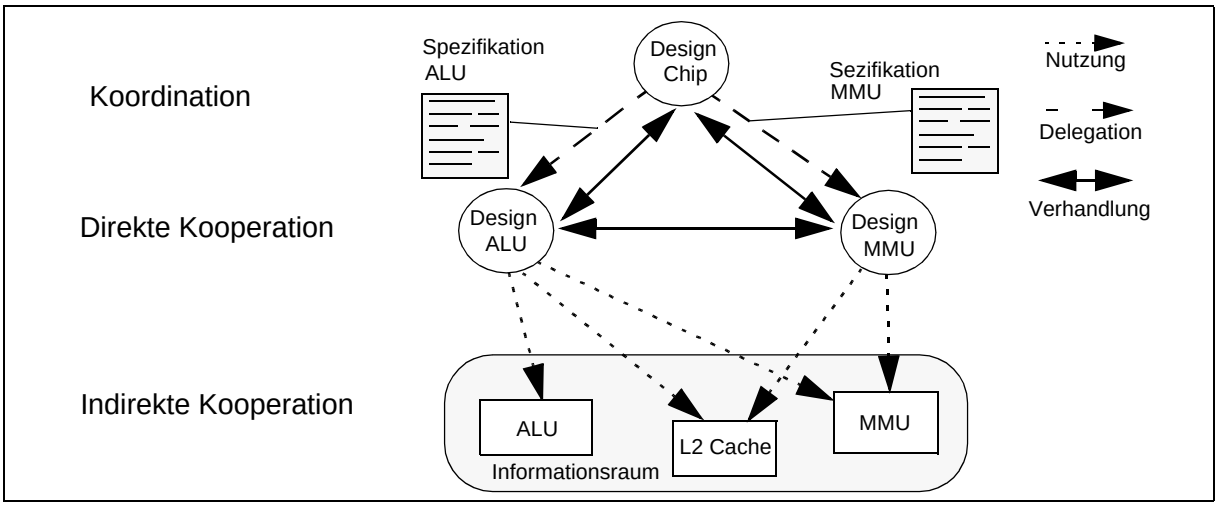


Abbildung 61: Verschiedene Ebenen der Konfliktbehandlung durch Interaktionsprotokolle

Zur Definition unseres Kommunikationsmodells im ADM ist es notwendig zuerst die zugrundeliegende Kommunikation zu beschreiben. Das Modell besteht aus den auszutauschenden Nachrichten und den darauf aufbauenden Protokollen (siehe Abbildung 62). Die Protokolle definieren Interaktionen zwischen den Entitäten, also Aktivitäten, Objekten und Abhängigkeiten. (1) stellt eine Interaktivitätenkommunikation dar, wie sie beispielsweise bei der Delegation auftritt. Die Objektnutzung (2) basiert auf dem zuvor beschriebenen Zugriff auf Objekte im Informationsraum. Wie in Abbildung 60 bereits angedeutet, nehmen auch die Abhängigkeiten zwischen Objekten an der Kommunikation teil (3).

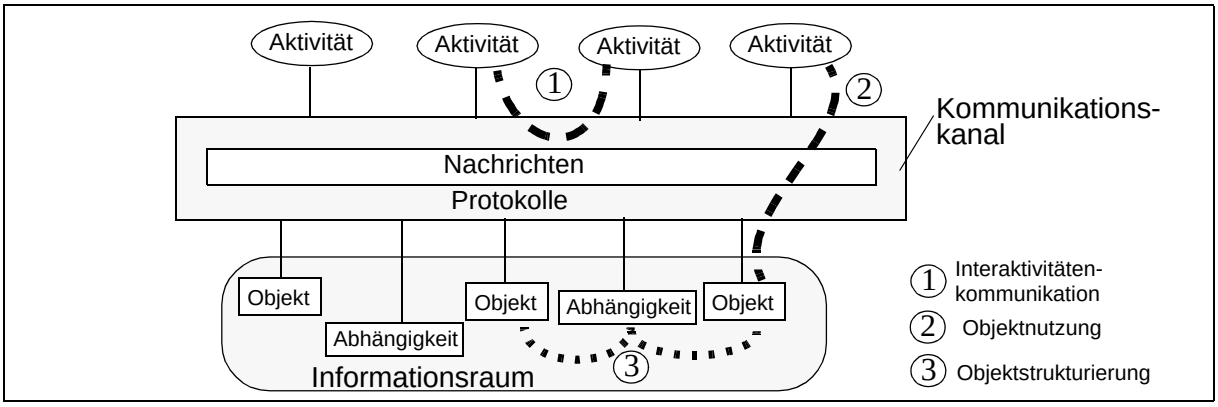


Abbildung 62: Nachrichten und Protokolle für Verhandlungen

4.7.1 Protokolldefinition

Ein Protokoll läßt sich durch einen Zustandsübergangsgraphen beschreiben. Eine *Konversation* ist die Durchführung einer Interaktion zwischen mehreren Kommunikationspartnern, die ein gemeinsames Protokoll verwenden. Ein Protokollgraph kann aus zwei verschiedenen Arten von Zuständen bestehen: *wartende Zustände* und *interne Zustände*. Die internen Zustände lösen den nächsten Zustandsübergang selbst aus (z.B. durch das Senden einer Nachricht), wohingegen wartende Zustände auf ein Ereignis (d.h. in der Regel eine Nachricht) warten, um den nächsten Übergang zu schalten. Zwei Teilnehmer in einer Konversation tauschen somit über Paare von wartenden Zustand/internen Zustand Nachrichten aus. Dabei ist zu beachten, daß ein Zustand nicht unbedingt eine Nachricht an einen Kommunikationspartner schicken muß, sondern auch zur Abwicklung interner Abläufe spontane Übergänge veranlassen kann.

Die Kanten eines Protokollgraphen werden durch Nachrichten präsentiert, die als zu senden bzw. zu empfangen gekennzeichnet sind, z.B. *Send: WRITE*. Eine Nachricht besitzt die Merkmale:

- Kennzeichnung des Senders,
- Angabe der Empfänger,
- ein Kommunikationsprimitiv, welches die Art der Anfrage kennzeichnet (z. B. *WRITE*),
- ein Identifikator (*Kontext-ID*), der die Nachricht einer Konversation zuordnet und
- weitere Parameter und Argumente (Name/Wert-Paare, z.B. [NEW_VALUE, 527833]).

Ein Zustand implementiert Aktionen, die bei dessen Erreichen ausgeführt werden. Danach werden die Nachbedingungen geprüft, die entsprechend einen Übergang erlauben, und gegebenenfalls eine Nachricht verschicken. Es muß ein definierter Ausgangszustand und mindestens ein Endzustand existieren. Es können beliebig viele Zwischenzustände definiert sein.

Ein Interaktionsprotokoll zweier Parteien besteht aus einem *Initiatorprotokoll* und einem *Komplementärprotokoll*. Im Falle von mehr als zwei Kommunikationspartnern existiert eine Menge von Komplementärprotokollen. Diese Protokolle ergänzen sich in dem Sinne, daß die sendenden Zustandsübergänge des einen Protokolls den empfangenden des/der anderen gegenüberstehen. Betrachten wir zwei Kommunikationsparteien, so sind im einfachen Fall die Zustände und Kanten reziprok im Sinne, daß die Sende-Kanten der einen Partei den Empfangskanten der anderen entsprechen und wartende Zustände den aktiven gegenüber stehen. Jedoch können in Protokollen auch weitere Zwischenzustände und -kanten definiert sein, die aber nicht zur Kommunikation mit dem Pendant dienen, sondern z. B. internen Abläufen oder der Kommunikation mit dritten Parteien.

4.7.1.1 Beispiel: Generisches Verhandlungsprotokoll

Da die vorherige Beschreibung von Protokollen recht abstrakt ist, soll dies an dem Beispiel eines allgemeinen Verhandlungsprotokolls illustriert werden (siehe Abbildung 63). Über dieses Protokoll können zwei Kommunikationsteilnehmer einen Vorschlag verhandeln. Der Initiator (Verhandlungspartner 1) stellt seinen Vorschlag mittels der Nachricht *propose*, wobei der Vorschlag in den weiteren Parametern der Nachricht spezifiziert wird. Dadurch kann die Verhandlungsphase beginnen, in der die beiden Parteien sich Änderungsvorschläge schicken können (*modify/propose*), bis der Vorschlag schließlich vom Verhandlungspartner 2 endgültig abgelehnt oder angenommen wird (*accept/reject*). Wie man sieht ist die Struktur der beiden Graphen identisch, bis auf die umgekehrten *Send*- und *Receive*-Kanten.

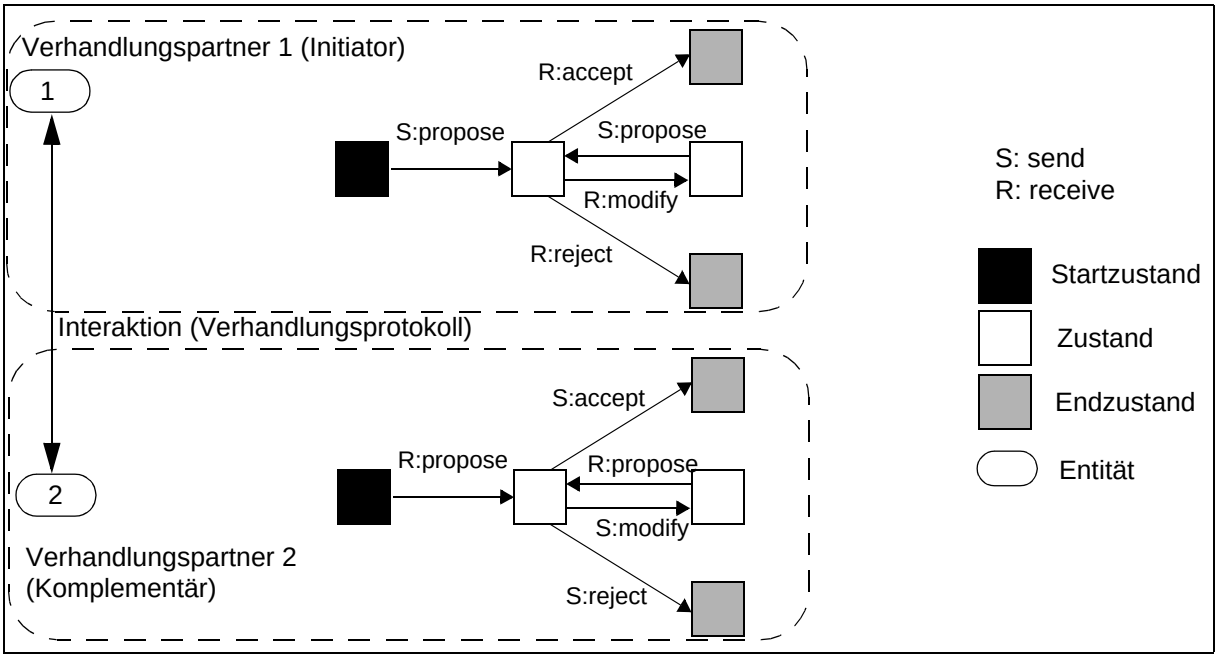


Abbildung 63: Protokoll für die allgemeine Verhandlung

4.7.1.2 Spektrum an Protokollen

Für die verschiedenen Interaktionen (z.B. Objektzugriff, Delegation usw.) sind entsprechende Protokolle zu definieren. Wir unterscheiden auch verschiedene Stufen von Protokollen bei einer Implementierung des ADM:

- *Generische Protokolle*
  Diese Protokollart erlaubt es, anwendungsspezifische Protokolle zu realisieren ohne ein neues Protokoll entwerfen zu müssen. Ein Beispiel ist die generische Verhandlung, die in Abbildung 63 dargestellt ist. Sie kann durch die Parameter einfach angepaßt werden. Bspw. kann eine darauf basierende Anfrage zur Rotation eines CAD-Elements durch die folgenden Parameter erreicht werden: *propose, [TOPIC, ROTATE], [ANGLE, 42.3]*
- *Allgemein verfügbare ADM-Protokolle, wie Objektzugriff oder Delegation*
  Wir definieren bestimmte Protokolle vorab, wie etwa die Delegation, die generell im Rahmen des ADM verfügbar sind und so in einer Implementierung auch zur Verfügung stehen sollen.
- *Spezifische Protokolle, wie Zugriffe auf spezifische Datenobjekte (z.B. CAD-Element)*
  Durch spezifische Protokolle können Interaktionen optimal an die Gegebenheiten einer Anwendung angepaßt werden. So kann ein CAD-Objekt andere Verhandlungsprotokolle als ein Dokumentenobjekt aus dem Dateisystem verwenden.

Durch diese Strukturierung kann ein ADM-System zum einen vorab Protokolle definieren, die spezifisch für dessen Einsatz geeignet sind. Andererseits steht eine Basis an Protokollen zur Verfügung, die bei neu auftretenden Anforderungen für Interaktion genutzt werden kann, ohne die Protokolle an sich definieren zu müssen. Der zweite Fall bedarf natürlich der Möglichkeit die Parametrisierbarkeit der Nachrichten auch ad hoc ausnutzen zu können. Dies kann zum Beispiel durch das Eingreifen der menschlichen Akteure geschehen, denen dann eine entsprechende Schnittstelle zur Verfügung stehen muß.

4.7.2 ADM-Protokolle

Im folgenden werden wir die grundlegenden Protokolle für das ADM beschreiben. Das allgemeine Verhandlungsprotokoll wurde bereits in Abschnitt 4.7.1.1 gezeigt. Wir werden nun Protokolle zur Delegation und Objektnutzung vorstellen.

4.7.2.1 Delegation

Wir haben in den Anforderungen bereits die Zerlegung des Entwurfsartefakts und die darauf aufbauende Delegation als wichtig für den Entwurf erkannt. Die Delegation beschreibt den Prozeß der Abgabe von Unteraufträgen an andere Aktivitäten. Im Falle des ADM beinhaltet dies jeweils die Übergabe der Entwurfsobjekte mit einer Spezifikation, die den Zielzustand für das Objekt beschreibt. Damit kann das typische “divide et impera” von Entwurfsprozessen nachvollzogen werden. Die Delegation kann man in drei Phasen unterteilen:

- *Aufbau der Delegationsbeziehung:*
  Eine delegierte Aktivität erhält einen Unterauftrag zur Bearbeitung. Dieser kann dabei verhandelt werden. Dies unterstützt die Idee, daß der Unterauftrag an einen Spezialisten gestellt wird, der die Erfolgsaussichten für den spezifizierten Auftrag selbst am besten abschätzen kann. Bei einer Delegation werden typischerweise Spezifikation und das dazugehörige Entwurfsobjekt (bzw. eine Referenz darauf) übergeben.
- *Durchführung des Auftrags*
  In dieser Phase kann es zu Änderungen an der Spezifikation kommen, die entweder durch die delegierende Aktivität mitgeteilt werden können, aber auch durch die delegierte Aktivität. Dies ist etwa der Fall bei sich ändernden Anforderungen, die “nach unten” propagiert werden oder, wenn sich der Unterauftrag als nicht realisierbar erweist und die delegierende Aktivität um die Änderung der Spezifikation gebeten wird. Während dieser Phase wird auch die Nutzungsbeziehung zur Bearbeitung des Entwurfsobjekts angewendet, dies ist jedoch nicht Teil des Delegationsprotokolls.
- *Beendigung der Delegation*
  Der Abschluß kann von beiden Seiten aus erfolgen. Da die delegierende Aktivität die Integration der verschiedenen Entwurfsobjekte gewährleisten sollte, können natürlich Probleme auftreten, die eine Überarbeitung eines oder mehrerer Entwurfsobjekte erfordert. Dies wird in dem Protokoll entsprechend berücksichtigt.

Definition der Entwurfszielspezifikation

Das ADM schreibt für die Spezifikation in einer Delegation bewußt keine formale Darstellung vor. Dies soll sicherstellen, daß das ADM an verschiedenste Anwendungsszenarien angepaßt werden kann. Beispielsweise läßt sich das Schreiben eines Briefs inhaltlich kaum formal fassen, und es bietet sich daher eher eine textuelle Beschreibung an (z.B.: “Entschuldigung an Kunde über die verzögerten Prototypen”). Andererseits ist es gerade bei technischen Entwurfsaufträgen oft möglich den Auftrag formal zu beschreiben. Beispiele hierfür sind in [RITTER 1997B] zu finden, wie etwa Prädikate über OQL (Object Query Language).

Aufbau einer Delegationsbeziehung

Das Protokoll zum Aufbau der Delegation ist ein Interaktivitätenprotokoll und in Abbildung 64 dargestellt. Dieses ähnelt sehr dem Verhandlungsprotokoll, jedoch ist das Kommunkationsprimitiv der Anfrage *delegate* und die Verhandlungsphase ist von beiden Verhandlungsparteien beendbar (*reject/accept*). Dies ist notwendig, da die Modifikation der Spezifikation (*refine*) von beiden Seiten aus stattfinden kann.

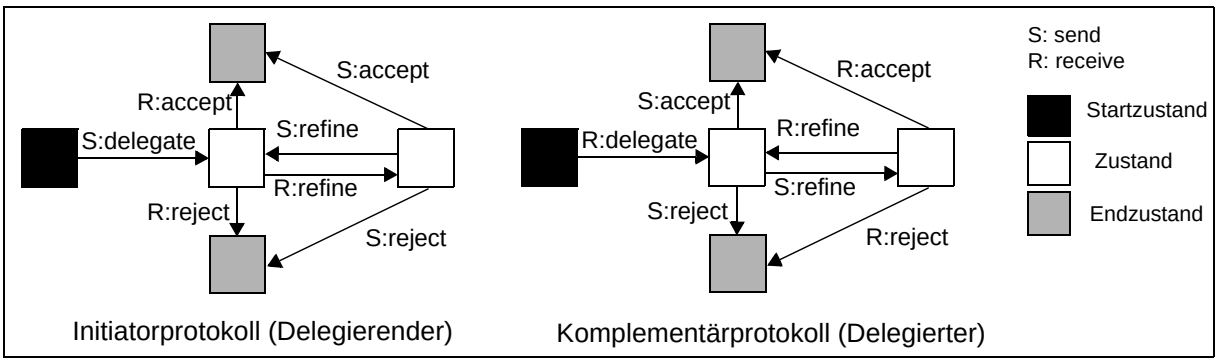


Abbildung 64: Delegationseinleitung

Die Nachricht zum Aufbau der Delegation hat folgende Struktur:

```
DELEGATE,<context_ID>, OBJECT, <object_id>, SPECIFICATION, <spec_data>
```

Der Aufbau der *refine*-Nachricht entspricht dem delegate, jedoch ist das Kommunikationsprimitiv hier *refine* mit einer geänderten Spezifikation:

```
REFINE,<context_ID>, OBJECT, <object_id>, SPECIFICATION, <changed_spec_data>
```

Die Spezifikation kann somit in der Verhandlungsphase geändert werden. Bei einer Ablehnung schlägt die Delegation fehl.

Durchführungsphase

Zwischen Aufbau und Ende der Delegation werden typischerweise das Entwurfsobjekt und andere benötigte Objekte des Informationsraums bearbeitet. Nun kann es jedoch zu Konflikten oder geänderten Voraussetzungen kommen, die eine Änderung der Spezifikation erfordern. Die Spezifikation kann daher auch verhandelt werden. Dies trifft vor allem bei zwei Bedingungen zu:

1. Die Spezifikation ist nicht erfüllbar, d.h. die delegierte Aktivität verlangt eine Änderung, die zu einer erfüllbaren Spezifikation führt.
2. Die delegierende Aktivität verlangt aufgrund bestimmter Ereignisse (z.B. geänderten Vorgaben), daß die Spezifikation geändert wird.

Die Verhandlungen werden dann mit den typischen Verhandlungsprimitiven geführt und entsprechen daher dem Verhandlungsprotokoll mit einem Parameter für die Spezifikation. Damit ist diese Phase auch der Verhandlungsphase beim Aufbau der Delegation ähnlich und wird auch nicht näher erläutert.

Beendigung einer Delegation

Ist eine Spezifikation erfüllt, wird die delegierende Aktivität notifiziert. Im Falle einer formalen Spezifikation kann dies automatisch durch die delegierte Aktivität geprüft werden. Ansonsten ist dies durch den Akteur zu veranlassen. Es kann aber auch die delegierende Aktivität die Beendigung des Auftrags verlangen. Diese entscheidet in beiden Fällen, ob die Spezifikation aus-

reichend erfüllt wurde und ob die Spezifikation geändert werden muß, falls bei der Integration der Teilentwürfe Fehler auftraten. Diese Änderung ist wie zuvor auch weiter verhandelbar, wie in Abbildung 65 (*redelegate/refine*) dargestellt.

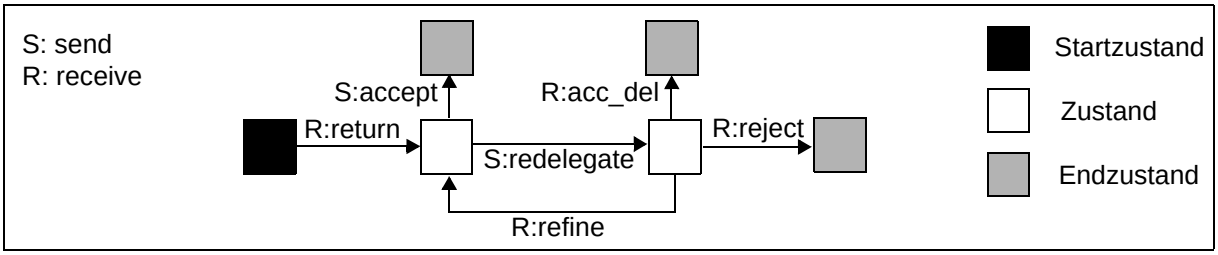


Abbildung 65: Protokoll zur Beendigung einer Delegation (Komplementärprotokoll)

4.7.2.2 Nutzungsprotokoll

Wir haben zuvor die Delegation beschrieben. In deren Durchführungsphase finden Objektzugriffe statt. Die Durchführung der Zugriffe sind im ADM durch eine vorherige Registrierung durch das sogenannte Nutzungsprotokoll möglich. Die Nutzungsbeziehung umfaßt den gesamten Zyklus von Anmeldung, Zugriff und Abmeldung. Das Nutzungsprotokoll ist somit in drei Phasen unterteilt:

- Bei einer Registrierung einer Aktivität auf einem Objekt wird dem Objekt die sogenannte Nutzungsanfrage zugeschickt. In dieser Nachricht sind als Parameter auch gewünschte Eigenschaften dieser Registrierung enthalten, wie etwa die Zugriffsrechte. Im Rahmen einer optionalen Verhandlungsphase werden diese Parameter ggf. noch nachverhandelt. Das Objekt kann letztendlich eine Nutzung annehmen oder ablehnen. Eine Ablehnung kann etwa dann erfolgen, wenn die Aktivität exklusive Schreibrechte verlangt, diese aber nicht verfügbar sind.
- Nach der Registrierung kann auf das Objekt zugegriffen werden. Die Protokolle zum Objektzugriff werden eingehend im folgenden Abschnitt behandelt. Zudem erfolgt in dieser Phase die Notifikation über Änderungen der Objekte.
- Zuletzt wird die Nutzung beendet. Dies kann sowohl durch das Objekt geschehen, wenn dieses beispielsweise aus dem System entfernt wird, oder durch die Aktivität, wenn diese das Objekt nicht mehr benötigt oder die Aktivität beendet wurde.

In Abbildung 66 wird das Nutzungsprotokoll dargestellt, das auch die Notifikation (*notify*) und Freigabe (*free*) des Objekts realisiert. Die Freigabe eines Objekts kann durch beide Entitäten (Aktivität oder Objekt) erfolgen. Die Möglichkeit die Freigabe zu erzwingen ist für das Objekt erforderlich, z.B. wenn es aus dem Informationsraum entfernt wird. Das Protokoll bietet keine Möglichkeit Objektzugriffe durchzuführen. Diese werden mittels spezieller Objektzugriffsprotokolle durchgeführt (s.u.).

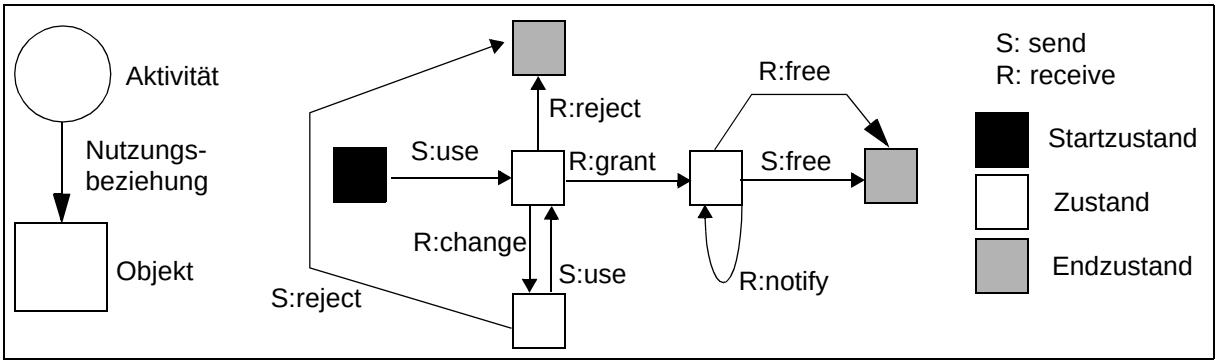


Abbildung 66: Das Nutzungsprotokoll (Initiatorprotokoll, d.h. Aktivitätssicht)

Durch die Dynamik unseres Modells können zu einem späteren Zeitpunkt Änderungen der Parameter einer Nutzungsbeziehung notwendig werden (z.B. eine Änderung an den Zugriffsrechten). Die Nachverhandlungen über solche Parameter in der Nutzungsbeziehung werden nicht in dem Nutzungsprotokoll erbracht. Sie erfolgen in eigenen Verhandlungen, da hier ggf. auch noch zusätzliche Entitäten involviert sein können, etwa bei einer Übergabe von Rechten.

Wir betrachten nun ein Beispiel der Nutzungsanfrage genauer und zeigen welche Parameter hier von Bedeutung sind. Dazu wird angenommen, daß das Objekt ein CAD-Element kapselt, welches exklusive Rechte und Leserechte anbietet. Die Aktivität will nun eine exklusive Sperre *X* erwerben. Dadurch erhält das CAD-Objekt folgende Parameter in einer Anfrage:

```
USE, <context ID>, [PRIVILEGE, 'X']
```

Nun kann es passieren, daß das CAD-Objekt bereits von einer anderen Aktivität mit dem Zugriffsrecht *"R"* genutzt wird, und dementsprechend unverträglich mit der exklusiven Sperre ist. Daher schlägt das Objekt eine Modifikation vor:

```
CHANGE, <context ID>, [PRIVILEGE, 'R']
```

Die Aktivität kann nun entscheiden, ob sie weiterhin auf *"X"* besteht und nochmals mit der gleichen Anfrage wie zuvor antwortet, oder ob sie die Anfrage in *"R"* ändert, oder ob sie das Protokoll mit einer *reject*-Nachricht abbricht. Bei einer erneuten *"X"*-Anfrage kann das Objekt entscheiden, ob sie diese ablehnt (*reject*) oder selbst in Verhandlung mit der sperrenden Aktivität tritt, um evtl. doch der anfragenden Aktivität das Recht gewähren zu können. Eine *"R"*-Anfrage kann das Objekt allerdings sofort annehmen (wenn nicht inzwischen eine andere Aktivität eine X-Sperre erlangt hat).

Objektzugriffe

Ein wichtiges Szenario für das ADM ist der Zugriff auf gemeinsame Daten des Informationsraums. Aktivitäten, die sich bei Objekten registriert haben und somit eine Nutzungsbeziehung eingegangen sind, werden diese auch für Operationen auf dem Objekt verwenden. Am Beispiel eines Datenobjekts, wie oben beschrieben, wären dies etwa folgende Funktionen: *READ*, *WRITE*, *DELETE*. Im folgenden werden wir die Lese- und Schreiboperationen vertiefen. Dabei ist zu beachten, daß diese für andere Objekttypen auch entsprechend der Objekteigenschaften variieren. Man kann diese Operationen auch in *kritische* und *unkritische* unterteilen. Kritische Operationen zeichnen sich durch potentielle Nebenläufigkeitskonflikte aus, unkritische Operationen erfordern keine Nebenläufigkeitskontrolle. Im Falle von Datenbanken existieren dafür entsprechende Synchronisationsverfahren, andere Objekttypen müssen entsprechende Nebenläufigkeitsverfahren selbst realisieren. Die Protokolle für Zugriffe können also den anwendungsbezogenen bzw. objektbezogenen Gegebenheiten angepaßt werden (z.B.: Sperrverfahren, Voting-Verfahren oder Floor-Control-Verfahren).

Unkritische Operationen

Als Beispiel für eine unkritische Operation, betrachten wir eine Nachricht zum Abfragen des Attributwerts 'COLOR' eines CAD-Elements:

```
READ, <context ID>, [ATTRIBUTE, 'COLOR']
```

Ist diese Anfrage erfüllbar, da die Aktivität Leserechte hat, so kann die Antwort lauten:

```
VALUE, <context ID>, [ATTRIBUTE, 'COLOR'], [RGB_STRING, "110,256,0"]
```

Aufgrund der einfachen Struktur solcher Protokolle werden wir diese hier nicht genauer untersuchen, sondern die Schreiboperation betrachten, da eine kritische Operation weitere zu berücksichtigende Eigenschaften besitzt.

Kritische Operationen

Kritische Operationen (z.B. Rotieren eines CAD-Objekts) erfordern eine Regelung dieser Zugriffe, inbesondere bei gemeinsamen Zugriff. Eine Schreiboperation auf ein Objekt kann durch ein (aufwendiges) Zugriffsprotokoll definiert sein. Wir betrachten hier verschiedene Implementierungen für ein Datenobjekt. Ein entsprechendes Protokoll kann auf verschiedene (der Anwendung angepaßte) Arten definiert werden, was durch drei Beispiele illustriert wird:

- Betrachten wir zunächst das einfache Schreibprotokoll in Abbildung 67. Hier wird eine Schreibanfrage gestellt (*write*). Daraufhin entscheidet das Objekt, ob die Anfrage durchgeführt werden soll (1), z.B. anhand vergebener Zugriffsperren. Falls die Operation nicht erlaubt ist, oder doch noch fehlschlägt, wird dies gemeldet (2). Dabei können etwa Referenzen zu den blockierenden Aktivitäten übermittelt werden. Diese Aktivitäten können dann im Rahmen einer weiteren Verhandlung aufgefordert werden die Schreibrechte zu übertragen. Im Falle einer erfolgreichen Schreiboperation, wird dies bestätigt, und es können gegebenenfalls andere registrierte Aktivitäten über die Objektänderung informiert werden (3).

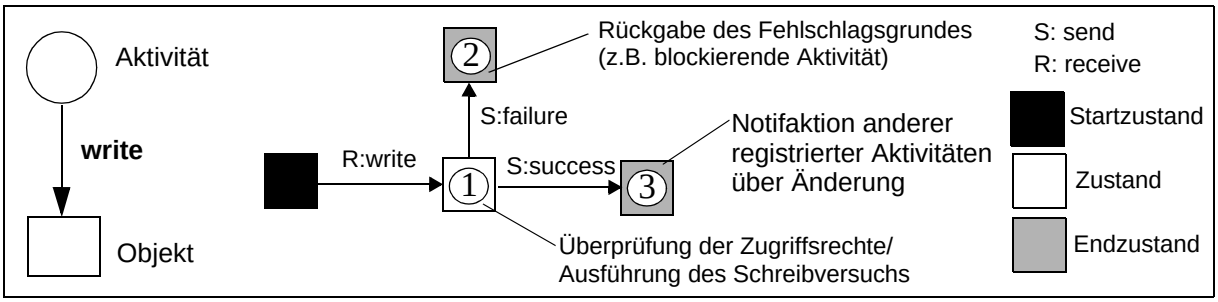


Abbildung 67: Schreibprotokoll für Datenobjekte

- In Abschnitt 3.7.2 wurde TOGA vorgestellt, das gemeinsame Informationen nur nach Abstimmung mittels eines *Two Phase Commit* (2PC) ändert. Eine Abstimmungsphase kann auch im ADM erreicht werden, wenn man das einfache Schreibprotokoll um ein Abstimmungsprotokoll erweitert, wie in Abbildung 68 dargestellt. Dort bewirkt die Aktion des Zustandes (1) aus Abbildung 67 den Aufruf eines Abstimmungsprotokolls. Im Rahmen dieses Aufrufs werden die Aktivitäten der Kooperationsmenge um die Abgabe ihrer Entscheidung bezüglich der Schreibanfrage aufgefordert. Erst wenn dieses mit einem positivem Bescheid (*execute*) beendet ist, wird das übergeordnete Protokoll entsprechend beendet (*success/failure*) und die beteiligten Entitäten über die Entscheidung benachrichtigt. Durch diese Schachtelung der Protokolle ist eine einfache Erweiterung von Protokollen mittels Verwendung bereits existierender Protokolle möglich.
- Als letztes Beispiel für eine Schreibanfrage wird ein Protokoll beschrieben, das eine weitergehende Autonomie für ein Objekt realisiert. Das Objekt versucht dabei eine Schreibanfrage möglichst durchzuführen. Falls also im Rahmen der Prüfung des Zugriffs Konflikte erkannt werden (Abbildung 67(1)), kann das Objekt selbstständig in Verhandlung mit den blockierenden Aktivitäten treten und eine Übertragung der Schreibrechte fordern. Diese Verhandlungen erfolgen durch Protokollaufrufe, ähnlich wie in Abbildung 68 im Falle des Abstimmungsprotokolls, nur daß ein Verhandlungsprotokoll gestartet wird (vgl. Abschnitt 4.7.1.1). Dadurch kann man die Last der Sperrenverhandlung der Aktivität auf das Objekt verlagern.

An diesen drei Beispielen kann man sehen, daß zwar immer das strukturell gleiche Schreibprotokoll angewandt wurde, jedoch dieses durch geschickte Implementierung der Aktionen oder Erweiterung durch andere Protokolle, spezifischen Anforderungen angepaßt wurde. Dadurch ist

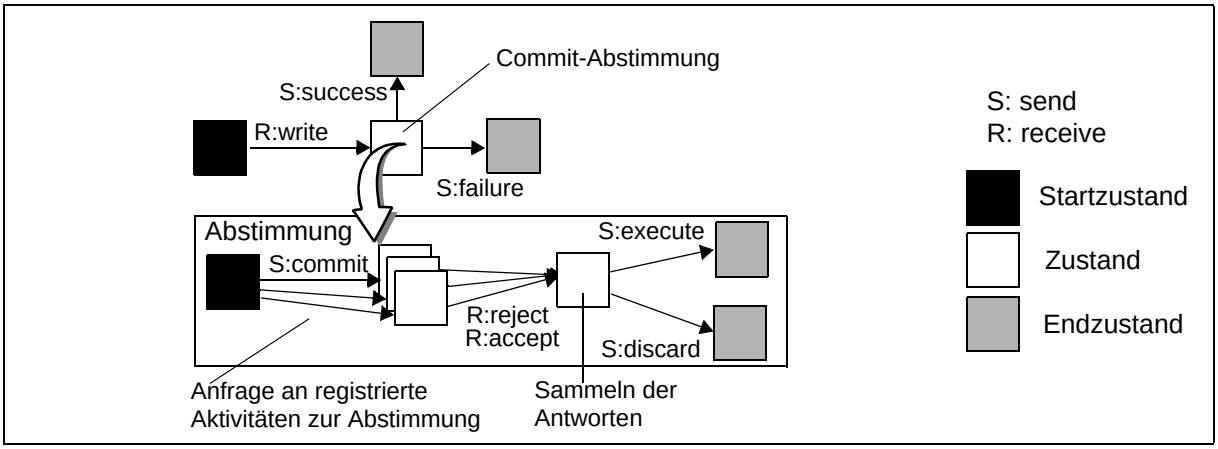


Abbildung 68: Schreibprotokoll mit Abstimmungsphase

es möglich, bei einer Implementierung der Schreibprotokolle die Eigenheiten der verschiedenen Objektarten zu berücksichtigen, wobei der Zugriff nicht geändert werden muß. Wir werden diese Aspekte in Abschnitt 4.7.3 eingehender erläutern.

4.7.3 Anpaßbarkeit von Protokollen

Um das ADM an spezifische Anforderungen anzupassen, ist es notwendig die Protokolle einfach wiederzuverwenden und erweitern zu können. Dies kann man durch die Schachtelung von Protokollen und entsprechend implementierte Aktionen erreichen. Zudem stellen wir uns eine generische Protokollmaschine vor, die einfach in die Entitäten (also Aktivitäten und Objekte) des ADM integriert werden kann (siehe Abbildung 69). Eine Entität besitzt zwei wesentliche Schnittstellen. Die Systemaufruf-Schnittstelle realisiert die spezifischen Objektzugriffe und Verwaltungsfunktionen (z.B. Systemanmeldung). Die Interaktionsschnittstelle erhält Nachrichten, die durch die Protokollmaschine abgearbeitet werden. In den Protokollen werden die Aktionen aufgerufen, die das Bindeglied zur Entitätenimplementierung darstellen und somit letztendlich die Zugriffe realisieren. Wir beschreiben im folgenden auf welche Weise eine Anpassung der Protokolle durchgeführt werden kann.

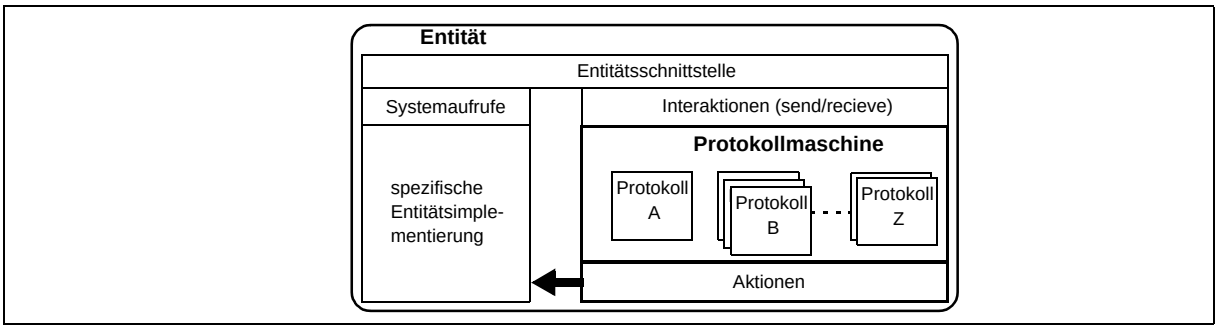


Abbildung 69: Einbettung einer Protokollmaschine in Entitäten

Bei den Beispielen können wir vordefinierte Protokolle und anwendungsspezifische Protokolle unterscheiden. Das Nutzungsprotokoll ist ein vordefiniertes Protokoll, das in jeder ADM-Implementierung zu berücksichtigen ist. Die Objektzugriffe können hingegen anwendungsspezifisch implementiert werden. Dies ist auf mehreren Ebenen möglich. Es sind damit beispielsweise ein generelles Protokoll für Datenbankobjekte und spezielle Varianten für einzelne Datenbankobjekte realisierbar (wie oben beschrieben). Wir haben folgende Möglichkeiten zur Entwicklung von Protokollen:

- Einführung eines neuen Protokolls

- Nutzung eines existierenden Protokolls mit Veränderung der Aktionen der Zustände:
  Dadurch kann diese Protokoll auch von der Gegenseite ohne Änderungen weiterhin genutzt werden, da die Übergänge (und damit die Nachrichten) beibehalten werden.
- Nutzung eines existierenden Protokolls mit Änderung von Aktionen und Nachrichtenparametern:
  Hierbei muß die Gegenseite auf die geänderten Parameter reagieren können, der Ablauf an sich ändert sich nicht, da die Übergänge nur durch das Kommunikationsprimitiv bestimmt sind.
- Transformation in ein neues Protokoll durch Einführung neuer Kommunikationsprimitive:
  Dies erfordert mehr Aufwand, da alle Teilnehmer jeweils diese Primitive verstehen müssen. Daher fordern wir für die Implementierung eine textuelle und/oder grafische Definition durch die derartige Änderungen leicht durchführbar sind.

Somit sind die Zustandübergangsdiagramme und die Aktionen der Zustände unabhängig voneinander implementierbar. Dies verschafft uns eine Abstraktionsebene, die sowohl die Wiederverwendbarkeit von Aktionen und Protokollen erhöht, als auch die durchzuführenden Änderungen minimiert, da eine alleinige Änderung der Aktionen nicht zwangsweise eine Änderung auf der Gegenseite nach sich zieht. Daher empfiehlt es sich bei einer ADM-Implementierung zu versuchen, möglichst viele Interaktionsprotokolle strukturell gleich zu halten. Damit ist die Interoperabilität verschiedenster Entitäten leichter zu erreichen, da nur Aktionen anzupassen sind. Diese Vorgehensweise entspricht der Polymorphie.

Wir wollen dies anhand des Schreibprotokolls zeigen. Dabei wurde auch versucht das eigentliche Protokoll (siehe Abbildung 67) für alle Implementierungen strukturell zu erhalten. In Abbildung 70 wird das Schreibprotokoll mit verschiedenen Implementierungen einer Aktion gezeigt und die Ausnutzung existierender Protokolle dargestellt. Verschiedene Aktionsimplementierungen werden durch die Nummern (1) bis (4) angezeigt. Wie man sieht, sind die Übergänge des Protokolls nicht zu ändern, da diese wie vorgesehen geschaltet werden (z.B. *transit(FAILURE)* bei (1), was auch die entsprechende Nachricht versendet). Die interne Realisierung der Validierung des Schreibzugriffes erfolgt aber unterschiedlich:

1. In dieser Aktion wird das Zugriffsrecht überprüft und nur bei einer exklusiven Sperre (*X*) wird der Schreibzugriff durchgeführt.
2. Diese Aktion versucht einfach den neuen Wert zu schreiben und schaltet je nach Gelingen den entsprechenden Übergang.
3. In dieser Implementierung wird der Aufruf eines Benutzerdialogs angedeutet (`showYESNODialog()`), der als Ergebnis die entsprechende Berechtigung zurückliefert.
4. Hier wird nun eine Objekteigenschaft überprüft (*new_data > 50*) um zu entscheiden, ob weitere Aktivitäten dieser Änderung zustimmen müssen. Ist eine Abstimmung notwendig, so wird das 2PC-Protokoll aufgerufen (5). Andernfalls wird entsprechend (2) verfahren. Dies läßt sich zum Beispiel bei der Änderung eines Kostenwertes sinnvoll einsetzen, der ein bestimmtes Budget nicht überschreiten sollte.

Ähnlich wie bei Aktion (3) kann auch sofort eine Schachtelung von Protokollen zugelassen werden (6), ohne eine gesonderte Aktion zu implementieren, wie dies schon in Abschnitt gezeigt wurde. Wir etablieren hier das Konzept des *Protokoll-Reservoirs*, das eine Menge von Protokollen enthält, die in anderen Protokollen wiederverwendet werden können. Dies stellt auch die Basis für das komplexe Zugriffszenario aus Abschnitt 4.6.4 dar, das durch eine Kaskadierung von Protokollen gekennzeichnet ist.

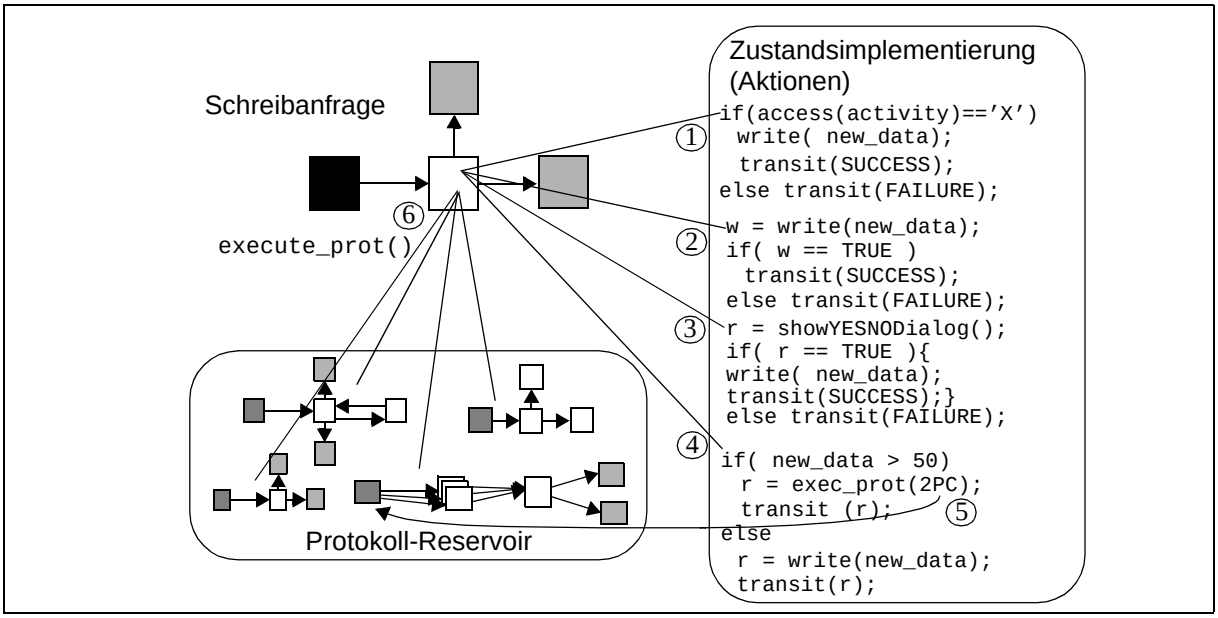


Abbildung 70: Protokoll-Reservoir und Aktionsimplementierungen

4.7.4 Protokolle als Kooperationsbasis und integrierender Aspekt des ADM

Wie an den ausgewählten Beispiele gezeigt, stellen die Protokolle ein mächtiges Instrument des ADM dar. Durch diese können Interaktionen zwischen Akteuren, Aktivitäten und Objekten geführt werden. So sind die Delegationsprotokolle halb-automatische Protokolle, da hier oft auch Benutzerentscheidungen einfließen können, wenn beispielsweise die Spezifikation geändert werden muß. Ein Beispiel für so eine Einbindung wurde oben gegeben. Eine Implementierung findet sich in Abschnitt 5.8.5.

Die Protokolle realisieren zudem eine Integrationsschicht zwischen den Bestandteilen des ADM und verbinden diese auf flexible Weise. Sie unterstützen die Koordination durch Delegation und Zugriffsregelung und ergänzen damit das Aktivitätenmodell. Außerdem erreichen die Protokolle die Interaktion zwischen den verschiedenen gekapselten System (z.B. Workflow-Management und Groupware). Die Aktionen stellen einen wichtigen Aspekt zur Integration dar, da durch diese die spezifischen Funktionen von Entitäten genutzt werden können, ohne die konzeptionelle Ebene der Protokolle zu beeinflussen. Eine Workflow-Aktivität implementiert möglicherweise andere Aktionen als eine Groupware-Aktivität und sicherlich andere als Objekte. Damit können anwendungsspezifische Eigenheiten berücksichtigt werden, während die Kommunikation ohne Änderung der Protokolle selbst durchgeführt werden kann.

4.8 Zusammenfassung

Zum Abschluß des Kapitels gehen wir hier zusammenfassend auf die vorgestellten Konzepte ein und stellen dar, was durch sie erreicht wird.

4.8.1 Aktivitätenmodell

Wie man sieht, werden bestehende Groupware- und Workflow-Konzepte übernommen und geschickt als konstruktive Teile einer Designflow-Spezifikation eingesetzt. Damit übernimmt das ADM die Fähigkeiten dieser bestehenden Technologien als auch neuer Aspekte aufgrund der Benutzung des gemeinsamen Informationsraums und der verfügbaren Verhandlungsfunktionalität. Unser Aktivitätenkonzept unterscheidet sich in den grundlegenden Eigenschaften zunächst nur unwesentlich von bekannten Workflow-Konzepten. Jedoch erfolgen die Behandlung

der Ablaufabhängigkeiten und die Ressourcenvergabe auf eine den Entwurfsprozessen angepaßte Weise, die so bisher nicht realisiert wurde. Die Aktivitäten sind in die Kooperationsunterstützung eingebunden, d.h. über den Informationsraum und die Interaktionsprotokolle, und durchbrechen damit systemseitig das Konzept isolierter Aktivitäten.

4.8.2 Informationsraum

Wichtige Merkmale der untersuchten kooperativen Prozesse sind die Bearbeitung gemeinsamer Ressourcen und die Delegation von (zerlegbaren) Entwurfsobjekten. Daher wurde ein gemeinsamer Informationsraum etabliert, der als Kooperationsbasis für gemeinsame Objekte dient. Aktivitäten gehen mit diesen Objekten sogenannte Nutzungsbeziehungen ein, um Zugriffe auf den Objekten durchführen zu können. Da eine Forderung die Zerlegung von Entwurfobjekten war, wurden hierarchische Informationsräume untersucht, die eine entsprechende Strukturierung unterstützen. Diese wurde jedoch abgelehnt, da die Hierarchisierung des Informationsraums die Dynamik unseres Modells deutlich erschwert hätte. Deswegen wurden Kooperationsmengen innerhalb des Informationsraums eingeführt, welche jeweils alle Aktivitäten umfassen, die das selbe Objekt nutzen. Um die Zerlegbarkeit von Entwurfsobjekten zu erreichen und weitergehende Eigenschaften zwischen Objekten beschreiben zu können, wurden Abhängigkeiten wie part-of eingeführt, welches den Aufbau komplexer Objekte gestattet.

4.8.3 Interaktionsprotokolle

Wenn wir die Kommunikation zwischen den Entitäten im ADM betrachten, so können wir dies in verschiedene Kategorien unterteilen. Dabei sind die menschlichen Benutzer auch indirekt als Entität im Modell zu sehen, die über Aktivitäten Zugriff besitzen. Wir werden diese Integration von Akteuren bezüglich der Implementierung in Abschnitt 5.8 beispielhaft zeigen. Die verschiedenen Kommunikationsarten definieren sich aus der Kombination der Kommunikationspartner und dem zugrunde liegenden Automatismus und spannen damit den Bogen der durch das ADM abgedeckten Kommunikationsarten:

1. **Direkte Kommunikation**
   Die Kommunikation erfolgt direkt zwischen Entitäten. Typischerweise ist dies eine bidirektionale Interaktion, wie sie etwa bei der Objektnutzung oder Verhandlung auftritt.

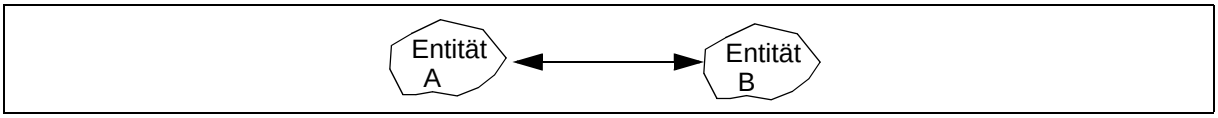


Abbildung 71: Direkte Kommunikation zwischen Entitäten

2. **Indirekte Kommunikation**
   Eine indirekte Kommunikation tritt auf, wenn die Auswirkung, die bspw. durch den Zugriff einer Aktivität auf ein Objekt veranlaßt wird, eine andere Aktivität beeinflußt. Dies kann beispielsweise der Fall sein, wenn ein Objekt verändert wird, und es alle registrierten bzw. abhängigen Entitäten benachrichtigt. Hier kann der Nachrichtenfluß unidirektional sein.

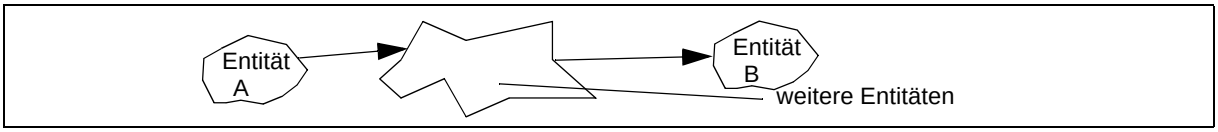


Abbildung 72: Indirekte Kommunikation zwischen Aktivitäten

3. **Automatisierte Kommunikation**
   Hier wird die Kommunikation ausschließlich durch die Entitäten ohne die Einflußnahme

von Personen durchgeführt. Es existieren vordefinierte und bereits implementierte Protokolle, die je nach den *Strategien* (Regeln nach denen eine Entität ein Protokoll durchführt) der beteiligten Entitäten genutzt werden.

4. **Protokoll-gestützte manuelle Kommunikation**
   Diese Art der Kommunikation dient der geregelten Interaktion mit der Beteiligung von menschlichen Akteuren, die im Rahmen eines Verhandlungsprotokolls miteinander kommunizieren. Es existieren Protokollmaschinen, die den Akteuren je nach Zustand der Kommunikation eine Auswahl an möglichen Kommunikationsprimitiven anbieten. Somit ist eine konsistente Anwendung des Protokolls gewährleistet. Dabei kann einer der Kommunikationspartner aber auch eine ADM-Entität (Aktivität, Objekt) sein.
5. **Semi-automatisierte Kommunikation**
   Diese Zwischenform tritt dann auf, wenn beispielsweise eine automatisierte Kommunikation an einen Punkt kommt, der eine Entscheidung eines (menschlichen) Akteurs verlangt. Dieser kann dann im Rahmen einer protokollgestützten manuellen Kommunikation eingebunden werden. Ein Beispiel ist ein durch existierende Strategien nicht auflösbarer Konflikt, der nur durch Einwirkung von außen gelöst werden kann.

Die vorgestellte Interaktionsunterstützung realisiert die Integration und Kooperation der verschiedenen ADM-Entitäten. Durch die flexible Ausgestaltung der Protokolle können auch verschiedene Anwendungen geeignet unterstützt werden, etwa an Objekte angepaßte Zugriffsprotokolle.

4.8.4 Übersicht der Kooperationsunterstützung

Ein wesentliches Merkmal des ADM ist der Kooperationsgedanke. Die Kooperation wird über die Nutzung gemeinsamer Objekte und direkte Verhandlungen zwischen Aktivitäten ermöglicht. Dazu werden verschiedene Mechanismen aus dem Bereich CSCW angewendet, wie in Tabelle 7 aufgeführt:

CSCW-Mechanismus	Ziel	Umsetzung
konkurrierende Nutzung von Objekten nach bestimmten Regeln (data sharing)	Kooperation auf diesen Objekten	Einbringung der Objekte in Informationsraum
implizite Kooperation auf Objekten	indirekte Koordination der Arbeitsschritte	Informationsraum mit Anwendung von Nutzungs- und Zugriffsprotokollen
explizite Kooperation	Direkte Verhandlung zwischen Entitäten und Akteuren	Einführung von Verhandlungsprotokollen und Einbindung menschlicher Akteure
Awareness	- Kenntnis über gemeinsame Objektnutzung und potentielle Konflikte - Delegation macht die Entwurfsabhängigkeiten offensichtlich	Zugriffskonflikte werden durch Informationsraum und Protokolle bekannt und regelbar Objektstruktur (z.B. Datenabhängigkeiten der komplexen Objekte) Notifikationsmechanismus bei Objektänderungen

Tabelle 7: CSCW-Mechanismen im ADM

Durch die Wahl der oben ausgeführten Funktionalität wird eine geeignete Unterstützung von Entwurfsprozessen erreicht. Betrachtet man dies aus der Workflow-Sicht, so können Teilprozesse, die vorab nicht definierbar sind durch die Koordination auf gemeinsamen Objekten, durch Verhandlungsprotokolle, durch Constraints und mittels der Delegation realisiert werden. Wir können somit die wichtigsten Aspekte unserer Anforderungen mit dem ADM beschreiben.

Es stellt sich nun die Frage wie das ADM umzusetzen ist. Dies umfaßt insbesondere die Integration existierender Systeme, wie es durch die Basisdienste erreicht werden soll. Wir werden diese Aspekte im folgenden Kapitel vertiefen und den Prototypen CASSY vorstellen.

5 Implementierung: Aspekte und Konzepte

In diesem Kapitel besprechen wir den Prototypen CASSY, eine prototypische Implementierung des ADM. CASSY basiert auf den drei wesentlichen Pfeilern des ADM: Aktivitäten-Management, Informationsraum und Interaktionsprotokolle.

5.1 Überblick

In Kapitel 4 wurde das ADM eingehend besprochen, und es wurden bereits Hinweise auf die Architektur des Prototypen CASSY gegeben. In diesem Kapitel detaillieren wir die Implementierung. Im folgenden Abschnitt wird ein Überblick der Architektur gegeben. Wir besprechen dort CORBA als Middleware für CASSY und spezifische Probleme, die uns bei einem Integrationsansatz erwarten. Danach beginnen wir mit der Beschreibung der wichtigsten Dienste, Facilities und Werkzeuge. In Abschnitt 5.3 wird das Objektmodell präsentiert auf dem CASSY basiert. Daraufhin besprechen wir in Abschnitt 5.4 die Event Engine. Die Implementierung der Workflow-Facility diskutieren wir, als Beispiel für die typischen Integrationsmöglichkeiten, eingehend in Abschnitt 5.5 und anschließend die Groupware-Facility in Abschnitt 5.6. Die letzte wesentliche Komponente in CASSY ist die Protokollmaschine, die in Abschnitt 5.7 behandelt wird. In Abschnitt 5.8 zeigen wir anhand einiger Beispiele die Benutzungsschnittstelle von CASSY. Wir fassen die wesentlichen Teile von CASSY in Abschnitt 5.9 zusammen und beschreiben kurz in Abschnitt 5.10 wichtige Abläufe bei der Durchführung von Designflows. Wir schließen das ganze mit einem Fazit in Abschnitt 5.11 ab.

5.2 Grundlagen der Implementierung

In Abschnitt 4.3.1 wurde ein schematischer Realisierungsansatz für das ADM vorgestellt. Hier beschreiben wir nun die technischen Aspekte zur Umsetzung dieses Realisierungsansatzes.

Da WFMS häufig dem Bereich CSCW zugeordnet werden, sollte es nicht sehr problematisch sein ein integriertes System zu erstellen. Jedoch kann man teils deutliche Unterschiede zwischen Workflow-Management und existierender Groupware bzw. allgemeinen CSCW-Bibliotheken feststellen (siehe Abschnitt 3.4.7). Gemeinsamkeiten lassen sich in Bezug auf eine Personenverwaltung, Datenverwaltung und Notifikationsmöglichkeiten erkennen. Aber es bestehen auch deutliche Differenzen zwischen kooperativen CSCW-Umgebungen und WFMS. Dies leitet sich unter anderem aus der Prozeßorientierung von WFMS gegenüber der Objektorientierung[1] von Groupware ab [SCHALLER UND SCHWAB 1997]. Im folgenden wird diese Problematik anhand einiger Punkte verdeutlicht.

Ein gemeinsames Aktivitätenmanagement aus diesen beiden Ansätzen zu realisieren scheint sehr aufwendig, da der offenere Ansatz von CSCW-Anwendungen und -Ressourcen der asynchronen und isolierten Behandlung in Workflow-Aktivitäten gegenübersteht. Groupware-Systeme verwenden einen gemeinsamen Informationsraum zur Realisierung kooperativen Arbeitens und zur Erhöhung des Gruppenbewußtseins. Dafür werden z. B. bei Lotus Notes Versionierungs- und Replikationsmechanismen eingesetzt. WFMS verwenden im Gegensatz dazu einen gerichteten Datenfluß, z.B. über die Ein- und Ausgabedaten von Aktivitäten und sichern so die Konsistenz dieser Daten zu. Offensichtlich besteht hier also ein wesentlicher Unterschied zum Informationsraum, der entsprechend bei einer Integration zu berücksichtigen ist. Dieser Punkt ist von zentraler Bedeutung für ein integriertes System, und es müssen entsprechende

1. Objektorientierung bezogen auf gemeinsame Objekte, wie z. B. Dokumente.

Konzepte realisiert werden, um ein Miteinander dieser Techniken zu bewerkstelligen. Weitere Bereiche, die nicht leicht verträglich sind, umfassen beispielsweise: Awareness, Applikationsfunktionalität oder auch die Unterstützung von direkter Kommunikation.

Die grundlegende Architektur von CASSY ist zum einen durch die Integration einer Workflow- und einer Groupware-Facility gekennzeichnet (siehe Abbildung 73). Durch das sogenannte Designflow-Management wird die Durchführung von Workflows bzw. Groupware-Anwendungen innerhalb von ADM-Aktivitäten ermöglicht. Diese werden um Informationsraum und Interaktionsprotokolle sowie ein übergeordnetes Aktivitäten-Management innerhalb des Designflow-Managements erweitert. Verwendete Basisdienste sind etwa ein Name Service und die Protokollmaschine.

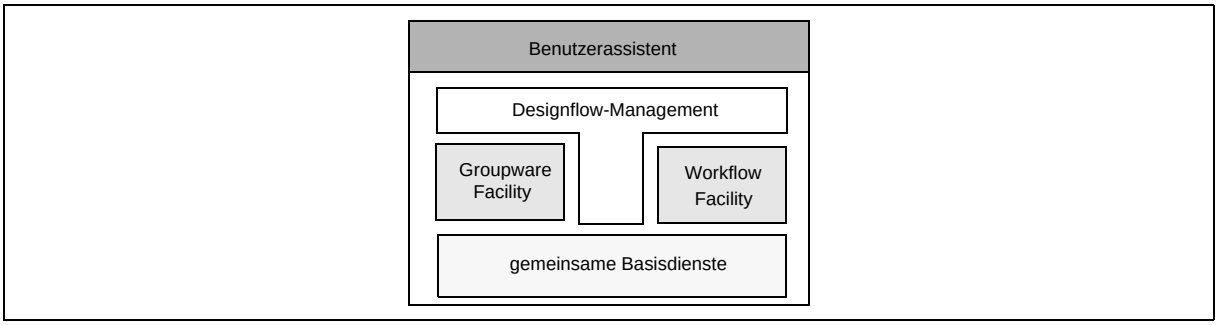


Abbildung 73: Einfaches Architekturbild für CASSY

CASSY bietet eine Schnittstelle für das Designflow-Management an, in der auch die Funktionen der Workflow- und Groupware-Facility zu integrieren sind. Daher ist eine geeignete Abbildung deren Funktionalität auf die CASSY-Schnittstelle durchzuführen (siehe Abschnitt 5.3). Ebenso sind weitere externe Programme, wie z.B. CAD-Werkzeuge oder Software-Entwicklungsumgebungen einzubinden. Diese unterscheiden sich durch ein fehlendes Aktivitätskonzept und eine in der Regel fehlende Beschreibung des Datenflusses oder der Datennutzung. Wenn diese also innerhalb von ADM-Aktivitäten genutzt werden, geschieht dies durch die ADM-eigene primitive Aktivität. Die Abbildung der für den Informationsraum relevanten Daten hat gesondert zu erfolgen, und die Art und Weise dieser Abbildung unterscheidet sich von System zu System (siehe Abschnitt 5.2.2).

Die übergeordneten Aktivitäten (Workflow-, Groupware-, und Designflow-Aktivität) kann man - wie links in Abbildung 74 dargestellt - als Hüllen auffassen, die jeweils ihre besonderen Eigenschaften im Rahmen der Designflow-Schicht des ADM zur Verfügung stellen und die enthaltenen Aktivitäten kapseln [MARIUCCI 1998]. Dadurch können die Aktivitäten existierender Systeme in unser Modell eingebunden werden. Zusätzlich erlaubt es eine solche Hülle, Aktivitäten um unsere Konzepte zu erweitern oder anzupassen (etwa Informationsraum, gemeinsames Organisationsmodell, Aktionen in Protokollen [FRANK UND MITSCHANG 2001]). Entsprechend kapseln Objekte die Ressourcen aus bestehenden Systemen mit einer Objekthülle (Abbildung 74, rechts). In diesen Hüllen wird die Protokollmaschine eingesetzt und durch Protokolle aus dem Protokoll-Reservoir und entitätsspezifischen Aktionen an das gekapselte System angepaßt.

Bei einer solchen Integration sind Entscheidungen bezüglich der grundlegenden Technologien und Architektur zu treffen, wie es im folgenden Abschnitt beschrieben wird. Die dabei zu verfolgende Methodik zur Einbindung existierender Systeme wird in Abschnitt 5.2.2 diskutiert. Wie dies auf ein Workflow-System und eine Groupware-Facility für CASSY angewendet wurde, wird in Abschnitt 5.5 bzw. Abschnitt 5.6 gezeigt.

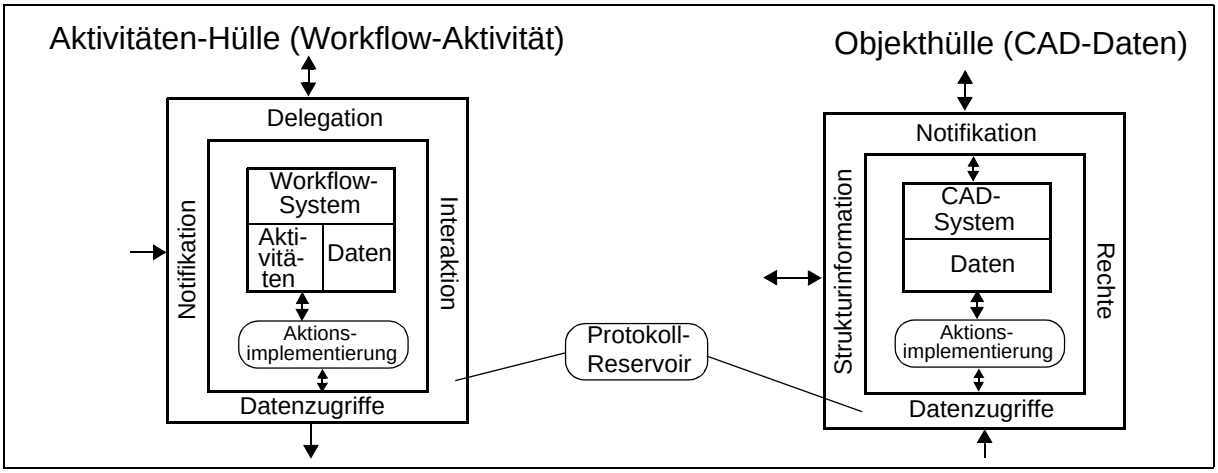


Abbildung 74: Hüllenkonzept anhand einer ADM-Workflow-Aktivität

Durch die beschriebenen Schritte können Aktivitäten und Daten in CASSY eingebracht werden. Um diese zu nutzen benötigen wir ein Aktivitäten-Management, die Informationsraumverwaltung und für deren Zusammenspiel eine Protokollmaschine. Letztlich ist auch eine Benutzungsschnittstelle notwendig, um Designflows zu erstellen und überwachen zu können. Diese Komponenten wurden eigens entsprechend dem ADM für CASSY entwickelt und wir besprechen diese ebenfalls in den folgenden Abschnitten.

5.2.1 CORBA als Middleware

Zuvor wurde das komponentenbasierte Konzept für CASSY erläutert. Um dieses konkret umzusetzen bietet CORBA[1] [OMG 1996] eine geeignete Middleware-Technologie an, da es die Erstellung verteilter Systeme in einer heterogenen Umgebung unterstützt. Durch den objektorientierten Ansatz können die Schnittstellen von Komponenten und Diensten definiert werden, die durch verschiedene Systeme realisiert sind. Dies resultiert in einem sogenannten *Wrapper-Ansatz*. CORBA nutzt die *IDL* (*interface definition language*) für eine Plattform- und Programmiersprachen-unabhängige Beschreibung von Objekten. Auf diese Schnittstellen sind die zu kapselnden Systeme abzubilden. Durch den Einsatz von CORBA können folgende Punkte realisiert werden:

- Leichter Zugriff auf die verschiedenen Funktionalitäten
- Dienstkonzept
- Unterstützung der Integration
- Erweiterbarkeit und Anpaßbarkeit.

In Abbildung 75 wird die von uns gewünschte Integrationsarchitektur dargestellt, die auf dem in Abbildung 43 auf Seite 81 illustrierten Konzept basiert. In der unteren Hälfte der Abbildung befinden sich die einzubindenden externen Systeme und Dienste (Workflow-System, Groupware-System, DBMS usw.). Die Basisdienste stellen allgemeine Dienste wie einen Name Service (hier nicht dargestellt) und einheitliche Schnittstellen zu den Systemen zur Verfügung (Workflow- und Groupware-Facility). Wie man am Beispiel der Workflow-Facility sehen kann, wird über CORBA eine Schnittstelle zur Benutzung eines existierenden Workflow-Systems realisiert. Dafür muß eine Implementierung der Abbildung vom existierenden System auf die CORBA-Objekte stattfinden. Dieser Ansatz wurde eingeführt, um verschiedene Workflow- bzw.

1. Eine sehr kurze Einführung zu CORBA wurde bereits in Abschnitt 3.2.2 gegeben. [SAYEGH 1997] und [ORFALI ET AL. 1997] bieten sich als Einführung und Übersicht zu CORBA an. Für einen tieferen Einblick in CORBA eignet sich [ORFALI UND HARKEY 1998].

Groupware-Systeme über eine gleichbleibende Schnittstelle einzubinden. In der Designflow-Schicht befinden sich zum einen die ADM-Objekte, die Ressourcen aus den Systemen und Basisdiensten repräsentieren. Diese sind ebenfalls durch eine spezifische Implementierung jeweils als CORBA-Objekt bereitzustellen. Wir wollen dies am Beispiel des Workflow-Systems verdeutlichen. Die Abbildung eines ADM-Objekts stützt sich entweder auf die proprietäre Schnittstelle des WFMS (gestrichelte Pfeile in der Abbildung) oder kann bei geeigneter Implementierung der Workflow-CORBA-API über diese erfolgen (siehe dazu auch Abschnitt 5.5). Entsprechendes gilt für die Groupware-Facility. Die Objekte aus anderen Systemen sind entsprechend über diese Systeme abzubilden (z.B. DB/2). Zur Designflow-Schicht gehören weiterhin das Aktivitäten-Management, die Protokollmaschine und die Informationsraumverwaltung. Diese verwalten einerseits die Aktivitäten und Objekte und andererseits unterstützen sie die verschiedenen Konversationsmuster. Die Informationsraumverwaltung nutzt die ADM-Objekte, das Aktivitäten-Management entsprechend die Aktivitäten. Die Protokollmaschine realisiert die Verbindung der verschiedenen Komponenten und Objekte mittels der Durchführung entsprechender Interaktionen.

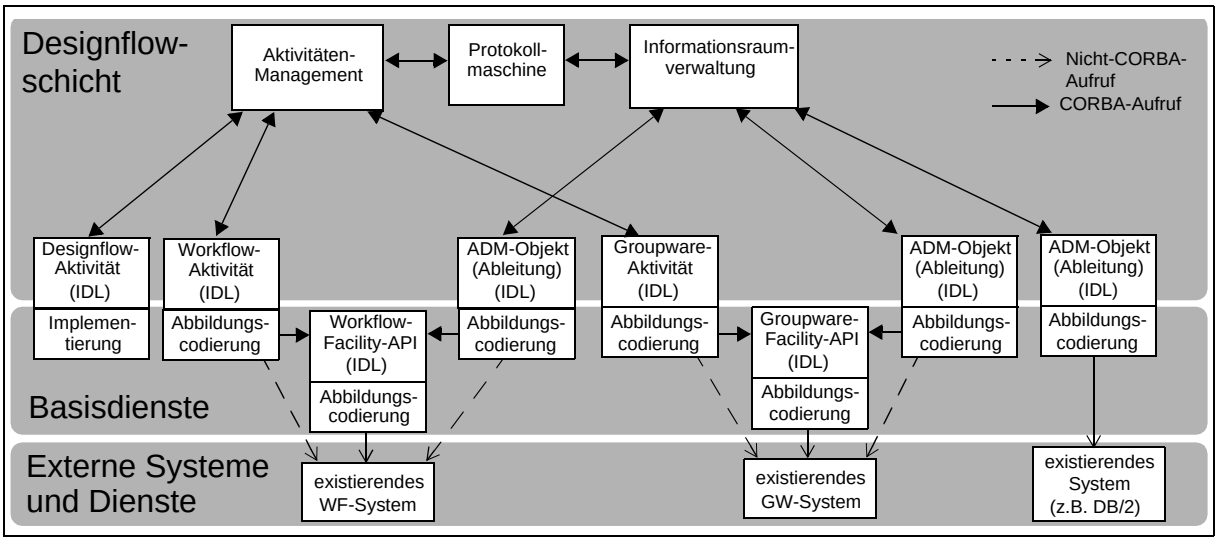


Abbildung 75: CORBA-basierte Integrationsarchitektur von CASSY

5.2.2 Einbindung externer Systeme am Beispiel von Daten

Eines der grundlegenden Probleme des hier vorgestellten Integrationsansatzes ist die geeignete Abbildung externer Systeme auf die CORBA-Objekte, die durch CASSY zu verwalten sind. Wir besprechen dies hier am Beispiel der ADM-Objekte. Entsprechendes gilt auch für die Funktionalität externer Systeme, jedoch zeigt die Einbindung von Daten bereits die prinzipiellen Vorgehensweisen.

Die Werkzeuge und Applikationen, die in Aktivitäten eingebunden sind, arbeiten auf eigenen Daten. Wir wollen mit dem Informationsraum jedoch eine gemeinsame Nutzung über verschiedene Aktivitäten und damit Applikationen hinweg realisieren. Diese Realisierung wird durch die häufig proprietär implementierten und damit nur begrenzt offenen Systemen erschwert. Um diese Problematik zu verdeutlichen, beschreiben wir im folgenden einige Aspekte, wie eine Integration in CASSY stattfinden kann.

Unser Ansatz vermeidet das Replizieren von Entwurfsobjekten. Wir wollen die Daten somit in ihren originären Systemen behandeln. Dazu benötigen wir einen Zugriff auf diese Daten und idealerweise die ’Verwaltungshoheit’ über sie. Dieser Wunsch wird in Abbildung 76 auf der linken Seite dargestellt. Dort werden alle Zugriffe auf die Daten durch CASSY-Objekte verwaltet, egal ob es sich um ein Legacy-System handelt, dessen Daten hier gekapselt werden oder

CASSY-Aktivitäten. Dadurch können Zugriffe verschiedener Aktivitäten allein durch CASSY geregelt werden. Ein solches Einklinken in ein Legacy-System ist leider nur selten durchführbar. Zwei realistischere Szenarien sind ebenfalls in Abbildung 76 dargestellt. Falls eine von außen zugreifbare Datenquelle existiert, kann diese von CASSY genutzt werden. Regelt die Datenquelle selbständig konkurrierende Zugriffe, wie etwa ein DBMS, so werden auch die konkurrierenden Zugriffe von CASSY und dem Legacy-System durch diese Datenquelle geregelt. Jedoch werden dadurch die Zugriffe des Legacy-System nicht unbedingt CASSY bekannt gemacht und somit sind beispielsweise Änderungen schwerer zu propagieren[1]. Der letzte dargestellte Fall zeigt den Zugriff von CASSY auf die Datenquelle über die API des Legacy-Systems. Wie weitgehend dadurch die Integration der Daten möglich ist, hängt von dieser API ab. Dies reicht von Zugriffsverweigerung bis hin zur umfassenden Nutzung. Es ist also jeweils notwendig eine geeignete Umsetzung zu wählen.

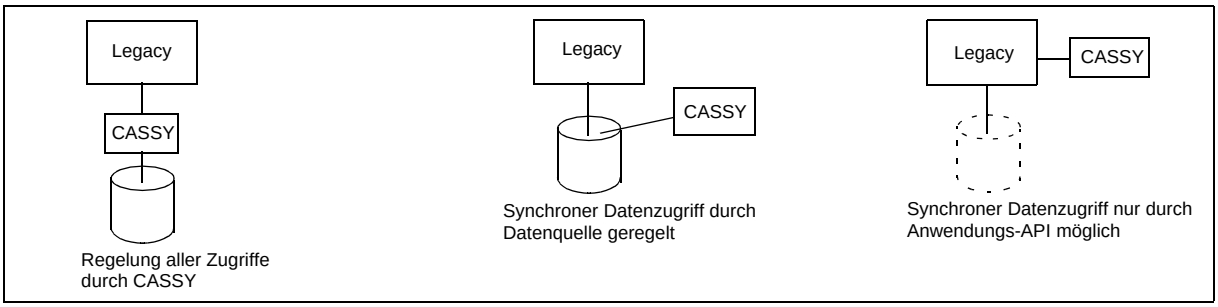


Abbildung 76: Arten des Datenzugriffs bei existierenden Systemen

Prinzipiell gilt, daß offene Schnittstellen von Anwendungssystemen eine Integration der Daten besser unterstützen. Dabei ist es wichtig, wie weitgehend diese Schnittstellen einen Eingriff in die Datennutzung einer Applikation erlauben. Eine komponentenbasierte Applikation hat aufgrund ihrer Architektur häufig viele Möglichkeiten auf die verschiedenen Schichten zuzugreifen, es können einfach Zwischenschichten eingezogen oder sogar ganze Komponenten ausgetauscht werden. Ein monolithisches System hingegen bietet nur begrenzt Schnittstellen an, von denen es abhängig ist, wie tiefgreifend ein Eingriff möglich ist.

Wie die geschilderten Integrationsmöglichkeiten in der gewünschten Architektur umgesetzt werden, besprechen wir in den nachfolgenden Abschnitten. Dort wird auch gezeigt wie die Funktionalität externer Systeme in CASSY eingebunden wird.

5.3 Die CORBA-Schnittstelle von CASSY

Wir werden im folgenden auf die wesentlichen Aspekte und Interfaces der CORBA-Schnittstelle von CASSY eingehen, welche die zuvor beschriebene Designflow-Schicht darstellt. Zuerst geben wir einen Überblick über das gesamte Objektmodell und in welchen IDL-Dateien die Interfaces zu finden sind. Danach besprechen wir die zentralen Klassen des Objektmodells im einzelnen.

Das Objektmodell kann man in die vier Bereiche Organisationsmodell, Informationsraum, Aktivitätenmodell und Interaktionsmodell aufteilen, wie in Abbildung 77 durch die schraffierten Blöcke illustriert. Das Organisationsmodell basiert im wesentlichen auf dem Modell, welches für die Groupware-Facility umgesetzt wurde (siehe Abschnitt 5.6). Es basiert auf Akteuren (Actor), denen Rollen (Role) zugewiesen werden, die durch Eigenschaften (Property) definiert sind. Akteure können in Gruppen (Group) zusammengefaßt werden. Benutzer (User) und Ressourcen

1. Im Falle eines DBMS können Trigger dazu genutzt werden, einen Zugriff des Legacy-Systems einer CASSY-Instanz mitzuteilen.

(Resource-Adapter), wie Programme oder Maschinen, können Aktivitäten durchführen und sind daher Spezialisierungen des Akteurs. Der Informationsraum besteht aus den Objekten (Resource-Adapter) und den Objektabhängigkeiten (Resource-Dependency). Das Aktivitätenmodell enthält die verschiedenen Aktivitätsarten, lediglich die primitive Aktivität wurde nicht aufgeführt, da sie i.w. mit der einfachen Aktivität (Activity) übereinstimmt. Das Interaktionsmodell ist als solches nicht in der CORBA-Schnittstelle enthalten, sondern als Komponente in CASSY, die in den Objekten und Aktivitäten eingesetzt wird. Sie wurde dennoch zur Verdeutlichung dargestellt.

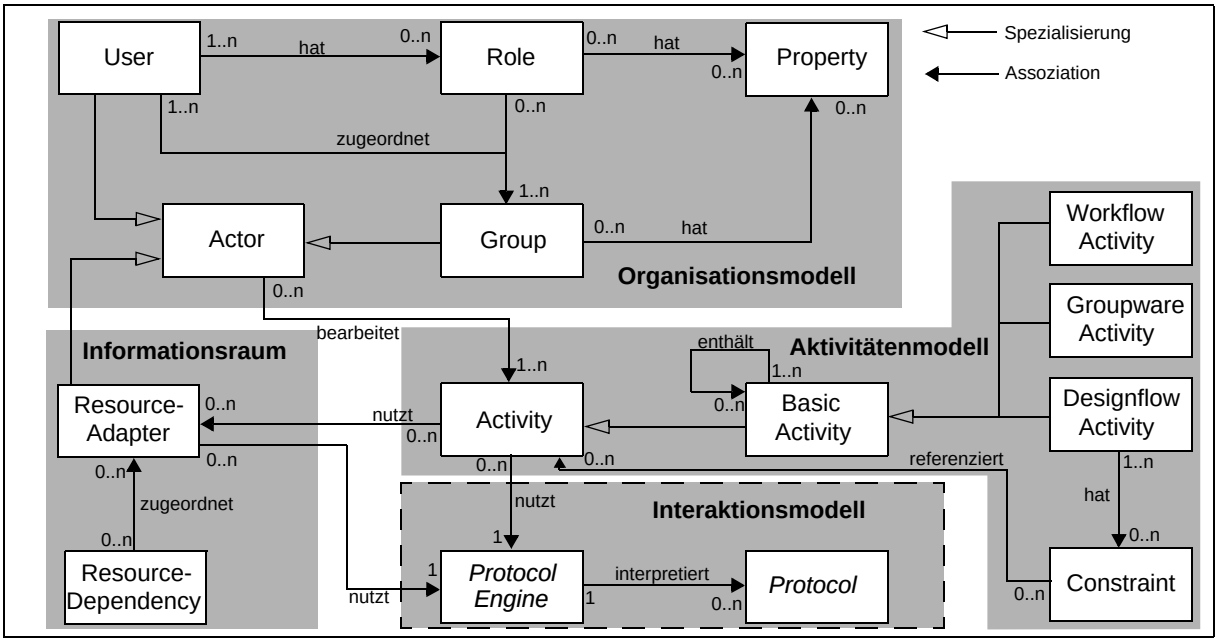


Abbildung 77: CASSY-Objektmodell

Tabelle 8 zeigt eine Übersicht der von CASSY verwendeten IDL-Dateien, die auch im Anhang zu finden sind. Die Workflow- und Groupware-Facility wurden mitaufgeführt (siehe Abschnitt 5.5 bzw. Abschnitt 5.6), da sie die zu integrierende Technologie liefern. `ActorInterface.idl` und `ActivityInterface.idl` stellen die Schnittstellen für die wesentlichen Bestandteile des ADM zur Verfügung und sind somit die Wrapper für die zu integrierenden Systeme (vgl. Abschnitt 5.3). Die Event Engine ist eine eigenständige Implementierung, welche die Constraints-Behandlung realisiert (siehe Abschnitt 5.4). Für die Protokollmaschine existiert keine Schnittstellenspezifikation, außer den Aufrufen in den betroffenen Elementen (Aktivitäten und Objekte), in die sie integriert wird (siehe Abschnitt 5.7). Die Abbildung der Systembestandteile auf diese Schnittstelle wird in den entsprechenden Abschnitten diskutiert.

IDL-Datei	Beschreibung
BoWorkflow.idl	Workflow-Facility
CSCW.idl	Grundlegende Definitionen der CSCW-Facility
ActorInterface.idl	Informationsraum und Organisationsmodell
ActivityInterface.idl	Aktivitätenmodell
EventEngine.idl	Ereignis- und Constraint-Behandlung

Tabelle 8: IDL-Dateien von CASSY

5.3.1 Schnittstellen für Organisationsstruktur und Ressourcen

Wie in Abschnitt 3.4.7 beschrieben, kann man CSCW-Funktionalität als eine Obermenge von Workflow-Funktionalität bezeichnen. Entsprechend wurde die CSCW-Facility entworfen und bietet daher die Grundlage für die meisten CASSY-Elemente. Dies zeigt sich besonders deutlich in der Datei `ActorInterface.idl`, die auf dem Organisations- und Ressourcenkonzept der CSCW-Facility basiert und dieses auch weitgehend übernimmt. Es wurden folgende Interfaces und Strukturen definiert[1]:

Actor

Ein `Actor` ist ein Objekt, das eine Aktivität ausführen darf. Er besitzt einen Namen, einen Identifikator und die Möglichkeit zur Abfrage des Typs. Die Abfrage ist wichtig, da `Group` und `ResourceAdapter` Spezialisierungen von Actor sind. Diese Entwurfsentscheidung wurde getroffen, da auch Maschinen oder Gruppen Aktivitäten ausführen dürfen. Ein Akteur besitzt eine Worklist mit den zugeteilten Aktivitäten und entsprechende Operationen, um diese zu manipulieren und abzufragen.

User

Ein `User`-Objekt ist eine Spezialisierung von `Actor` und repräsentiert eine reale Person mit ihren Rollen, Gruppenzugehörigkeit und zugewiesenen Aktivitäten (Abbildung 78, zeigt auch die Operationen des `Actor`-Interfaces). Für einen User wird unterschieden, ob er im System angemeldet ist oder nicht (`Connected`/`Disconnected`). Da ein Benutzer auch mit Entitäten des ADM interagieren darf, sind die entsprechenden Methoden zur Interaktion deklariert (`SendMessage()`, `ReceiveMessage()`). Die Rollen eines Benutzers und dessen Gruppenzugehörigkeiten lassen sich durch die entsprechenden Operation verwalten.

Group

Eine `Group` ist ein `Actor` und erlaubt es verschiedene `User`-Objekte in einer Gruppe zu verwalten (hinzufügen, entfernen, auflisten).

BasicResourceAdapter

Ein `BasicResourceAdapter` (Abbildung 79) wird genutzt um Ressourcen zu kapseln, somit auch Datenobjekte, die in den Informationsraum eingefügt werden können. Es werden damit auch Programme oder Maschinen gekapselt, die als `Actor` einer Aktivität auftreten können. Der Zugriff auf Daten wird mit einer Sende- und einer Empfangs-Operation durchgeführt, welche die Kommunikation mittels Interaktionsprotokollen unterstützen. Die Kapselung kann entweder durch die Referenzierung eines weiteren CORBA-Objekts geschehen, das durch `getImplementationObject()` abgefragt werden kann oder durch die Spezialisierung der Klasse und das Erweitern um objektspezifische Operationen. Der erste Weg dient vor allem der schnellen Integration, angestrebt wird hingegen die Verwendung der Spezialisierung, um die spezifische Funktionalität von Ressourcen abbilden zu können. Für eine Datebanktabelle könnten dann Operation wie die Ausführung von SQL-Anweisungen eingeführt werden.

ResourceDependency

Das Interface `ResourceDependency` (Abbildung 80) dient dazu die in Abschnitt 4.6.3 vorgestellten Objektabhängigkeiten zu erreichen. Hier können `BasicResourceAdapter`-Objekte an- und abgemeldet werden. Es besteht die Möglichkeit aus verschiedenen Notifikationsarten

1. Aus Platzgründen zeigen wir nur ausgewählten Quelltext. Die komplette Datei ist im Anhang zu finden.

```
interface User : Actor {
//Methoden aus der Klasse Actor:
   string getName();
   boolean setName(in string name);
   string getID();
   ActorType isa();
   boolean load();
   boolean save();
   boolean AssignActivity(in Activity act, in unsigned long mode);
   boolean RemoveActivity(in Activity act);
   boolean ReadAssignedActivities(out ActvityList acts);
//Methoden aus der Klasse Actor (ENDE)
   // Anmeldung am System
   enum ConnectionState {Connected,Disconnected};
   boolean getConnectionState(out ConnectionState connState);
   boolean ChangePassword(in string OldPwd, in string NewPwd);
   boolean CheckPassword(in string Pwd);
   string Connect() ;
   void Disconnect();
   boolean SaveUserData();
   // Protokollinteraktion
   boolean SendMessage(in Message msg);
   boolean ReceiveMessage(out Message msg);
   //Rollenverwaltung
   boolean AddRole(in Role r, in string groupid, in unsigned long mode);
   boolean RemoveRole(in string rolename, in string groupid, in unsigned long mode);
   boolean SaveRoles(in string groupid);
   boolean ReadRoles(in string groupid,out RoleNameList rnl);
   boolean ListRoles(in string groupid, out RoleList rl);
   //Gruppenverwaltung
   boolean AddGroup(in Group g, in unsigned long mode);
   boolean RemoveGroup(in string groupid, in unsigned long mode);
   boolean RemoveAllGroups();
   boolean SaveGroups();
   boolean ReadGroups(out GroupIDList gidl);
   boolean ListGroups(out GroupList gl);
};
```

Abbildung 78: User Interface (IDL, erweitert um Actor-Methoden)

```
interface BasicResourceAdapter : Actor {
   enum ResourceType { FILE, PERSISTENT_OBJECT, GW_RESOURCE, WF_RESOURCE, SPECIFIC };

   // liefert “original” Objekt einer Facility oder eines externen Systems zurück
   Object getImplementationObject();
   ResourceType GetResourceType();
   ResourceAdapterType GetResourceAdapterType();
   string isA();
   boolean AssignResourceObject(in Object obj);
   Object GetRessourceObject();

   boolean SaveAdapterInformation(in ResourceAdapterType adapter_type);
   boolean AssignExclusively(in Activity act);
   boolean FreeFromActivity();

   //Informationsraum
   boolean RegisterInInfoSpace();
   boolean RemoveFromInfoSpace();

   //Interaktion
   boolean receiveMessage( in any msg );
   boolean sendMessage( in CORBA::Object receiver, any msg );
   //...(gekürzt)
};
```

Abbildung 79: BasicResourceAdapter Interface (IDL)

(`NotificationType`) zu wählen. Abgeleitete Interfaces sind `PartOfDependency` und `IsADependency`, die ein Super-Objekt bzw. eine Basisobjekt für die Abhängigkeit einführen (siehe Anhang).

```
enum NotificationType{NONE,      //keine Notifikation
                      ALL,       //Weiterleitung jeder Änderung (mit Objektreferenz)
                      SPECIAL,   //für zusätzliche Modi
                      PASSIVE};  //Nur Mitteilung, daß eine Änderung stattgefunden hat

interface ResourceDependency {
   readonly attribute   ResourceList Objects;
   attribute            NotificationType notification;
   readonly attribute   string description;

   boolean register( in BasicResourceAdapter object);
   boolean unregister( in BasicResourceAdapter object);
   boolean receiveMessage( in any msg );
   boolean sendMessage( in CORBA::Object receiver, any msg );
};
```

Abbildung 80: ResourceDependency Interface (IDL)[a]

a. Das gezeigte Interface nutzt das IDL-Sprachelement `attribute`. Dies wird durch den IDL-Compiler in Zugriffsfunktionen umgesetzt, um das Attribut lesen und schreiben zu können. Daher ist es genauso möglich lediglich die Zugriffsfunktionen zu deklarieren, wie in den vorherigen Interfaces.

Role, Property

Rollen beschreiben die `User`-Objekte, d.h. deren organisatorische Positionen und Fähigkeiten. Diese Eigenschaften können durch die Properties beschrieben werden. So kann sich ein `Role`-Objekt auf eine Menge von `Property`-Objekten beziehen. Einem User können Rollen zugewiesen sein. Somit entsprechen diese dem Objektmodell der Groupware-Facility aus Abschnitt 5.6.1.

5.3.2 Die Aktivitätenschnittstellen

Die Aktivitätenschnittstellen werden in der Datei `ActivityInterface.idl` (Anhang E) beschrieben. Es existieren die im ADM vorhanden Aktivitätstypen.

Activity

Das `Activity`-Interface dient als Oberklasse für alle Aktivitäten und beinhaltet alle übergreifenden Funktionen für Aktivitäten (Abbildung 81). Diese sind die Zustandsänderungen, Verwaltung der Akteure, Name und ID einer Aktivität, Zugriff auf eine Elternaktivität, ausführende Applikation oder Facility, Verwaltung der nutzbaren Ressourcen, Persistenz und die Interaktionsschnittstelle. Die Ressourcen umfassen auch die Entwurfsobjekte, die in dem Informationsraum eingetragen sind. Letztendlich entspricht die Activity der primitiven Aktivität.

BasicActivity

Die `BasicActivity` erweitert das `Activity`-Interface um Funktionen, die eine Schachtelung von Aktivitäten erlauben, also die Verwaltung von Kindaktivitäten (abfragen, hinzufügen, löschen, usw.).

WorkflowActivity

Workflow-Aktivitäten des ADM unterscheiden sich von der Basis-Aktivität lediglich durch die Verwaltung der zugrunde liegenden Prozeßdefinition.

GroupwareActivity

Da eine Groupware-Aktivität eine möglicherweise komplexe aber nicht strukturierte Anwendung von Groupware-Funktionen realisiert, ist es nicht sinnvoll weitere Operationen in das Interface aufzunehmen. Wir werden bezüglich der Groupware-Facility aber beschreiben, wie mächtig eine Groupware-Anwendung modelliert werden kann.

```
interface Activity {
   attribute unsigned long resources_length;
   ActivityType is_a();
   //state management
   boolean getstate(out stateType state);
   boolean init();
   boolean start();
   boolean stop();
   boolean terminate();
   boolean reset();
   boolean cancel();
   boolean resume();
   //actor management
   boolean getActor(out ActorInterface::Actor actor);
   boolean setActor(in ActorInterface::Actor actor);
   //descriptive activity name
   boolean getName(out string name);
   boolean setName(in string name);
   //ID
   boolean getID(out string id);
   boolean setID(in string name);
   //hierarchy (parents only)
   boolean getParentActivity(out BasicActivity activity );
   boolean setParentActivity(in BasicActivity activity);
   //execution
   boolean setProgram(in Program program, in string description)
   boolean getProgram(out Program program, out string description)
   //resource management
   boolean getResources( out ResourceList resources);
   boolean setResourceList(in ResourceList resources, in long len);
   boolean getResource(in string id, out ActorInterface::BasicResourceAdapter bra);
   boolean addResource(in ActorInterface::BasicResourceAdapter bra);
   //persistency
   boolean saveActivity();
   boolean loadActivity(in ToplevelActivity top);
   //interaction
   boolean receiveMessage( in any msg );
   boolean sendMessage( in CORBA::Object receiver, in any msg );
};
```

Abbildung 81: Activity Interface (IDL)

DesignflowActivity

Für die Einbringung von Constraints ist es notwendig diese in der entsprechenden Designflow-Aktivität (siehe auch Abschnitt 5.4), wie auch die weiteren Design-Primitive (uses, delegates), dort zu verwalten. Ansonsten unterscheidet sich das Interface (Abbildung 82) nicht von der `BasicActivity`.

```
Interface DesignflowActivity : BasicActivity {
   attribute ConstraintList constraints;
   attribute DesignPrimitiveList designPrimitives;
   Constraint createConstraint(in string name,in ObjectSeq designItems);
   void removeConstraint(in Constraint c);
};
```

Abbildung 82: DesignflowActivity Interface (IDL)

Die geringen Unterschiede zwischen den verschiedenen Aktivitätstypen werfen die Frage auf, warum die verschiedenen Interfaces notwendig sind. Dies ist eine Entscheidung, um die typische Behandlung der verschiedenen Aktivitäten durch unterschiedliche Technologien realisieren zu können, wie in den folgenden Abschnitten zu sehen sein wird. Jedoch wird die Behandlung durch das Aktivitäten-Management stark vereinfacht, da hier nur auf wenige Spezifika der Aktivitätstypen geachtet werden muß. Somit kann die Polymorphie der Ableitungshierarchie hier gut ausgenutzt werden.

5.4 Die Event Engine

Um die Ausführung der zuvor beschriebenen Aktivitäten zu regeln, wurde in Abschnitt 4.4.4 die Funktionsweise der Event Engine auf Basis von definierbaren Constraints vorgestellt. Die vorliegende Implementierung erfolgte mittels Java und ORBacus als CORBA-System. Basis für die Behandlung von Constraints sind die definierbaren Constraints-Ausdrücke (`ConstraintDefinition`[1], siehe Abbildung 83), die eine Folge von verknüpften Ereignissen beschreiben. Gültige Operatoren und Elemente sind: Und, Oder, Sequenz, Komplement, Konstanten, Ereignisse und weitere Constraints-Definitionen. Auf Basis dieser Definitionen werden in Designflow-Aktivitäten Constraints (`Constraint`-Interface) zwischen Aktivitäten und/oder Ressourcen spezifiziert. Ein Constraint bezieht sich dabei auf eine Constraint-Definition und liefert die notwendigen Parameter zur Instantiierung eines Constraints.

```
interface Constraint {
   // Die DesignFlowActivity ist die Factory für Constraints
   readonly attribute string name;
   readonly attribute ObjectSeq parameters;
};

//Ein Ereignis besitzt einen Namen und eine Folge von Parametern
struct Event { string name; NameValueSequence parameters;};
typedef sequence<Event> EventList;

enum ConstraintOperator { AND_DEP, OR_DEP, SEQ_DEP, EVENT, COMPLEMENT, CONSTANT };
union ConstraintDefinition switch (ConstraintOperator) {
    case AND_DEP:
    case OR_DEP:
    case SEQ_DEP: sequence<ConstraintDefinition> operands;
    case EVENT:
    case COMPLEMENT: Event event;
    case CONSTANT: boolean value;
};
```

Abbildung 83: Constraint Interface und benötigte Strukturen (IDL)

Die in XML codierten Constraint-Definitionen werden zum Start der Event Engine eingelesen und befinden sich in der Datei `constraints.xml` (siehe Anhang D). Wie in Abbildung 84 zu erkennen ist, können die Benennungen direkt auf die Definitionen der IDL-Schnittstelle abgebildet werden. Das vorliegende WHILE wird durch folgenden Ausdruck beschrieben: *(s($1) - s($0) - t($0) - t($1)) OR (NOT(s($0))*. Die mit '$' bezeichneten Zahlen sind die Parameter, die bei einer Constraint-Instanz, Aktivitäten, Ressourcen oder freie Variablen repräsentieren. Der gesamte Ausdruck besagt also, daß entweder die Ausführung der Aktivität $0 während der Ausführung der Aktivität $1 stattzufinden hat, ansonsten darf Aktivität $0 nicht ausgeführt werden.

Durch die Definition der Constraints in einer XML-Datei kann CASSY an verschiedene Ausführungssemantiken angepaßt werden, da die Event Engine diese Definitionen interpretiert und gegen die eintreffenden Ereignisse abgleicht. Die an der Ereignisabarbeitung beteiligten Komponenten sind in Abbildung 85 dargestellt. Im einzelnen sind dies:

- **Definitions-Werkzeuge**
  Mittels des Designflow-Editors (Abschnitt 5.8.2) und den XML-basierten Constraints-Definitionen können die in einem Designflow verwendeten Aktivitäten und deren Constraints verändert bzw. die Constraint-Definitionen modifiziert werden.
- **Actor Worklist**
  Die Worklist eines Akteurs beinhaltet die vom Benutzer zu bearbeitenden Aktivitäten.

1. Tatsächlich heißen die Constraint-Definitionen in der vorliegenden Implementierung Dependencies (siehe Anhang). Wir haben diese hier umbenannt um Verwechslungen mit den Objekt-Abhängigkeiten auszuschließen.

```
<CONSTRAINTDEFINITION NAME="while" VERSION="0.1">
<OR>
  <SEQUENCE>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
    <EVENT NAME="terminate">
        <PARAMETER NAME="id" VALUE="$1.id"/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME="start"><PARAMETER NAME="id" VALUE="$0.id"/></EVENT>
  </COMPLEMENT>
</OR>
</CONSTRAINTDEFINITION>
```

Abbildung 84: Definition des WHILE-Constraints (XML)

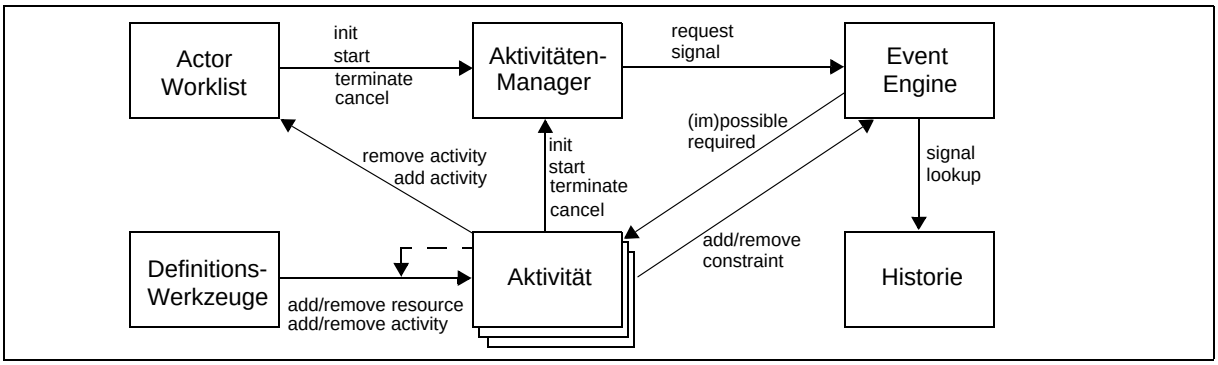


Abbildung 85: Architektur der Event Engine

Diese können bereits gestartet sein oder zum Start anstehen. Soll der Zustand einer Aktivität geändert werden (z.B. Starten) wird der Aktivitäten-Manager mit einer entsprechenden Nachfrage aufgerufen, der dann die entsprechende Durchführbarkeit zurückmeldet. Aufgrund des Zustands der Event Engine kann der Fall eintreten, daß eine zum Start bereitstehende Aktivität derzeit nicht ausgeführt werden darf, da sie durch veränderte Ereignisse den entsprechenden Constraints ansonsten widerspricht. Dennoch wird diese Aktivität nicht aus der Liste entfernt, da dieses Wissen einem Benutzer hilft die aktuellen Abläufe eines Entwurfsprozesses besser zu verfolgen. In diesem Falle erfährt ein Akteur durch welche Constraints die eigentlich anstehende Aktivität blockiert wird.

- **Aktivitäten**
  Eine Designflow-Aktivität beinhaltet die Constraints bezüglich der enthaltenen Sub-Aktivitäten (s.o.). Diese werden zum Start der Aktivität in die Event Engine eingelesen. Außerdem werden von allen Aktivitäten Zustandsänderungen an die Event Engine propagiert. Schließlich wird eine Designflow-Aktivität informiert, wenn triggerbare Ereignisse ausführbar werden oder wenn notwendige Ereignisse nicht mehr ausführbar sind, wie in Abschnitt 4.4.4.3 beschrieben (impossible, possible, required). Änderungen von Constraints werden an die Event Engine weitergeleitet.
- **Aktivitäten-Manager**
  Der Aktivitäten-Manager prüft Anfragen von Benutzern und Aktivitäten bezüglich des Eintretens von Ereignissen. Ist das Ereignis erlaubt, ruft der Aktivitäten-Manager die entsprechende Operation der betroffenen Aktivität auf. Die IDL-Schnittstelle (siehe Anhang) des `ActivityManager`-Interface besteht aus den Operationen: `init()`, `start()`, `stop()`, `terminate()`, `reset()`, `cancel()` und `resume()`.
- **Event Engine**
  Die Event Engine erhält zum einen die Anfragen des Aktivitäten-Managers (`requestEvent()`, Abbildung 86) und zum anderen arbeitet sie eintreffende Ereignisse ab (`signa-`

lEvent()). Sie stellt sicher, daß dem Aktivitäten-Manager zugesagte Ereignisse auch tatsächlich eintreten dürfen. Beim Start einer Designflow-Aktivität werden deren Constraints der Event Engine übergeben und in die Regelbasis eingebracht, die der Abarbeitung von Ereignissen dienen. Die Abarbeitung erfolgt entsprechend Abschnitt 4.4.4. Die Event Engine erlaubt die Abfrage der vorhanden Constraints-Instanzen, Teil-Sequenzen und -Definitionen bzw. deren Löschung.

```
interface Engine {
   //Erfragt den Zustand (EventState) des Ereignisses.
   EventState queryEvent(in Event e, out Dependency blockingDependency);
   //Fordert ein Ereignis an und blockiert es damit für andere.
   boolean requestEvent(in Event e, out Dependency blockingDependency);
   boolean cancelRequestEvent(in Event e);
   //Das Ereignis gilt mit dem Aufruf dieser Methode als eingetreten, auch wenn es nicht hätte
   //eintreten sollen. Die Exception wird erst nach dem Einarbeiten des Events geschmissen.
   void signalEvent(in Event e) raises (DependencyViolation);
   //Die Methode nimmt die ConstraintDefinition in die Menge der zu beachtenden Regeln auf.
   void createConstrainDefinition(in ConstraintDefinition definition);
   //Entfernen von Constraints
   boolean removeConstraintDefinition(in string name);
   //Gibt eine Liste aller aktuell registrierten Constraints zurück.
   ConstraintList constraintDefinitions();
   //Die Methode liefert den Rest einer Constraint-Instanz zurück. Dieser Rest ist aus der
   //Definition des Constraints im Laufe der eingetroffenen Events entstanden und gibt den
   //noch zu betrachtenden Teil an.
   ConstraintDefinition residual(in string name);
};
```

Abbildung 86: Event Engine Interface (IDL)

- **Historie**
  In der Historie werden die eingetretenen Ereignisse gespeichert. Damit dient sie zum einem dem Recovery und zum anderen der Nachverfolgung eines Designflows.

Die vorliegende Implementierung der Abarbeitung kann in Widersprüche geraten, wenn eine 'ungünstig gewählte' Menge von Regeln in der Regelbasis auftritt [WAGNER 2001]. Dies vorab zu erkennen und zu vermeiden ist ein NP-vollständiges Problem. In der Literatur wird dieses Problem entweder gar nicht erkannt [SINGH 1997A], oder es wird "vermutet", daß bei Workflows diese Probleme in polynomialer Zeit lösbar sind [ATTIE ET AL. 1996]. Da eine Auflösung zur Laufzeit möglich ist, bedeutet dies für den Prototypen keine wesentliche Einschränkung, weswegen auf eine genauere Untersuchung verzichtet wurde.

5.5 Die Workflow-Management-Facility

Um die Durchführung von Workflow-Aktivitäten des ADM zu erreichen wurde nach einer entsprechenden Schnittstelle gesucht, welche die gewünschte Funktionalität über CORBA anbieten kann und auch auf die gängigen Workflow-Systeme abbildbar ist. Wir werden am Beispiel der entstandenen Workflow-Facility detaillierter auf die Vorgehensweise zur Einbindung eines existierenden Systems eingehen, um die Integrationsweise und auftretende Hindernisse zu verdeutlichen.

Für die Realisierung der Workflow-Dienste bestand die Möglichkeit diese selbst zu entwickeln, proprietäre (CORBA-)Herstellerlösungen zu verwenden (z.B. ARIS-Workflow), die IDL-Bindings der WfMC [WFMC96] oder die geplante OMG *Workflow Management Facility* (s. u.) zu übernehmen. Eine Eigenentwicklung wurde abgelehnt, da die Nutzung eines 'Standards' verspricht, daß Hersteller bereits oder in naher Zukunft eigene *Mappings* liefern. Eine proprietäre Lösung hätte den Einsatz eines bestimmten Produktes vorausgesetzt, was aber nicht für einen allgemeinen Einsatz sinnvoll ist. Die IDL-Bindings der WfMC wurden nicht gewählt, da sie zum einen fehlerhaft sind und zum anderen als "einfache" Schnittstelle zu einem WFMS nicht dem Service-Konzept von CORBA entsprechen [HEIß 1998].

Gleichzeitig mit unseren Untersuchungen wurde von der OMG ein RFP (*request for proposal*) für eine OMG Workflow Facility ausgeschrieben. Die WfMC reichte über einen Verbund von Industrieunternehmen (z.B., CoCreate Software, IBM, IABG) einen entsprechenden Vorschlag ein [COCREATE SOFTWARE ET AL. 1997]. Diesen Vorschlag nennen wir im folgenden kurz JFlow (*joint workflow proposal*). Da eine OMG Workflow Management Facility eine zukunftssichere und OMG-konforme Lösung darstellt, wurde JFlow zur Implementierung der Workflow-Dienste herangezogen.

Für die Implementierung des JFlow-Proposals wurde das Workflow-System FlowMark der IBM verwendet. Die Realisierung der CORBA-Objekte erfolgte durch ein *Wrapping*. FlowMark bietet eine C++-API an, die auf die entsprechenden CORBA-Objekte des jFlow-Proposals abgebildet wurde. Dadurch entsteht eine mehrschichtige Architektur, wie in Abschnitt 5.5.3 beschrieben. Die JFlow-Implementierung stellt die Workflow Management Facility dar und realisiert in CASSY die Workflow-Aktivitäten.

5.5.1 Das JFlow-Modell

In diesem Abschnitt vertiefen wir das JFlow-Objektmodell (Abbildung 87). Dort stehen Objekte für die dynamischen und statischen Aspekte eines Workflows zur Verfügung. Der Einstiegspunkt ist der `WorkflowManager`, über den die einzelnen Objekte erfragt werden können. Die statischen Aspekte umfassen die Verwaltung der Historie (`HistoryIterator`), Elemente (`ElementIterator`), und Prozeßdefinitionen (`ProcessDefintion`, `DefinitionIterator`). Zur Suche von Objekten werden sogenannte Filter verwendet, die durch eine Zeichenkette repräsentiert werden. Die Implementierung der Filter ist offen gelassen, es bietet sich jedoch die Verwendung regulärer Ausdrücke an. Bei der Durchführung eines Prozesses werden den Aktivitäten und Prozeßinstanzen (Spezialisierungen von `ExecutionElement`) Daten (`ProcessData`) und Akteure (`Participant`) zugeordnet.

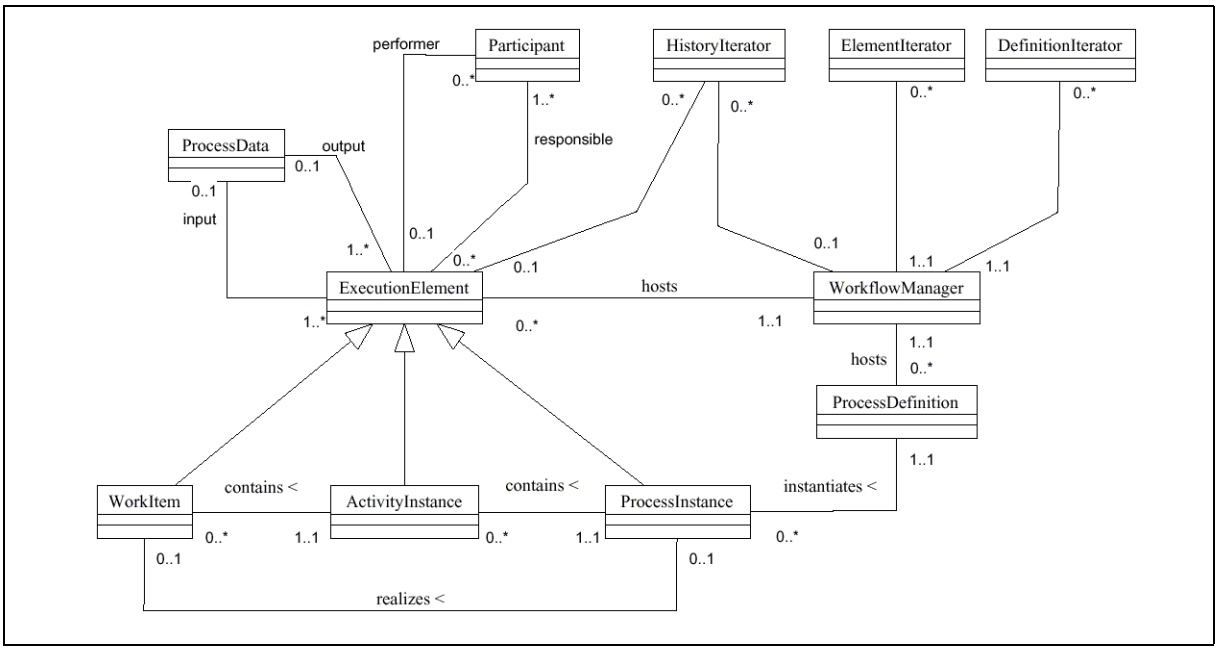


Abbildung 87: JFlow-Objektmodell (UML)

Das `WorkflowManager`-Interface erfüllt folgende Funktionalitäten als zentrale Instanz des Objektmodells (Abbildung 88):

- Es erlaubt den Zugriff auf die im WFMS vorhandenen *Prozeßdefinitionen*, in dem es Referenzen auf `ProcessDefintion`-Objekte liefert. Diese Referenzen können entweder

anhand eines eindeutigen Schlüssels direkt vom `WorkflowManager`-Objekt oder durch die Angabe eines Filters über das `DefinitionIterator`-Interface bezogen werden.

- Ebenso können *Prozeßinstanzen, Aktivitäten* und *Work Items* als Spezialisierung des `ExecutionElement`-Interfaces über die Angabe von Schlüsseln oder mittels Filtern für ein `ElementIterator`-Objekt zurückgeliefert werden.
- Die Workflow-Historie kann über einen Filter und das `HistoryIterator`-Interface abgefragt werden.

Die jeweiligen Iteratoren werden durch die entsprechenden Aufrufe im Sinne einer CORBA-Factory erzeugt.

```
interface WorkflowManager {
   readonly attribute WfMKey key;
   ElementIterator list_elements(in element_type type, in Filter filter)
      raises (InvalidFilter);
   ExecutionElement get_element(in element_type type, in WfKey key) raises (InvalidKey);
   HistoryIterator list_history(in element_type type, in Filter filter) raises (InvalidFilter);
   DefinitionIterator list_definitions(in Filter filter) raises (InvalidFilter);
   ProcessDefinition get_definition(in WfKey key) raises (InvalidKey);
};
```

Abbildung 88: Workflow Manager (IDL)

Das `ExecutionElement`-Interface ist die Basis für die Instanzen eines Prozesses und somit die Basisklasse für Aktivitäten (`ActivityInstance`), Work Items (`WorkItem`) und Prozeßinstanzen (`ProcessInstance`). Die IDL des `ExecutionElement`-Interfaces ist in Abbildung 89 dargestellt. Ein `ExecutionElement` wird durch einen eindeutigen Namen (`name`) in lesbarer Form identifiziert und kann eine Priorität (`priority`) besitzen. Eine textuelle Beschreibung der Instanz kann in `description` gespeichert werden. Die Ein- und Ausgabedaten werden in `input` und `ouput` beschrieben. Die Funktion `destroy()` erlaubt eine Löschung des Elements und im Falle einer Prozeßinstanz oder einer geschachtelten Aktivität werden die enthaltenen Elemente auch gelöscht bzw. eine Ausnahme (`TransitionNotAllowed`) ausgelöst. Die Hauptfunktionen umfassen die Abfrage der Zustände und Übergänge eines Elements und deren Ausführung. Für `ExecutionElement` und deren abgeleiteten Interfaces wurden entsprechende Zustandsübergänge definiert, die in [CoCreate Software et al. 1997] zu finden sind. Diese sind auch mit dem im ADM vorgeschlagenen Modell vereinbar. Letztlich können die Akteure ermittelt werden und die Historie kann abgefragt werden.

Eine Prozeßinstanz ist eine Spezialisierung des `ExecutionElements` und repräsentiert eine Workflow-Instanz, die aus einem `ProcessDefintion`-Objekt erzeugt wird (Abbildung 90, Attribut `implements`). Daher kann sie auch mehrere `ActivityInstance`-Objekte enthalten, die über `list_activities()` abgefragt werden können. Es kann eine umfassende Historie abgerufen werden (`list_full_history()`), und es stehen Operationen zum Starten, Beenden und Abbrechen einer Prozeßinstanz zur Verfügung (`start()`, `terminate()`, `abort()`). Das `ActivityInstance`-Interface fügt dem ExecutionElement lediglich einen Verweis auf deren übergeordnete Prozeßinstanz (`contained_in`) und eine Abfrage der assoziierten Work Item hinzu (`list_work_items()`).

Ein `WorkItem` (Abbildung 91) stellt die Verbindung eines `ActivityInstance`-Objekts mit genau einem `Participant`-Objekt (Akteur) dar. Eine `ActivityInstance` darf mehrere `WorkItem`-Objekte umfassen.

```
interface ExecutionElement {
   attribute WfName name;
   attribute unsigned long priority;
   readonly attribute WfKey key;
   attribute string description;
   readonly attribute ProcessData input;
   readonly attribute ProcessData output;

   void destroy() raises(TransitionNotAllowed);
   StateList list_valid_states();
   TransitionList list_valid_transitions();
   State get_state();
   void change_state(in State new_state)
      raises(InvalidState, StateChangeNotAllowed, StateChangeNotPerformed);
   State make_transition (in WfName transition_name)
      raises (InvalidTransition, TransitionNotAllowed, TransitionNotPerformed);
   HistoryIterator get_history(in Filter filter)
      raises(HistoryNotAvailable);
   ParticipantList get_responsibles();
   void set_responsibles(in ParticipantList responsibles)
      raises(UpdateParticipantNotAllowed, ParticipantAssignmentFailed);
   ParticipantList get_performers();
   void set_performers(in ParticipantList performers)
      raises(UpdateParticipantNotAllowed, ParticipantAssignmentFailed);
};
```

Abbildung 89: ExecutionElement (IDL)

```
interface ProcessInstance : ExecutionElement {
   readonly attribute ProcessDefinition implements;
   readonly attribute WfKey realizes_work_item;
   readonly attribute WfKey realizes_activity_instance;
   ActivityInstanceList list_activities(in Filter filter);
   HistoryIterator list_full_history(in Filter filter)
      raises(HistoryNotAvailable, InvalidFilter);
   void start () raises(TransitionNotAllowed, TransitionNotPerformed);
   void terminate() raises(TransitionNotAllowed, TransitionNotPerformed);
   void abort() raises(TransitionNotAllowed, TransitionNotPerformed);
};

interface ActivityInstance : ExecutionElement {
   readonly attribute ProcessInstance contained_in;
   WorkItemList list_work_items(in Filter filter) raises(InvalidFilter);
};
```

Abbildung 90: ProcessInstance und ActivityInstance (IDL)

```
interface WorkItem : ExecutionElement {
   readonly attribute WfKey realized_by;
   readonly attribute ActivityInstance contained_in;
   void reassign (in Participant new_owner) raises (ParticipantAssignmentFailed);
   string start() raises(TransitionNotAllowed, TransitionNotPerformed);
   void complete() raises(TransitionNotAllowed, TransitionNotPerformed);
};
```

Abbildung 91: WorkItem (IDL)

Für die Realisierung des Informationsraums im ADM ist der Zugriff auf die im Workflow verwendeten Daten von Bedeutung. Dafür ist das Interface `ProcessData` zu verwenden, das eine Abfrage und Änderung dieser Daten erlaubt (Abbildung 92). Die Daten selbst werden in einem `DataElement`-Objekt gespeichert, das eine Liste von Namen/Wert-Paaren bereit stellt. Die IDL des `Participants` und die übrigen Teile des Objektmodells finden sich in Anhang F.

Der JFlow-Ansatz verwirklicht nicht ein ’CORBA-WFMS’, das alle funktionalen Bereiche durch Objekte realisiert, sondern lediglich eine Schnittstelle zu einem monolithischen WFMS, das die Steuerung übernimmt und nur geringen Einfluß durch die CORBA-Objekte erlaubt. Dem fallen auch die Möglichkeiten zur Modellierung von Workflows, der funktionalen Erweiterung oder ad-hoc Workflows zum Opfer. Andererseits vereinfacht dies auch die Abbildung auf die Schnittstelle.

```
struct DataElement {
   WfName name;
   any value;
};
typedef sequence<DataElement> DataElementList;

interface ProcessData {
   attribute WfName name;
   DataElement get_data_element(in WfName name)
      raises (InvalidName, DataElementUndefined);
   DataElementList get_data_element_list(in WfNameList name_list)
      raises (InvalidName, DataElementUndefined);
   void assign_data_element(in DataElement data_element)
      raises (InvalidName, InvalidType, DataElementAssignmentFailed);
   void assign_data_element_list(in DataElementList data_element_list)
      raises (InvalidName, InvalidType, DataElementAssignmentFailed);
};
```

Abbildung 92: DataElement und ProcessData (IDL)

5.5.2 Die C++ Schnittstelle von FlowMark

Zur Erreichung der Workflow-Funktionalität dient in CASSY das Produkt FlowMark V2.3 von IBM. Wir beschreiben hier kurz die wesentlichen Punkte der Schnittstelle von FlowMark [IBM 1995], die auf das JFlow-Objektmodell abgebildet werden (siehe Abschnitt 5.5.3). Abbildung 93 zeigt eine Übersicht der wesentlichen Klassen der Flowmark Client API für C++. Die Instanzen der Flowmark-Klassen sind stets nur Proxy-Objekte der in Flowmark verwalteten und gegebenenfalls persistenten Objekte. Wir beschreiben kurz die Funktionalität der wesentlichen Objekte, eine umfassende Beschreibung ist in [IBM 1995] zu finden.

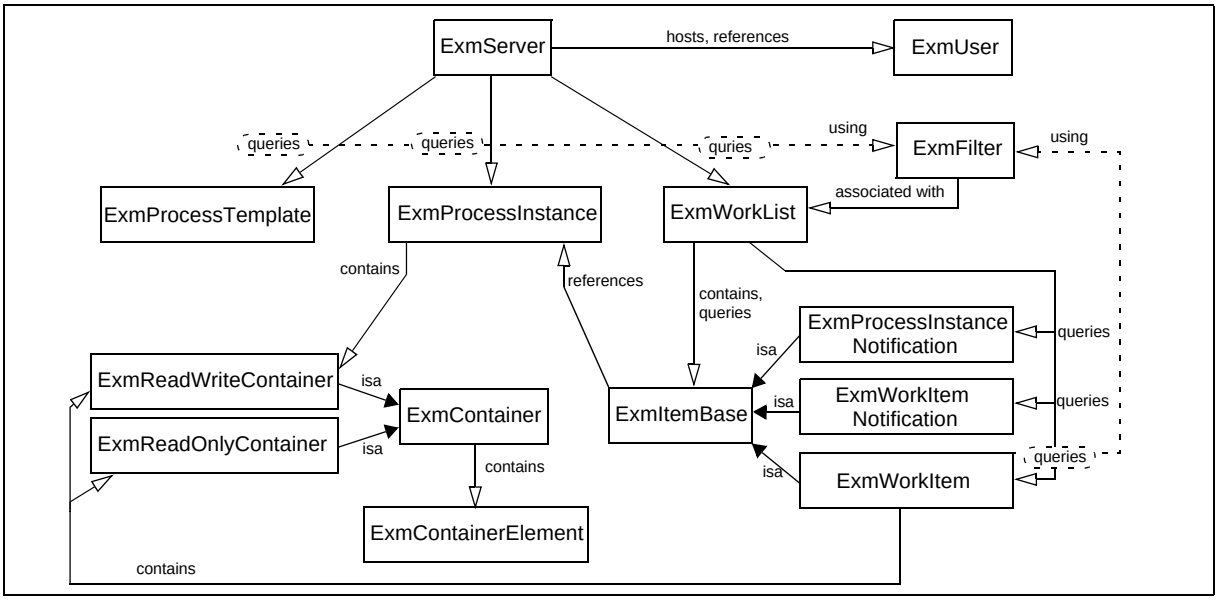


Abbildung 93: Klassen der C++ Client API von Flowmark

ExmServer

Der Server realisiert vor allem die Sitzungsverwaltung (Verbindungsaufbau, Autorisierung, Anmelden, etc.) mit dem Flowmark-Server. Ein Benutzer (`ExmUser`) darf nur einmal pro Server angemeldet sein. Weiter erlaubt der Server die Abfrage von Benutzerinformationen und das Ändern von Passwörtern. `ExmServer` bietet Anfrage-Operationen für Listen von Prozeßdefinitionen, Prozeßinstanzen und Worklisten an. Diese Anfragen (queries) werden mit Hilfe von `ExmFilter`-Objekten formuliert.

ExmProcessTemplate

Eine verfügbare Workflow-Definition wird durch das Objekt `ExmProcessTemplate` repräsentiert. Es existieren verschiedene Operationen zum Zugriff auf die Attribute des Templates (Name, Kategorie, Beschreibung, Administrator, etc.). Die Operation `CreateInstance()` erzeugt eine konkrete Instanz einer Definition. Funktionen zur Erzeugung und Manipulation von Definitionen werden nicht angeboten, dies ist lediglich durch die Buildtime-Komponente von Flowmark möglich.

ExmProcessInstance

Eine Prozeßinstanz besitzt i.w. die gleichen Attribute wie das entsprechende Template und zusätzliche, wie zum Beispiel die Startzeit und den Starter der Instanz. Zur Durchführung stehen folgende Operationen zur Verfügung: `Start()`, `Restart()`, `Suspend()`, `Resume()`, `Terminate()`. Desweiteren existieren Funktionen zur Abfrage des aktuellen Zustands sowie der Bestimmung der Eingabedatenstruktur.

ExmWorkList

Jedem Flowmark-Benutzer ist wenigstens eine Worklist zugeordnet. Zur Gruppierung von Work Items besteht die Möglichkeit weitere Worklisten zu verwenden. Die Work Items können zudem durch einen der `ExmWorkList` zugeordneten `ExmFilter` aussortiert werden.

ExmItemBase, ExmWorkItem

`ExmItemBase` ist die Basisklasse der in einer Worklist enthaltenen Items und realisiert die Abfrage deren Attribute, etwa Name, Beschreibung, Status, Startzeit, Startbedingungen, Akteure. Ein `ExmWorkItem` ist eine Spezialisierung von `ExmItemBase` und repräsentiert eine zur Ausführung bereite oder in Bearbeitung befindliche Aktivität. Es bestehen zusätzlich Operationen, um die Referenzen auf Ein- und Ausgabestrukturen der Work Items zu erhalten. Ein Work Item kann auf eine andere Worklist verschoben werden, wo auch deren Status verändert werden kann (ähnlich der Prozeßinstanz). Ein Item besitzt Attribute zur Beschreibung externer Applikationen. Eine zugeordnete Applikation kann gestartet werden, entweder durch den sogenannten *Tool Agent*, basierend auf der Operation `Start()`, oder durch die Operation `Checkout()`, die externe Anwendungen mit Eingabedaten startet und nach deren Beendigung mit der Operation `CheckIn()` die entsprechende Ausgabestruktur zurückliefert. Es existiert in der Flowmark Client API keine direkte Entsprechung für eine Aktivität.

Exm(ReadWrite/ReadOnly)Container

Die Ein- und Ausgabestrukturen für Work Items werden durch von `ExmContainer` abgeleitete Klassen repräsentiert. Ein Container verwaltet eine hierarchisch organisierte Attributliste aus Name/Wert-Paaren. Die elementaren Datentypen sind `STRING`, `FLOAT` und `LONG`. Die Elemente der Attributliste sind Instanzen von `ExmContainerElement`. Ein Zugriff auf diese erfolgt über vollqualifizierte Pfadnamen, z.B. liefert `ExmContainerElement::GetElement(Person[42].Adresse[0].Wohnort)` eine Zeichenkette mit dem entsprechenden Eintrag zurück. Ein `ExmReadOnlyContainer` gestattet lediglich lesenden Zugriff, ein `ExmReadWriteContainer` ermöglicht auch Änderungen. Die Struktur der vorliegenden Daten selbst kann jedoch nicht geändert werden.

5.5.3 Implementierung von JFlow

In Abbildung 94 wird ein Überblick über die JFlow-Implementierung gegeben. Über einen ORB werden die JFlow-Objekte für eine Abbildung auf die CASSY-IDL zur Verfügung gestellt. Der Flowmark-Server kann auf einem dedizierten Rechner betrieben werden. Die Flowmark Client API erlaubt den Zugriff auf diesen mittels einer TCP/IP-Verbindung. Darauf setzt unsere JFlow-Implementierung auf. Die JFlow-Objekte können auf weiteren Host-Rechnern über IIOP genutzt werden. Dies ist die Stelle, an der die Abbildung auf CASSY-Objekte vollzogen wird (siehe unten). Die Implementierung von JFlow erfolgte in C++, als ORB wurde ORBacus genutzt [HEIß 1998].

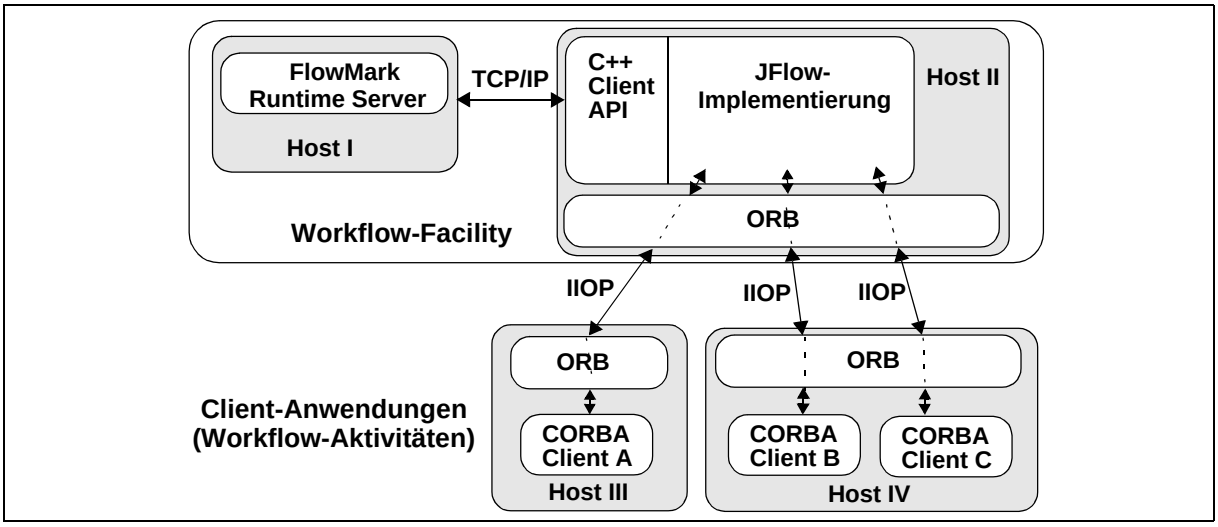


Abbildung 94: Übersicht der JFlow-Implementierung

Zur Durchführung der Implementierung war die Abbildung der Flowmark-Objekte auf JFlow-Objekte notwendig. Eine eins-zu-eins Abbildung ist aufgrund der unterschiedlichen Konzepte nicht möglich, jedoch bestehen prinzipielle Affinitäten zwischen den verschiedenen Klassen, wie sie in Tabelle 9 dargestellt sind. Dabei wurden Klassen, die nur marginal bei der Abbildung genutzt werden, kursiv dargestellt.

JFlow-Element	Flowmark C++ Client API
WorkflowManager	ExmServer
ProcessDefinition	ExmProcessTemplate, *ExmServer*
ProcessInstance	ExmProcessInstance
ActivityInstance	*ExmWorkItem*
WorkItem	ExmWorkItem
ProcessData	ExmReadWriteContainer, ExmReadOnlyContainer, *ExmContainer-Element*
DefinitionIterator	*ExmServer*, *ExmFilter*
ElementIterator	*ExmWorkList*, *ExmServer*, *ExmFilter*
HistoryIterator	(keine Entsprechung)
Participant	ExmUser

Tabelle 9: Abbildung zwischen JFlow und Flowmark

Es bestehen hier einige Diskrepanzen zwischen diesen Ansätzen. Die schwerwiegendsten sind:

- Einige der Operationen von JFlow sind nicht über die Flowmark C++ API zu realisieren, bspw. das Setzen von Namen von Workflow-Elementen (`ProcessDefinition`, `ProcessInstance`, `ActivityInstance` und `WorkItem`).

- Es gibt keine zugreifbare Historienfunktion in der Flowmark API (obwohl durchaus eine entsprechende Funktionalität von Flowmark realisiert wird).
- Es existieren in der Flowmark API keine expliziten Aktivitätsobjekte, jedoch kann man durch `ExmWorkItem` eine weitgehend ähnliche Funktionalität erreichen.
- Das `Participant`-Interface von Flowmark ist nur unzureichend spezifiziert.

Im folgenden stellen wir die Implementierung ausgewählter Teile vor, die einige Lösungen für von Problemen der Abbildung verdeutlichen. Eine detaillierte Beschreibung ist in [HEIß 1998] nachzulesen.

Sitzungsverwaltung

Der `ExmServer`[1] wird für jede neue Sitzung instantiiert, während der `WorkflowManager` als sitzungsübergreifend konzeptioniert wurde. In der Implementierung wurde der Weg gewählt, den `WorkflowManager` vom `ExmServer` abzuleiten und damit der Vorgabe von Flowmark zu folgen. Eine andere Option wäre die Einführung eines übergeordneten Managers gewesen, was die Übersichtlichkeit erschwert hätte. Der `WorkflowManager`, wie er im JFlow-Objektmodell spezifiziert wurde, besitzt keine Funktionen zur Anmeldung an einen Workflow-Server, was hier nun auch hinzugefügt wurde. Zur Autorisierung sind damit entsprechende Flowmark-Konten notwendig. Die Erzeugung von `WorkflowManager`-Objekten wird durch eine neu eingeführte `WorkflowManagerFactory` durchgeführt, welche die Konteninformation benötigt. Somit wurde ein Kompromiß zwischen Veränderung des JFlow-Modells und der logischen Funktionalität eingegangen.

Behandlung von Proxy-Objekten

Die instantiierten JFlow-Objekte nutzen transiente Proxy-Objekte, nämlich die entsprechenden Flowmark-Objekte. Dies kann zu Problemen führen, wie am folgenden Beispiel illustriert wird: Bei der Erzeugung eines neuen `ProcessDefinition`-Objekts durch den Aufruf `WorkflowManager::get_definition("a_process_definition")`, wird dieses Objekt an das entsprechende `ExmProcessTemplate` gebunden, das selbst die persistente Workflow-Definition mit dem `key` '`a_process_definition`' repräsentiert. Die Aufrufe an das JFlow-Objekt werden anschließend an das Flowmark-Objekt weitergeleitet. Da das `ProcessDefintion`-Interface keine Operation zur Zerstörung des Proxy-Objekts nach beendeter Benutzung besitzt, kann es geschehen, daß bei einem weiteren Aufruf von `get_definition ("a_process_definition")` ein neues Proxy-Objekt erzeugt wird und damit zwei Proxies für das gleiche Objekt existieren. Zwar besitzen einige JFlow-Interfaces eine `destroy()`-Operation, jedoch dient diese zur Zerstörung des persistenten Objekts. Um die Problematik mehrfacher Proxy-Objekte zu lösen, wurde in den Proxy-Klassen Tabellen eingeführt, die alle Instanzen speichern. Durch einen als `protected` deklarierten Konstruktor und die Einführung der Operation `ObtainRef` (man spricht hier auch von einem *named constructor*) wurde auch die kontrollierte Erzeugung und Behandlung von Proxy-Objekten sichergestellt.

Implementierung von `ProcessData`

Die Konzepte zur Verwaltung Workflow-relevanter Daten bei JFlow und Flowmark ähneln sich. Beide nutzen Name/Wert-Paare und Ein-/Ausgabestrukturen. Die Datentypen von Flowmark lassen sich leicht auf die CORBA-Datentypen `CORBA::string`, `CORBA::long` und `CORBA::double` abbilden. Die rekursiven Strukturen von Flowmark werden zum einen durch den Namen der Flowmark Struktur und der Wert durch eine Liste der Elemente repräsentiert.

1. Wir verzichten im folgenden auf die deutliche Benennung von Flowmark- gegenüber JFlow-Objekten, da Flowmark-Objekte durch das Präfix `Exm` identifizierbar sind.

JFlow-Anwendung

Wir beschreiben hier kurz eine Beispielanwendung, um zu zeigen wie der Zugriff auf die JFlow-Implementierung erfolgt. In Abbildung 95 ist ein Auszug aus einem JFlow-Client in C++ dargestellt[1]. Zu Beginn wird der ORB und der BOA (*basic object adapter*) initialisiert, um mit dem ORB auch in Verbindung treten zu können. Danach wird eine `WorkflowManagerFactory` abgefragt, die bereits gestartet sein muß. Die Factory wird genutzt um einen `WorkflowManager` zu erzeugen, der zur Verwendung von JFlow notwendig ist. Mittels des Managers können nun die wesentlichen Elemente von JFlow abgefragt werden. In dem Beispiel suchen wir eine Prozeßdefinition mit dem Schlüssel "`Sample_Process`", die uns in einem `ProcessDefinition`-Objekt zurückgegeben wird. Aufgrund dieser Definition können wir eine `ProcessInstance` erzeugen (`ProcessDefinition::start()`), die auch von Flowmark gestartet wird.

```
//Initialisierung von ORB und BOA
orb = CORBA_ORB_init(argc, argv); boa = orb->BOA_init(argc, argv);

//WorkflowManagerFactory holen
CORBA_Object_var objPtr;
objPtr = orb->get_inet_object(HOST, PORT, OB_SRVNAME);
factory = BoWorkflow_WorkflowManagerFactory::_narrow(objPtr);

//Logon
BoWorkflow_WorkflowManager_var wfm = factory->open(FLOWMARKSRV, FLOWMARKDB, USER, PASSWORD);

//Abfragen einer Prozeßdefinition
const char[] def_name = "Sample_Process";
BoWorkflow_Filter filter;
filter.type = 1; filter.filter_specification = def_name;
BoWorkflow_DefinitionIterator_var iter = wfm->list_definitions(filter);
BoWorkflow_ProcessDefinition_var process_definition;
iter->get_next(process_definition);

//Ausgabe von Name und Beschreibung einer Prozeßdefinition
cout << process_definition->key() << endl;
cout << process_definition->description() << endl;

//Erzeugung und Starten einer Prozeßinstanz
BoWorkflow_ProcessInstanec_var process_instance;
process_instance = process_definition->create_process_instance("Sample_instance");
process_instance -> start();
```

Abbildung 95: C++ JFlow-Client (C++, Auszug)

5.5.4 Abbildung von JFlow auf CASSY-IDL

Die Abbildung von JFlow auf die WorkflowActivity von CASSY entspricht einer Client-Anwendung von JFlow, wie sie oben beschrieben wurde. Aus der Funktionalität der vorliegenden JFlow-Implementierung ergeben sich Einschränkungen. Beispielsweise fehlen Dienste zur Definition von Workflows, zur Verwaltung von Workflow-Daten und zur Organisationsverwaltung. Die nutzbare Funktionalität umfaßt vor allem die Durchführung bereits existierender Workflows. Ein Zugriff auf die Definitions- und (interne) Verwaltungsebene ist nur in geringem Maße möglich. Dies begründet sich zum einen anhand der ungenügend spezifizierten JFlow-Facility ([SCHULZE 1997], [HEIß 1998]) und zum anderen in der in gewissen Bereichen fehlenden Unterstützung durch die C++-API von FlowMark. Besonders betroffen ist die Behandlung von Workflow-Definitionen, die nicht über die API von Flowmark abfragbar sind. Dies kann man umgehen, in dem man die vorliegenden Definitions-Dateien eines Flowmark-Workflows in eine Workflow-Aktivität durch ein entsprechendes Programm einliest. Jedoch muß dann beachtet werden, daß bei Änderung der Definition in Flowmark diese Datei wiederum neu einzulesen ist.

1. Das Programmbeispiel enthält der Übersichtlichkeit wegen keine Ausnahme- oder Fehlerbehandlung

Die obige Implementierung erschwert leider den Zugriff auf Workflow-Daten und Workflow-relevante Daten. Zwar verwendet FlowMark in der hier verwendeten Version das objektorientierte Datenbanksystem ObjectStore, jedoch bestehen nicht nur ungenügende Möglichkeiten eines Zugriffs via der FlowMark-API, sondern auch bezüglich der rudimentären Datenbehandlung in JFlow (`ProcessData`-Klasse). Dies schränkt die Integration von Daten für CASSY erheblich ein.

Wir betrachten zwei Arten von Daten, die in eine Workflow-Aktivitätsdefinition des ADM einzubringen sind. Zum einen sind dies die Workflow-relevanten Daten, die nur zur Laufzeit eines Workflows verfügbar sind (die Ein- und Ausgabedaten von Aktivitäten), und Daten, die durch die in den Aktivitäten eines Workflows aufgerufenen Applikationen verwendet werden und dem WFMS nicht bekannt sind (Applikationsdaten). Die Applikationsdaten können unterstützt werden, in dem sie in einer Designflow-Aktivität über `uses` explizit bekannt gemacht und wie in Abschnitt 5.2 beschrieben in den Informationsraum eingebracht werden. Die Einbringung von Workflow-relevanten Daten ist aufwendiger, da hier die begrenzte Laufzeit eines Workflows und damit auch die begrenzte Lebensdauer der Daten zu berücksichtigen ist. Wir sorgen daher dafür, daß diese Objekte zur Beendigung einer ADM-Workflow-Aktivität aus dem Informationsraum entfernt werden. Die ADM-Objekte selbst werden als spezialisierte `BasicResourceAdapter` realisiert, die um die Operationen von `ProcessData` erweitert werden und nur lesenden Zugriff gestatten, um die korrekte Durchführung des Workflows zu gewährleisten. Als Erweiterung ist es denkbar auch schreibenden Zugriff zu gestatten. Dafür müssen jedoch zusätzliche Informationen über die Workflow-Aktivitäten aufgenommen werden, um automatisch zu entscheiden, wann ein solcher Zugriff möglich ist. So darf ein schreibender Zugriff nicht während der Durchführung einer Aktivität des Workflows stattfinden, die diese Daten verändert.

Die neueren Vorschläge für eine Workflow-Facility für die OMG stellen eine weitgehendere Funktionalität zur Verfügung, wie zum Beispiel die Änderungsverfolgung von Workflow-Daten. Dies läßt hoffen, daß die meisten existierenden Workflow-Systeme auch dahingehend erweitert werden, die Funktionalität nach außen bereit zu stellen, um nicht mehr umständlich die Workflow-Definitionen einlesen zu müssen.

5.5.5 Die aktuelle Workflow Management Facility der OMG

Seit der erfolgten Implementierung wurden einige Änderungen bezüglich der Workflow Facility durch die Mitglieder der OMG vorgenommen [OMG 2000]. Dabei wurden sinnvolle Funktionalitäten ergänzt und problematische Bestandteile berichtigt, was zur Hoffnung beiträgt, ein insgesamt stimmigeres Konzept realisieren zu können. Besonders wollen wir folgende Punkte hervorheben:

- Die Verfolgung von Abläufen wurde verbessert.
- Eine umfassende Historien-Funktionalität wurde integriert.
- Prozeßdefinitionen werden leider noch immer nicht als Bestandteil integriert.
- Das Objektmodell stellt immer noch eine relativ simple Schnittstelle für existierende WFMS dar.

Damit haben sich einige Punkte verbessert, die jedoch nicht entscheidend genug für eine Überarbeitung der vorliegenden Workflow Facility waren.

5.6 Die Groupware-Facility

Um Groupware-Funktionalität bereitzustellen, wurden verschiedene Produkte und Bibliotheken begutachtet. Leider zeigte sich, daß derzeit keine existierenden Systeme über eine umfassende Funktionalität verfügen, wie wir dies wünschen. Lotus Notes von IBM bietet nur Unterstützung asynchroner Zusammenarbeit [SENKEL 1998]. Das Toolkit GroupKit [ROSEMAN UND GREENBERG 1996] konzentriert sich hingegen auf synchrone Zusammenarbeit, unterstützt die Verwaltung des organisatorischen Umfelds (Rollen, Rechte usw.) oder Ressourcenverwaltung nur ungenügend [MUCHITSCH 1998]. Daher wurde eine eigenständige Groupware-Facility entwickelt, die auf dem Business Object Proposal für die OMG [NIIIP 1997] beruht.

5.6.1 Eine Übersicht zur Groupware-Facility

Für die Implementierung von CSCW-Diensten besteht leider kein genormter Vorschlag. In [ALMASI ET AL. 1994] wird jedoch eine breite Palette von Funktionalitäten für die Realisierung von CSCW-Systemen vorgeschlagen. In [SCHMIDT UND RODDEN 1996] findet sich ein Dienstkonzept für CSCW-Plattformen. Auf Grundlage dieser Publikationen wurde ein solcher Dienst modelliert und prototypisch implementiert. Hierbei wurde auch Rücksicht auf die Integrierbarkeit mit der Workflow Management Facility genommen. Die verschiedenen Dienste der CSCW Facility umfassen folgende Funktionalitäten [FRANK 1998]:

- Gruppenverwaltung,
- Ressourcenverwaltung,
- Informationsraumverwaltung,
- Sitzungsverwaltung,
- Kommunikationsdienste (asynchron/synchron) und
- Awareness-Unterstützung.

Die entwickelte Groupware-Facility basiert auf der Verwaltung von Organisationsdaten (Personen, Gruppen, Rollen), Workspaces als gemeinsame Arbeitsräume, der Verwaltung der gemeinsamen Ressourcen und dem Management von Gruppenaktivitäten ([MUCHITSCH 1998], siehe Abbildung 96). Das Ziel der Groupware-Facility ist es nicht nur externe Applikationen einzubinden, sondern auch eine Entwicklungsplattform für Applikationen zur Verfügung zu stellen. Daher umfassen die Ressourcen auch Event Channels, Realtime Channels und weitere Komponenten, die zur Erstellung von Groupware-Applikationen notwendig sind.

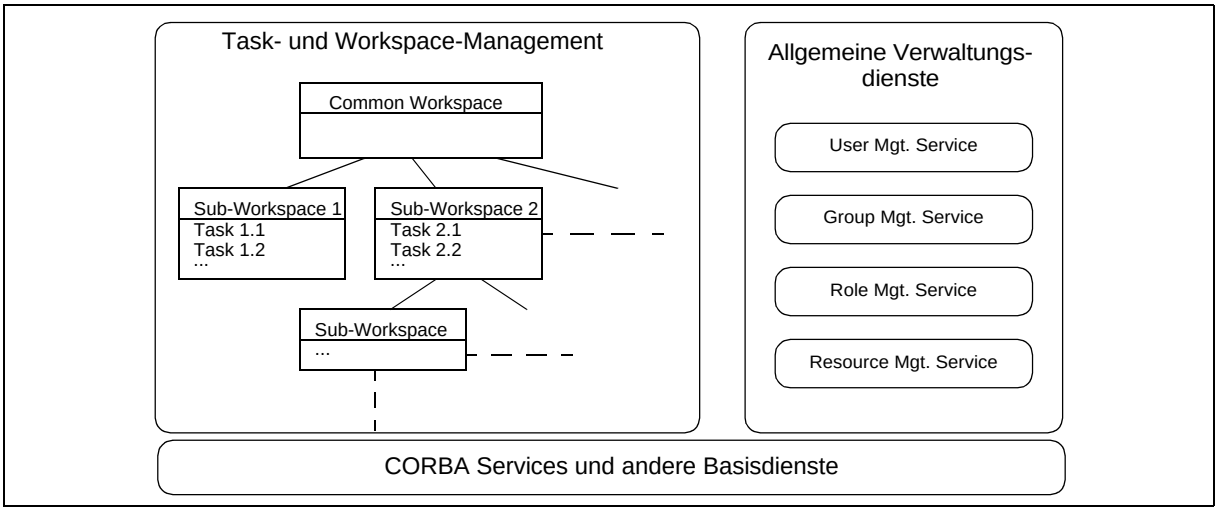


Abbildung 96: Grundlegende Aspekte und Dienste der CSCW-Facility

Wie es für CORBA Facilities verlangt wird, wurde bei dem Entwurf auf die Verwendung existierender CORBA Services geachtet. Weiterhin basiert das System auf anderen Basisdiensten, wie Versionierungs- und Replikationsdienst, Daten- und Dateienzugriff usw. Das Objektmodell (siehe Abbildung 97) selbst basiert auf dem Vorschlag des NIIIP-Konsortiums für die Business Object Task Force bezüglich Task und Session Objects [NIIIP 1997]. Über die Workspaces wird auf die allgemeinen Verwaltungsdienste zugegriffen. Die Workspaces selbst sind hierarchisch gegliedert, gruppenorientiert und beinhalten die Funktionalität zur Verwaltung von sogenannten Tasks (Anwendungsaktivitäten). Diesen Tasks[1] werden Ressourcen (Daten, Hardware, CORBA Event Channels, etc.), Personen und Anwendungen zugeordnet.

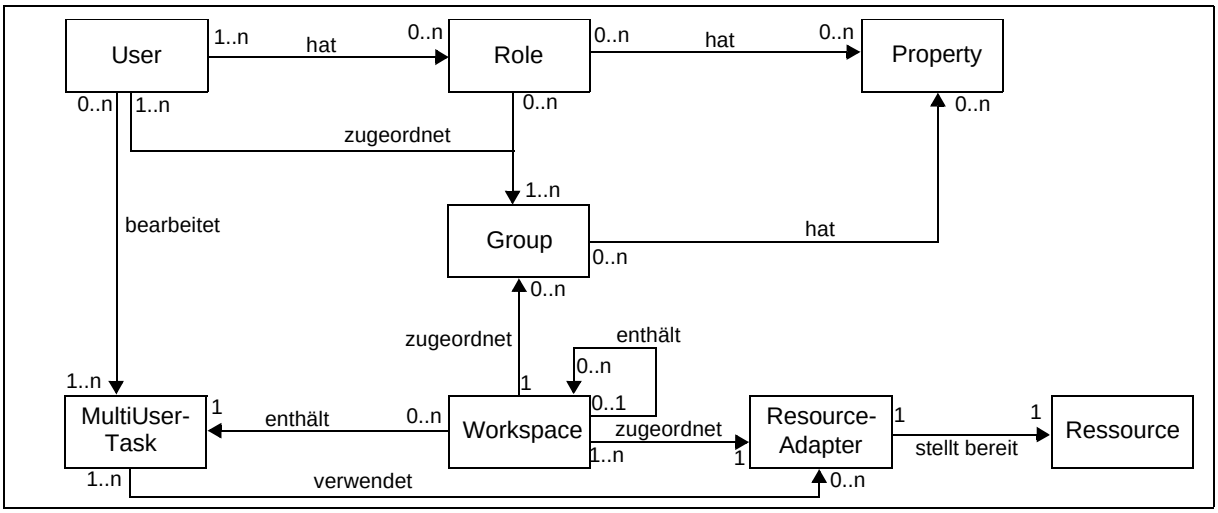


Abbildung 97: Objektmodell der CSCW Facility

Mechanismen wie die Kommunikation (z. B. Email und Videokonferenz) können durch diese Tasks und die Bereitstellung von CORBA Event Channels realisiert werden. Der Aufruf einer Werkzeuganwendung wird auch auf einen Task abgebildet. Durch die Workspace-Hierarchie, die damit erreichte Strukturierung und durch die Verwendung von Tasks ist die Darstellung komplexer Aufgaben möglich.

5.6.2 Spezifische Aspekte der Groupware-Facility

Die Schnittstelle der Groupware-Facility wurde soweit sinnvoll in die CASSY-Schnittstelle übernommen. Daher sind die meisten Interfaces bereits in Abschnitt 5.3 vorgestellt worden. Die Funktionalität der Workspace-Verwaltung wurde hingegen nicht übernommen, da sie, wie in Abschnitt 4.6 diskutiert, nicht unseren Anforderungen für Informationsräume entspricht. Wir werden hier auf die Konzepte der Ressourcen- und Workspace-Verwaltung eingehen.

Workspaces

Ein Workspace (Interface `Workspace`, Abbildung 98) stellt die gemeinsame Arbeitsumgebung für eine Gruppe von Personen bereit und dient damit als organisatorischer Rahmen kollaborativer Tätigkeiten (Tasks) in der Gruppe. Zur Unterstützung der Tätigkeiten beinhaltet ein Workspace Ressourcen, auf welche bei der Ausführung der einzelnen Tasks zurückgegriffen werden kann. Es gibt öffentliche und private Workspaces. Einen privaten Workspace können nur Personen der zugeordneten Gruppe nutzen und auf die darin verfügbaren Ressourcen direkt zugreifen. Ein öffentlicher Workspace kann von allen Personen genutzt werden.

1. Die Tasks repräsentieren das Aktivitätenmodell der CSCW-Facility.

```
interface Workspace {
   enum WorkspaceAccessPolicy { Open, Closed }; enum MigrationDirection { Upward, Downward };

   readonly attribute string WorkspaceID; attribute string WorkspaceName;
   attribute WorkspaceAccessPolicy Policy;
   readonly attribute Workspace ParentWorkspace;

   Group UserGroup();
   string UserGroupID();
   boolean AssignUserGroup(in Group grp, in unsigned long mode);
   boolean ReplaceUserGroup(in Group newgroup, in unsigned long mode);
   boolean RemoveUserGroup(in unsigned long mode);

   boolean ChangeParent(in Workspace parent);
   boolean AddWorkspaceUser(in User newUser);
   boolean RemoveWorkspaceUser(in string userid);
   boolean is_WorkspaceUser(in string userid);
   boolean EnterWorkspace(in User usr);
   boolean SaveWorkspaceData();
   Workspace CreateSubWorkspace(in string name, in WorkspaceAccessPolicy pol, in boolean save);
   boolean AddSubWorkspace(in Workspace ws);
   boolean MigrateSubWorkspace(in string ws, in Workspace target);
   boolean MigrateActualWorkspace(in Workspace ws, in MigrationDirection direction);
   boolean RemoveSubWorkspace(in string wsid, in boolean delete_sub);
   boolean RemoveSubWorkspaceEntry(in string wsid);
   Workspace LookupSubWorkspace(in string wsid, in unsigned long depth);
   void ListSubWorkspaces(out WorkspaceList workspaces);
   boolean ReadSubWorkspaces(out WorkspaceIDList workspaces);
   boolean LoadSubWorkspaces(in boolean auto_save, in boolean ref_actions);
   boolean is_SubWorkspace(in Workspace ws, in unsigned long level);
   boolean is_ParentWorkspace(in string wsid, in unsigned long level);
   boolean Merge(in Workspace ws);

   boolean AddResourceAdapter(in BasicResourceAdapter adapter,in unsigned long mode);
   boolean RemoveResourceAdapter(in string resourcename,
                  in BasicResourceAdapter::ResourceType resource_type,in unsigned long mode);
   BasicResourceAdapter LookupResourceAdapter(in string resourcename,
                  in BasicResourceAdapter::ResourceType resource_type);
   void ListResourceAdapter (out BasicResourceAdapterList adapters);
   boolean SaveResourceAdapter();
   boolean ReadResourceAdapter(out ResourceAdapterInformationList list);

   MultiUserTask CreateTask(in MultiUserTask::TaskAccessPolicy taskpolicy,
                  in User responsible, in string description);
   boolean AddTask(in MultiUserTask task);
   boolean RemoveTask(in string taskid);
   MultiUserTask LookupTask(in string taskid);
   boolean ListTasks(out MultiUserTaskList tasks);
   boolean SaveTasks();
   boolean LoadTasks(in boolean referential_actions);
   boolean ActiveTasks();

   void dump();
   void removeFiles();
};
```

Abbildung 98: Workspace Interface (gekürzte IDL)

Um der Dynamik von Gruppenprozessen zu entsprechen, können neue Sub-Workspaces in einem Workspace gebildet werden, in die auch Ressourcen transferiert werden können. Dies erfolgt nach bestimmten Regeln:

1. Die Zuordnung der untergeordneten Workspaces erfolgt immer nur an Untergruppen.
2. Die Rollen der User innerhalb der beiden Gruppen sind unabhängig voneinander.
3. Die Ressourcen des übergeordneten Workspaces sind auch in den untergeordneten Workspaces verfügbar, aber nicht umgekehrt.
4. Die einem Workspace untergeordneten Workspaces sind nur diesem sichtbar.
5. Die Tasks können bei ihrer Ausführung sowohl auf die Ressourcen des Workspaces zugreifen, dem sie zugeordnet sind, als auch auf alle Ressourcen, welche in übergeordne-

ten Workspaces verfügbar sind.

6. Die Workspace-spezifischen Eigenschaften, wie Name, ID oder Zugangspolitik der untergeordneten Workspaces sind unabhängig von denen der übergeordneten Workspaces.

Weiter bietet das Workspace-Management die Möglichkeit Workspaces zu migrieren oder zusammenzuführen. Dies kann z.B. erforderlich werden, wenn ein Workspace gelöscht wird, aber die untergeordneten Workspaces erhalten bleiben sollen.

Tasks

Im Gegensatz zu dem ADM stellen `MultiUserTasks` - als Ausführungseinheit der CSCW-Facility - eine gröbere Strukturierung von Aktivitäten dar. Sie besitzen eine mächtige Schnittstelle zur Ausführung mehrerer Applikationen und werden typischerweise durch eine Menge von Akteuren durchgeführt, wobei einer von ihnen der verantwortliche User ist (Attribut `ResponsibleUserID`, Abbildung 99). Zudem können ihnen Ressourcen (`AssignResourceAdapter()`) und Workspaces (`AssignWorkspace()`) zugewiesen werden. Die Task als ganzes wird durch die Funktionen `Start()`, `Suspend()`, `Restart()` und `Terminate()` gesteuert. Weiter wurde ein Checkpoint-Konzept eingeführt, welches das Zurück- und Wiederaufsetzen von Tasks unterstützt (`SetCheckpoint()`, `RecoverToCheckpoint()`).

```
interface MultiUserTask {
   enum TaskState {NotStarted, Running, Suspended, Terminated};
   enum TaskAccessPolicy {Open, Closed};
   readonly attribute stringTaskID; readonly attribute TaskStateState;
   readonly attribute TaskAccessPolicy AccessPolicy;
   readonly attribute stringResponsibleUserID;readonly attribute UserResponsibleUser;
   readonly attribute stringTaskdescription;readonly attribute stringWorkspaceID;
   readonly attribute WorkspaceAssignedWorkspace;readonly attribute booleanSupportSuspend;

   boolean AssignNewResponsibleUser(in User usr, in boolean auto_save);
   boolean AssignWorkspace(in Workspace ws);
   boolean Move(in Workspace ws);

   boolean AssignResourceAdapter(in BasicResourceAdapter adapter, in unsigned long mode,
                           in boolean auto_save);
   boolean RemoveResourceAdapter(in string resourcename, in ResourceType resource_type,
                           in unsigned long mode, in boolean auto_save);
   boolean FreeAllResources();
   //weitere ResourceAdapter-Operationen gekürzt...

   boolean AssignUser(in User usr, in unsigned long mode, in boolean auto_save);
   boolean RemoveUser(in string userid, in unsigned long mode);
   void GetAssignedUsers(out UserList users);
   boolean ReadAssignedUsers(out UserIDList users);
   boolean Join(in string userid,in boolean auto_save);
   boolean Leave(in string userid);
   boolean is_Participate(in string userid);
   boolean Connect(in string userid);
   boolean Disconnect(in string userid);
   boolean is_connected(in string userid);

   //behandlung von Applikationen
   boolean AddApplication(in string application_name, in string param1, in string param2,
                     in string param3, in boolean client_application, in boolean auto_save);
   boolean StartApplication(in string application_name,in boolean auto_save);
   //... weitere Operationen gekürzt: SuspendApp, RestartApp, RemoveApp, ListApp, RestoreApp

   boolean Start(); boolean Suspend(); boolean Restart(); boolean Terminate();

   boolean SetCheckpoint(out unsigned long checkpoint_id);
   boolean RecoverToCheckpoint(in unsigned long checkpoint_id);
};
```

Abbildung 99: MultiUserTask Interface (gekürzte IDL)

Ressourcen

Die `ResourceAdapter`-Klasse dient der einheitlichen Verwaltung heterogener Ressourcen. Die Schnittstelle wurde mit Änderungen (z.B. ohne Workspace-Behandlung) in die CASSY-IDL aufgenommen. Diese Ressourcen müssen als CORBA-Objekte der Klasse `Resource` vorliegen. Jede `ResourceAdapter`-Instanz wird genau einem Workspace zugewiesen und steht damit den Tasks des Workspace zur Verfügung. Eine solche Ressource kann auch in einen anderen Workspace migriert werden, außerdem ist sie über die untergeordneten Workspaces zugreifbar. Die Groupware-Facility berücksichtigt auch den konkurrierenden Zugriff auf die Ressource durch mehrere Tasks. Ressourcen, die nicht Multi-User-fähig sind, können auch exklusiv einzelnen Tasks zugeordnet werden. Eine Änderung dieser Zuweisung ist erst nach Beendigung des entsprechenden Tasks möglich. Die tatsächlichen Ressourcen werden nur lose an den `ResourceAdapter` gekoppelt (Referenz auf ein `Resource`-Objekt).

5.6.3 Einbindung der Groupware-Facility in CASSY

Die Schnittstelle der Groupware-Facility diente als Vorlage für die Ressourcen und Organisationsstrukturierung (Akteure, Rollen) für CASSY. Es wurden aber die Operationen für die Behandlung von Workspaces in Tasks und Ressourcen entfernt, da diese nicht unserem Konzept des Informationsraums entsprechen. Dies ermöglicht eine weitgehend reibungslose Integration der Groupware-Facility. Die Groupware Tasks erfüllen funktional die Anforderungen an die Groupware-Aktivität des ADM.

Vereinfacht wurde die Implementierung, da der Quelltext der Facility vorlag, wodurch kleine Änderungen der Facility vorgenommen werden konnten (im Gegensatz zu einem geschlossenem System wie FlowMark). Problematisch ist hingegen die Verwaltung der Ressourcen und Tasks, die aus der Groupware-Facility stammen, bezüglich der Behandlung in den Workspaces. Für CASSY bietet es sich einfachheit halber an alle Ressourcen aus der Groupware-Facility in den Wurzel-Workspace zu verschieben. Nicht-Multi-User-fähige Ressourcen müssen durch ein entsprechendes Zuordnungsverfahren, das typspezifisch in die CASSY-Ressourcen zu implementieren ist, den Aktivitäten exklusiv zugeordnet werden.

5.7 Die Protokollmaschine

Der letzte Bestandteil der Regelung in der Designflow-Schicht des ADM wird durch die Protokollmaschine erreicht, die zur Realisierung der Interaktionsprotokolle entwickelt wurde [BOZAK 2000]. Diese ist in Java programmiert und kann somit einfach in CORBA-Objekte integriert werden, die ebenfalls in Java programmiert sind. Da einige CASSY-Objekte in C++ vorliegen, mußte für diese ein anderer Weg gegangen werden, nämlich die Kopplung über die Interaktionsschnittstelle der betreffenden Entitäten. So werden empfangene Nachrichten an eine Instanz der Protokollmaschine, die als eigenständiges CORBA-Objekt realisiert ist, weitergereicht (Abbildung 100 links).

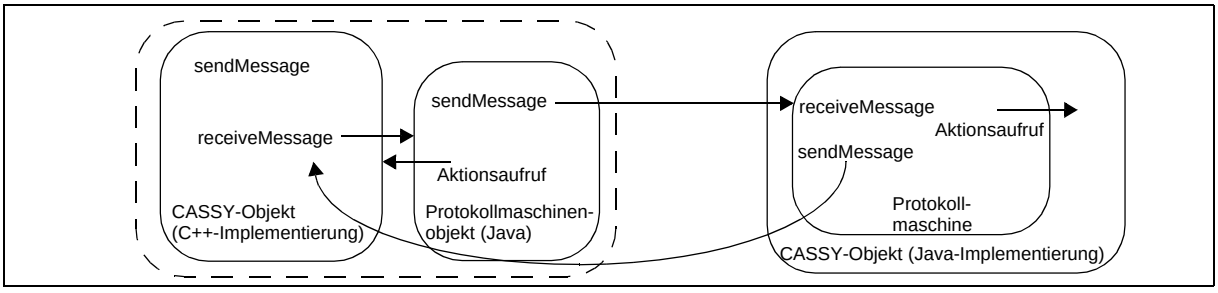


Abbildung 100: Integration der Protokollmaschine in C++ und Java-Objekte

Für Kommunikation zwischen Protokollmaschinen wurden KQML und KIF herangezogen (siehe Abschnitt 3.5.2). Dies ermöglicht eine einfache Definition der Konversationsmuster. In Abbildung 101 wird ein Auszug der Definition des Nutzungsprotokolls in Java dargestellt. Die Klasse `InteractionDefinition` dient als Basis für alle Protokolle. Über abgeleitete Klassen lassen sich Maschinen erzeugen (hier: `Usage::getInitiatorMachine(...)`), die das entsprechende Protokoll implementieren. Einem solchen Protokoll wird der Sender (hier[1]: `initiator`), Empfänger (hier: `counterpart`) und ein entsprechender Kontextidentifikator (`contextId`) übergeben. Die einzelnen Nachrichten (`KQMLmessage`) werden als Zeichenkette aufgebaut, die aus den in Abschnitt 4.7 beschriebenen Parametern bestehen. Vom Benutzer oder der Maschine änderbare Teile werden in spitzen Klammern gefaßt (siehe unten). Im Beispiel wird die Nutzungsanfrage mit dem Performative `use` erzeugt. Die Zustände des Protokollgraphen sind `ObserverState` und `AbstractState`, die auf Nachrichten warten bzw. diese senden. Eine Transition wird durch die Klasse `MessageObserverTransition` beschrieben. In dem vorliegenden Beispiel ist das der Übergang vom sendenden Zustand `s0` mittels der Nachricht `USAGE.actionRepertoire.getSendAction()`. Diese Methode ist die technische Realisierung der Send-Operation, welche die CASSY-Interaktionsmethoden aufruft.

```
public class Usage extends InteractionDefinition{
   public InteractionMachine getInitiatorMachine(String initiator, String counterpart,
                                                 String contextId) {
   InteractionMachine rval = null;
   ActionRepertoire actionRepertoire = this.getActionRepertoire();
   String sender = initiator;
   String receiver = counterpart;
   String context = contextId;

   KQMLmessage USAGE = new KQMLmessage("(use :object_ID <ID> :usage-type <type>  :context " +
      context + " :sender " + sender + " :receiver " + receiver + ")");
   // further messages...

   ObserverState o0 = new ObserverState( "O0");
   AbstractState s0 = AbstractState.getInstance( "S0" );
   // further states...

   MessageObserverTransition t0 = new MessageObserverTransition( o0, s0, "S(USAGE)", new
      Fact(USAGE), null, actionRepertoire.getSendAction());
   // further transitions...

   rval = new InteractionMachine(o0);
   rval.addState(s0);
   //...
   rval.addTransition(t0);
   //...
   return rval;
   }//InitiatorMachine()
} //class Usage
```

Abbildung 101: Definition des Nutzungsprotokolls (Java)

Die Nutzung von Java-Klassen erlaubt uns auch die Realisierung des Protokoll-Reservoirs, da die Klassen zur Laufzeit geladen werden können. Aktionen, die auf CASSY-Objekte wirken, werden in den entsprechenden Zuständen bzw. in den Zustandsübergängen aufgerufen.

1. Im Falle eines Komplementärprotokolls wäre der Sender der *counterpart* und der Empfänger der *initiator*.

5.8 Die Benutzungsschnittstelle

Das Konzept der Assistenzfunktion für Designflow-Systeme wird in [RITTER UND MITSCHANG 1997] diskutiert. Hierbei handelt es sich um die Schnittstelle zum Benutzer, der damit durch den Entwurfsprozeß geführt werden soll. D.h. es werden Dienste angeboten, aus denen der Benutzer die für ihn oder sie sinnvollen Aktionen wählen kann.

Für die Benutzungsschnittstelle wurden verschiedene Komponenten erstellt. Der Designflow-Editor erlaubt es, Designflows zu spezifizieren und in das System einzubringen (Abschnitt 5.8.1). Der Constraint-Editor ermöglicht es, Constraints zu definieren und die entsprechende XML-Definitionsdatei zu erzeugen, die von der Event-Engine genutzt wird (Abschnitt 5.8.3). Außerdem können zur Laufzeit die Constraints überwacht werden. Für eine übergeordnete Überwachung von Abläufen entstand ein Prototyp, der mittels Netzplantechnik Konflikte und Abhängigkeiten zwischen verschiedenen Arbeitsschritten übersichtlich darstellt (Abschnitt 5.8.4). Im Rahmen der Protokollmaschine wurde eine Oberfläche entwickelt, die es erlaubt, Protokollausführung zu verfolgen und Benutzer in die Durchführung von Protokollen einzubinden (Abschnitt 5.8.5).

5.8.1 Die Benutzungsoberfläche zur Spezifikation von Designflows

Wir haben in Abschnitt 5.4 die Funktionsweise der Event Engine vorgestellt und die Komponente Definitionswerkzeug eingeführt. Der Designflow-Editor xml2cassy (Abbildung 102) erlaubt es, in einer hierarchischen Sicht - eine sogenannte *tree view* - Designflows zu definieren (linke Darstellung). Der Wurzelknoten (`<cassy>`) beschreibt die Definitionssprache (siehe unten) für den Baum. Der erste Knoten enthält die Designflow-Aktivität `Standortplanung`. Die Spezifikation folgt dann der Definition der Designflow-Sprache des ADM und beschreibt Ressourcen, Subaktivitäten, Akteure, Constraints und das Entwurfsziel.

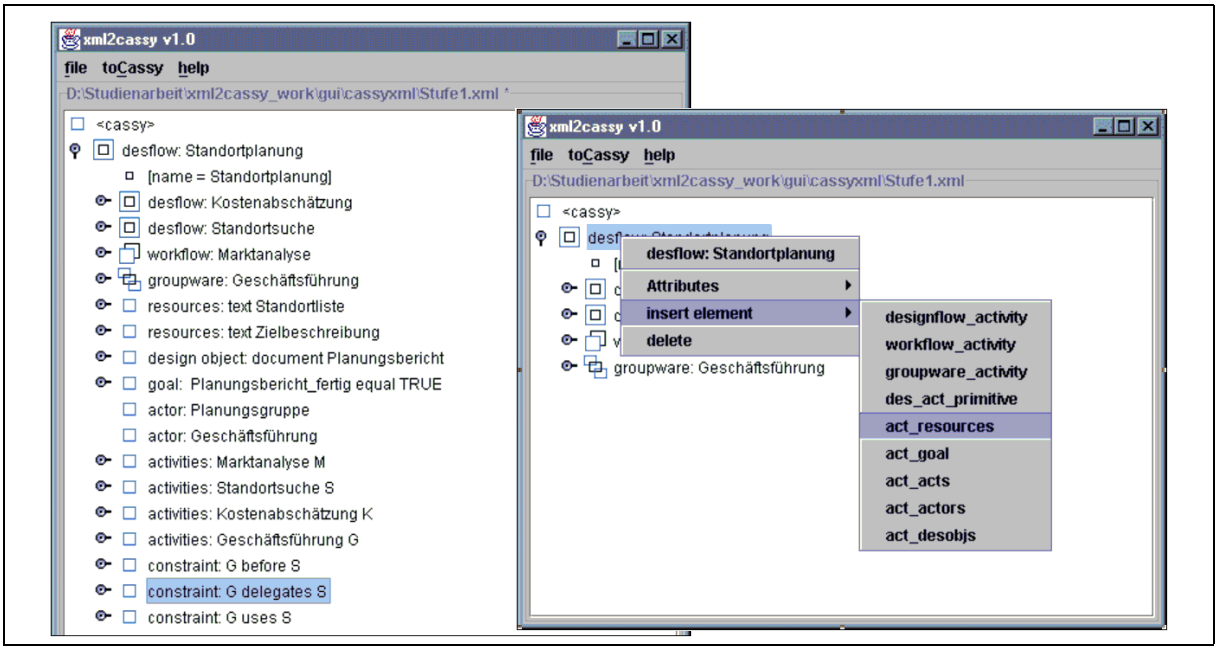


Abbildung 102: Oberfläche des Designflow-Editors

Auf der rechten Seite von Abbildung 102 wird am Beispiel der Aktivität Standortplanung gezeigt, wie eine Änderung der Spezifikation initiiert wird. Ein kontextabhängiges Menü bietet die möglichen Operationen an, wie hier die Belegung von Attributen, das Einfügen weiterer Elemente oder das Löschen des Elements. Diese Menüs sind abhängig von dem Element auf das sie angewendet werden, so besteht bei einem Attribut lediglich die Möglichkeit dessen Wert,

abhängig vom Typ (z.B. integer oder string), zu ändern. Designflows können in einer XML-Datei gespeichert und entsprechend wieder eingeladen werden. Über den Menüpunkt `toCassy` kann eine Designflow-Spezifikation in das System eingebracht werden, d.h. es werden die entsprechenden CORBA-Objekte erzeugt.

5.8.2 Die Konfigurations- und Designflow-Dateien

Die wichtigen Bestandteile und Konfigurationsdateien von xml2cassy sind in Abbildung 103 dargestellt. Als XML-Parser wurde xml4j von IBM gewählt, da dieser den Zugriff auf den erzeugten DOM-Baum erlaubt. Das DOM ist ein vom W3C-Konsortium ins Leben gerufener generischer Standard zur Repräsentation von Dokumenten und deren Manipulation. Das DOM repräsentiert ein Dokument mittels einem Dokument-Objekt-Baum. Zu diesem Baum können Elemente hinzugefügt oder entfernt werden. Der Inhalt von Elementen kann verändert werden, Attribute können als Attributknoten angehängt, geändert oder gelöscht werden. Dies ermöglicht eine leichte Änderbarkeit der Designflow-Definition ohne jedesmal erneut das ganze Dokument auf Korrektheit überprüfen zu müssen. Dieser Parse-Baum wird auch von dem xml2cassy-Interpreter genutzt um systematisch die CASSY-Objekte zu erzeugen.

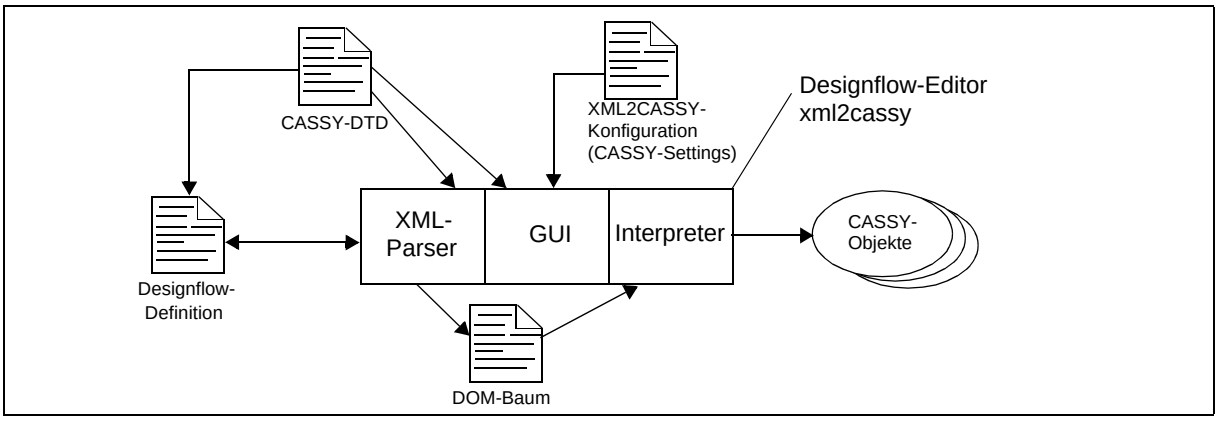


Abbildung 103: Der Designflow-Editor xml2cassy

Der Designflow-Editor benutzt i.w. zwei Dateien um sich zu konfigurieren. Zum einen die Beschreibung der Designflow-Sprache (CASSY-DTD) und zum anderen eine Konfigurationsdatei für die Oberfläche (CASSY-Settings). Ein gekürzte Version der Datei `cassy.dtd` wird in Abbildung 104 gezeigt (hier wird die Definition der Objekte, Programme und Ressourcen nur angedeutet). Diese beschreibt die Elemente und Attribute einer Designflow-Definition. Ein Element kann weitere Unterelemente besitzen. Im Falle einer Designflow-Aktivität sind dies: Designflow-Aktivitäten, Workflow-Aktivitäten, Groupware-Aktivitäten, Ressourcen, Entwurfsobjekte, Entwurfsziel, Akteure, Sub-Aktivitäten und Designflow-Primitive. Die Attribute werden separat definiert. Bei der Designflow-Aktivität ist dies lediglich deren Name.

Die DTD wird von xml2cassy genutzt um die Definition eines Designflows zu überwachen, d.h. die Kontextmenüs erlauben nur eine Eingabe von dort definierten Elementen und Attributen. Letztendlich wird eine in XML gefaßte Designflow-Definition erzeugt, wie in Abbildung 105 dargestellt. Dieses Beispiel entspricht dem Designflow der grafischen Darstellung in Abbildung 102.

Die GUI-Komponente kann zusätzlich bezüglich ihrer Darstellung konfiguriert werden. Das betrifft vor allem die grafischen Symbole, die den Einträgen zugeordnet sind (in Abbildung 106 ist dies zum Beispiel das Bild `groupware.gif`) und die Benennung der Knoten. Diese Definitionen werden in der Datei `cassysettings.xml` abgelegt. Die dazugehörige DTD (`cas-`

```
<?xml version='1.0' encoding='UTF-8' ?>
<!ELEMENT cassy (designflow_activity*, workflow_activity*, groupware_activity*, objects?,
   programs?, resources?) >
<!ELEMENT objects ...>
<!ELEMENT programs ...>
<!ELEMENT resources ...>
<!ELEMENT designflow_activity (designflow_activity*, workflow_activity*, groupware_activity* ,
act_resources*, act_desobjs*, act_goal, act_actors*, act_acts*, des_act_primitive*)>
<!ATTLIST designflow_activity name CDATA "">
<!ELEMENT workflow_activity (designflow_activity*, workflow_activity*, groupware_activity*,
act_resources*, act_desobjs*, act_goal, act_actors*, act_acts*, wf_act_definition*)>
<!ATTLIST workflow_activity name CDATA "">
<!ELEMENT groupware_activity (act_resources*, act_desobjs*, act_goal, act_actors*,
gw_act_primitive*) >
<!ATTLIST groupware_activity name CDATA "">
<!ELEMENT act resources(#PCDATA)>
<!ATTLIST act_resources type (int|text) "int">
<!ELEMENT act_desobjs (#PCDATA)>
<!ATTLIST act_desobjs type (document) "document">
<!ELEMENT act_goal EMPTY>
<!ATTLIST act_goal description CDATA "" object CDATA ""
         operator (equal|lessequal|less|greaterequal|greater) "equal" value CDATA "">
<!ELEMENT act_acts EMPTY>
<!ATTLIST act_acts type CDATA "" name CDATA "">
<!ELEMENT act_actors (#PCDATA)>
<!ELEMENT des_act_primitive EMPTY>
<!ATTLIST des_act_primitive object1 CDATA "" constraint (before|uses|delegates|after) "before"
         object2 CDATA "">
<!ELEMENT wf_act_definition EMPTY>
<!ATTLIST wf_act_definition wf_def_name CDATA "" wf_server_name CDATA "">
<!ELEMENT gw_act_primitive EMPTY>
<!ATTLIST gw_act_primitive type (bbs|teleconference|shared_editing) "bbs" name CDATA "">
```

Abbildung 104: CASSY DTD

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE cassy SYSTEM 'cassy.dtd'>
<cassy>
   <designflow_activity name="Standortplanung">
      <designflow_activity name="Kostenabsch&auml;tzung"></designflow_activity>
      <designflow_activity name="Standortsuche"></designflow_activity>
      <workflow_activity name="Marktanalyse"></workflow_activity>
      <groupware_activity name="Gesch&auml;ftsf&uuml;hrung"></groupware_activity>
      <act_resources type="text">Zielbeschreibung</act_resources>
      <act_resources type="text">Standortliste</act_resources>
      <act_desobjs type="document">Planungsbericht</act_desobjs>
      <act_goal operator="equal" object="Planungsbericht_fertig" description=""
         value="TRUE"></act_goal>
      <act_actors>Planungsgruppe</act_actors>
      <act_actors>Gesch&auml;ftsf&uuml;hrung</act_actors>
      <act_acts name="M" type="MARKTANALYSE"></act_acts>
      <act_acts name="S" type="STANDORTSUCHE"></act_acts>
      <act_acts name="K" type="KOSTENABSCH&Auml;TZUNG"></act_acts>
      <act_acts name="G" type="GESCH&Auml;FTSF&Uuml;HRUNG"></act_acts>
      <des_act_primitive object2="M,S,K" object1="G" constraint="before"></des_act_primitive>
      <des_act_primitive object2="M,S,K" object1="G" constraint="delegates">
         </des_act_primitive>
      <des_act_primitive object2="Standortliste" object1="G" constraint="uses">
         </des_act_primitive>
      <des_act_primitive object2="Standortliste" object1="K" constraint="uses">
         </des_act_primitive>
     </designflow_activity>
</cassy>
```

Abbildung 105: Definition eines Designflows in XML (Auszug)

`sysettings.dtd`) muß entsprechend der DTD der Designflow-Sprache (`cassy.dtd`) angepaßt sein, damit wie im Beispiel zu sehen, `groupware_activity` auch zugeordnet werden kann.

```
<elementdescription name="groupware_activity">
   <icon name="./images/groupware.gif"></icon>
   <view type="treeview">
      <text value="groupware: "></text>
      <attribute value="name"></attribute>
   </view>
</elementdescription>
```

Abbildung 106: Elementkonfiguration für Groupware-Aktivitäten

Ein vollständiger Designflow kann nun in CASSY eingebracht werden. Dafür wird der Parse-Baum traversiert und die entsprechenden CASSY-Objekte werden instantiiert und bei dem Aktivitäten-Management angemeldet. Dafür werden die Objekt-Factories der Aktivitäten genutzt. Die Objekte des Informationsraums müssen jedoch bereits existieren und werden nicht von xml2cassy erzeugt. Dies ist die Aufgabe spezieller Werkzeuge, die von den zu integrierenden Systemen abhängig sind (z.B. Datenbank, Workflow-System usw.).

5.8.3 Aktivitäten-Management und Event Engine

Im vorherigen Abschnitt wurde beschrieben, wie ein Designflow erstellt und in das System eingebracht werden kann. Hier werden wir kurz zeigen, daß einzelne Aktivitäten hinzugefügt werden können und wie die Behandlung und Ausführung von Aktivitätszuständen erfolgt. In Abbildung 107 wird ein einfacher CASSY-Client vorgestellt. Zuerst erstellen wir eine neue primitive Aktivität `prim1` in dem Dialog “Create New Activity”, die durch die CORBA-Factory `PrimitivieActivityObjectFactory` zu erzeugen ist. Diese wird instantiiert und in das System eingetragen. Im “CASSY Test Client” erscheint die Aktivität, wobei deren ID, IOR und Akteur rechts angezeigt werden. Zur Steuerung der Aktivität existieren die Schaltflächen init, start, stop, resume, terminate und cancel, welche die Zustandsänderungen der Aktivität auslösen, worauf wiederum die entsprechenden Ereignisse der Event Engine mitgeteilt werden. Bedienbar sind jeweils nur die von der Event Engine zugelassenen Übergänge bzw. eigentlich die daraus resultierenden Ereignisse.

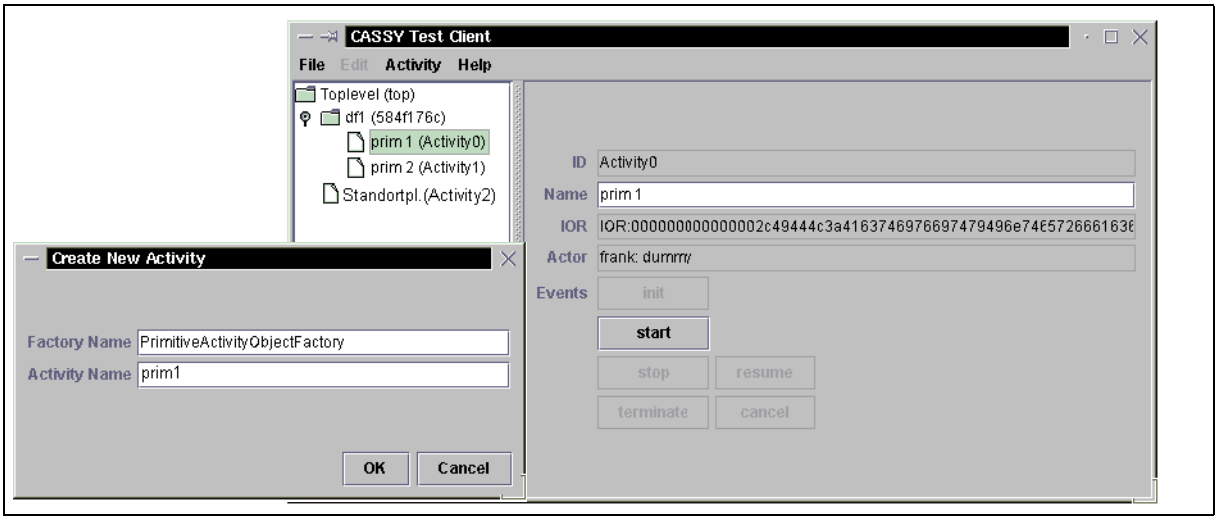


Abbildung 107: Instantiierung und Ausführung einer Aktivität

Wird nun die Schaltfläche “start” betätigt, wird dies an das Aktivitätsobjekt weitergereicht, das alle notwendigen Schritte für die Vorbereitung einer Bearbeitung durchführt (vgl. Abschnitt 4.4.3). Zum Beispiel wird die Nutzungsbeziehung für benötigte Entwurfsobjekte etabliert. Im Falle einer interaktiven primitiven Aktivität wird ein Programm oder Werkzeug gestartet, im Falle einer Workflow- oder Groupware-Aktivität wird die entsprechende Facility aufgerufen.

5.8.4 Ein Beispiel für eine Monitoring-Komponente

In [HÜBNER 1998] wird das System GRiPS beschrieben, welches die Planung komplexer Entwicklungsprozesse unterstützt. Der gewählte Ansatz stammt von der Netzplantechnik (vgl. Abschnitt 3.7.3) ab und wurde um die dezentrale Planung und Koordination in Entwicklungsprozessen erweitert. Eine wesentliche Neuerung ist die Unterstützung des Multi-User-Betriebs, der die Realisierung von Rechten zur Änderung, organisationsabhängige Transparenz und eine Awareness-Funktionalität beinhaltet. Der letzte Punkt ist für uns besonders interessant, da dies auch gewinnbringend für CASSY verwendet werden kann.

Die Netzplantechnik wird in der Projektplanung eingesetzt, um einzelne Vorgänge, deren Abhängigkeiten und Ecktermine zu definieren. Der gesamte Prozeß wird zunächst in einzelne Phasen aufgeteilt. Die Ecktermine trennen diese Phasen und beschreiben die jeweils zu erzielenden Ergebnisse. Die Phasen selbst werden rekursiv in Subphasen aufgeteilt, die ihrerseits mit einem zu erreichenden Meilenstein beschrieben werden. Danach erfolgt die Feinplanung, welche die einzelnen Aktivitäten in den Phasen beschreibt. Daher läßt sich auch eine Analogie zum ADM herstellen, speziell zu der Verfeinerung von Aktivitäten durch Hierarchisierung und Delegation. Auch die Zielbeschreibungen der einzelnen Aktivitäten erlauben eine ähnliche Semantik.

Ein solchermaßen entstandener Prozeßplan wird häufig grafisch in Form eines Netzplans beschrieben, der auch die Anordnungsbeziehungen zwischen Phasen und Aktivitäten berücksichtigt. Üblicherweise werden folgende Beziehungen zwischen zwei Vorgängen verwendet:

- Ende/Anfang: Entspricht der Sequenz (im ADM: Constraint *B AFTER A*).
- Anfang/Ende: Der zweite Vorgang muß beendet sein, bevor der erste anfängt (im ADM: Constraint *A BEFORE B*).
- Anfang/Anfang: Der zweite Vorgang darf erst anfangen, wenn der erste angefangen hat (im ADM entspricht dies dem Ausdruck *s(A) - s(B)*).
- Ende/Ende: Der erste Vorgang muß beendet sein, bevor der zweite beendet ist (im ADM: *(t(A) - t(B)) AND (s(A))*).

Vorgänge werden mittels Balken dargestellt, die entlang einer Zeitachse liegen. Die Beziehungen werden grafisch durch eine Linie mit Pfeil ausgedrückt. Diese Linie setzt bei dem ersten Vorgang an und führt zu dem nächsten. Durch die Darstellung der Zeitdauer und Ausführungsbeziehungen können Konflikte zwischen den Vorgängen einfach erkannt werden. Bei Projekten, die verschiedene Spezialisten-Teams umfassen, die über Abteilungs- oder sogar Unternehmensgrenzen hinweggehen, wird die Planung verteilt durchgeführt. So werden typischerweise die Ecktermine durch die Projektleitung vorgegeben, die Feinplanung erfolgt jedoch durch die einzelnen Teams [STUFFER 1993].

Um dies geeignet zu unterstützen, muß ein entsprechendes Planungssystem Konflikte erkennen, speziell solche, die durch die Abhängigkeiten der Ergebnisse verschiedener Teams entstehen. Da die Beziehungen in der Netzplantechnik bindend sind, werden alle Pläne “korrektgerechnet”. Im Falle der dezentralen Planung ist jedoch die Einführung sogenannter *weicher Verknüpfungen* notwendig, um die Entstehung von Konflikten zuzulassen, was auch der verteilten und lose gekoppelten Infrastruktur von GRiPS entgegenkommt [HÜBNER 1998][KLEEDÖRFER 1998]. Dadurch wird Awareness auf Benutzerseite bezüglich der Planungskonflikte erreicht. Es obliegt dann den menschlichen Akteuren mit diesem Wissen Änderungen zur Erreichung eines korrekten Plans durchzuführen. Konflikte sind allen beteiligten Parteien anzuzeigen, da ein Team nur das Recht besitzt seine eigene Teilplanung zu ändern. Diese Problematik wird an einem einfachen Beispiel in Abbildung 108 veranschaulicht. Hier wird von Team 2 der Balken V2.2 verlängert, wodurch die Anfang/Ende-Beziehung zu dem Vorgang V1.2 von Team 1 nicht

erfüllbar ist. Dieser Konflikt wird durch das System erkannt und weitergemeldet. Nun liegt es in der Hand der beteiligten Personen eine Lösung herbeizuführen. In dem Beispiel wurde der Ecktermin verlegt und somit Raum für die Verlegung von V1.2 geschaffen.

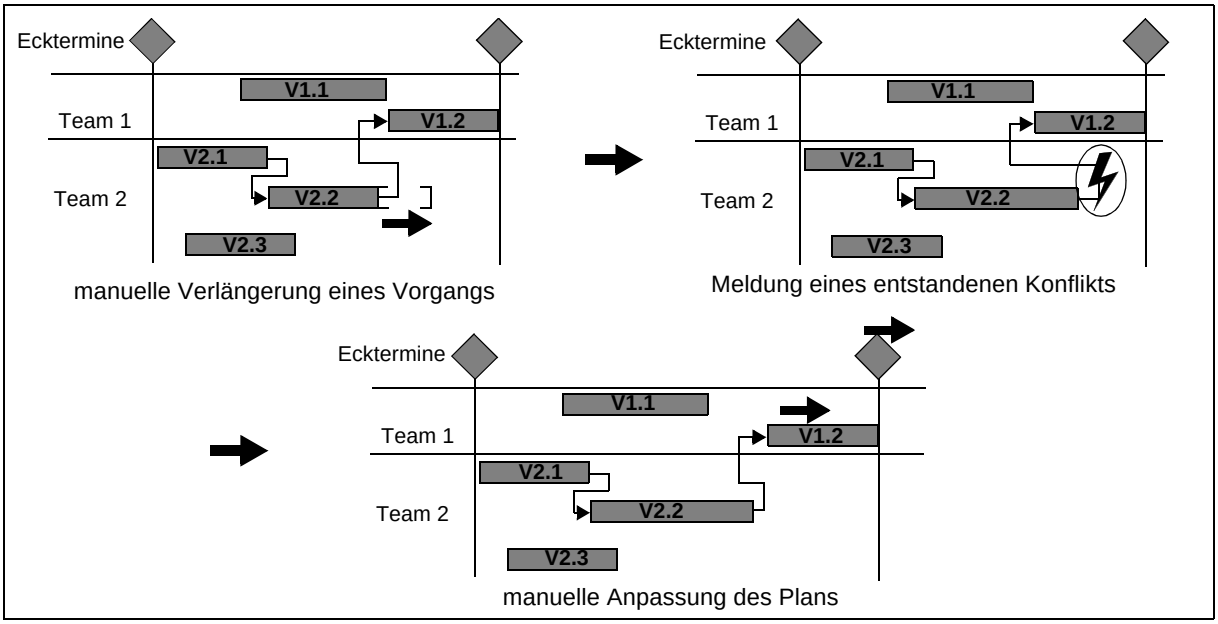


Abbildung 108: Automatische Erkennung von Konflikten in Prozessen

Einen weiteren interessanten Aspekt den GRiPS (siehe Abbildung 109) realisiert, ist die Benutzung von sogenannten *Sammelvorgängen*. Diese fassen eine Menge einzelner Vorgänge zusammen, um eine übersichtliche Darstellung eines Prozesses zu gewinnen. Dies korrespondiert in etwa mit dem Konzept von hierarchischen Aktivitäten, die entweder mit enthaltenen Aktivitäten oder als einzelne Aktivität dargestellt werden können.

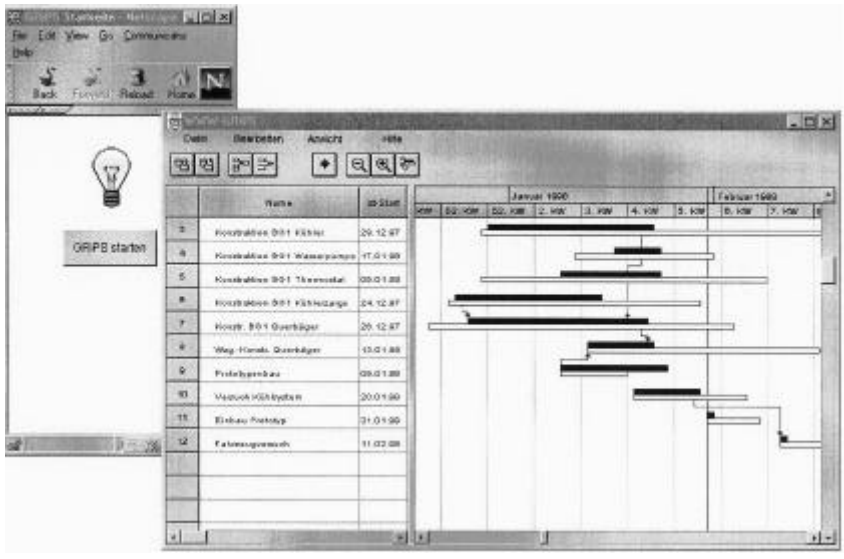


Abbildung 109: GRiPS Benutzungsoberfläche (Java)

Dieser Ansatz kann gewinnbringend für CASSY eingesetzt werden, um eine übergeordnete bzw. zusammenfassende Darstellung eines Entwurfsprozesses und die vorhandenen Abhängigkeiten anzuzeigen. Die Funktionalität zur Erkennung von Konflikten zur Laufzeit von Aktivitäten und deren Abhängigkeiten kann für ADM-Constraints (s. Abschnitt 4.4.4) umgesetzt werden. Dafür eignen sich die Constraints, welche sich direkt auf die Vorgangsbeziehungen (s. o.) abbilden lassen (z.B. BEFORE, AFTER). Leider wurde diese Anbindung noch nicht reali-

siert. Da die Schnittstelle von GRiPS auf JDBC basiert, bietet sich hier die Möglichkeit entsprechende Daten automatisch zu erzeugen und diese über JDBC einzulesen. Somit ist es möglich, eine mächtige Monitoring-Komponente in CASSY einzubringen.

5.8.5 Die Durchführung von Interaktionen

In diesem Abschnitt beschreiben wir die Durchführung des Verhandlungsprotokolls und zeigen dabei auch die Möglichkeiten zur Einbindung von Benutzern. In Abbildung 110 werden einige Screenshots der Kommunikations-Clients dargestellt. Das Hauptfenster zeigt den Graphen des instantiierten Protokolls an. Der aktuelle Zustand ist stark dunkelgrau gekennzeichnet. Zwischenzustände sind weiß, Anfangszustände hellgrau und Endzustände mittelgrau. Wartende Zustände sind mit einem führenden “O” (observer), sendende mit einem führenden “S” (sender) und Endzustände mit einem führenden “F” (final) gekennzeichnet. Die Übergänge sind mit “S (*performative*)” für Sende-Transitionen und “E (*performative*)” für Empfangs-Transitionen markiert. Für das dargestellte Verhandlungsprotokoll existieren folgende Performatives: PRO: propose, REF: refine, REJ: reject, ACP: accept.

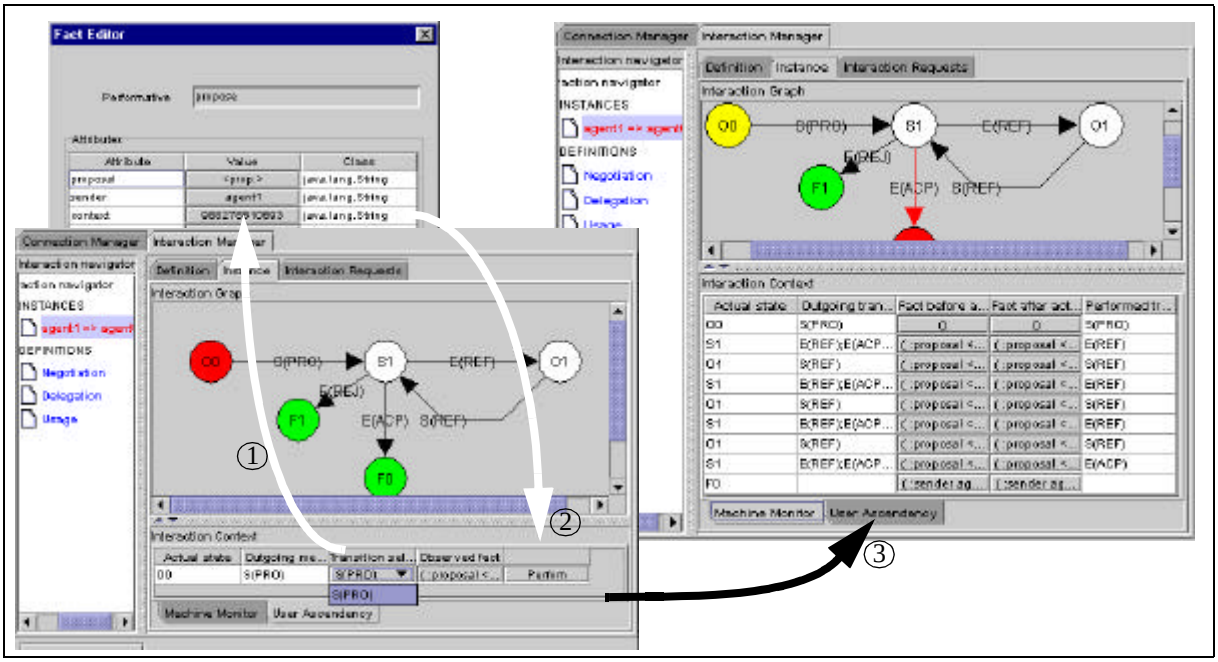


Abbildung 110: Durchführung des Verhandlungsprotokolls

Die Client-Oberfläche unten links in Abbildung 110 zeigt den Beginn eines Verhandlungsprotokolls an. Der Benutzer kann dabei die vom Ausgangszustand möglichen Transitionen auswählen (hier: S(PRO)). Mit dem sogenannten Fakten-Editor (in der Abbildung oben links, (1)) können die variablen Teile einer KQML-Nachricht geändert werden, die durch die spitzen Klammern gekennzeichnet sind. Wurde die zu sendende Nachricht bearbeitet, kann diese verschickt werden. Wie in der Abbildung unten links zu sehen ist, besteht nun die Möglichkeit, den gewählten Übergang über den Button ’*Perform*’ zu schalten (2). Dies führt zum Empfang der Nachricht bei dem Kommunikationspartner (3), wo auch der entsprechende Übergang geschaltet wird. Der Client oben rechts in Abbildung 110 zeigt das Ende der Verhandlung und die dabei aufgetretene Kommunikation an, die durch eine Iteration von *refine*-Nachrichten gekennzeichnet ist.

An diesem Beispiel ist zu sehen, wie unproblematisch die Einbindung von Benutzern möglich ist. In einer Weiterentwicklung sollten natürlich Beschreibungen zu den Übergängen existieren, die es einem menschlichen Benutzer erleichtern, den richtigen Übergang auszuwählen und ein-

gehende Nachrichten zu interpretieren. Dadurch, daß die Protokolle Aktionen in Entitäten auslösen und die Protokolle konsistent zur Semantik der Entwurfsanwendung spezifiziert sein müssen, werden alle Resultate einer Interaktion systemüberwacht ausgeführt. Wir haben also die geforderte umfassende Systemunterstützung erreicht, die sich nicht nur auf die korrekte Ablaufausführung der Entwurfsschritte, sondern auch auf konsistente Durchführung der Aktionen bezieht.

5.9 Zusammenfassung der Designflow-Schicht

Wir haben nun die wesentlichen Bestandteile von CASSY vorgestellt. Die Designflow-Schicht setzt sich aus dem Aktivitäten-Management, dem Informationsraum und der Protokollschicht zusammen. Um einen Designflow durchzuführen ist eine geeignete Interaktion der einzelnen Komponenten notwendig. Hier stellen die Interaktionsprotokolle ein verbindendes Element dar, wie in Abbildung 111 illustriert. Durch Protokolle wie die Delegation werden Akteure, Aktivitäten und Entwurfsobjekte in den Entwurf eingebunden. Die Zugriffs- und Nutzungsprotokolle realisieren den gemeinsamen Informationsraum und damit das Zusammenspiel zwischen Aktivitäten und Objekten. Und schließlich können Verhandlungsprotokolle zwischen allen Entitäten des ADM genutzt werden.

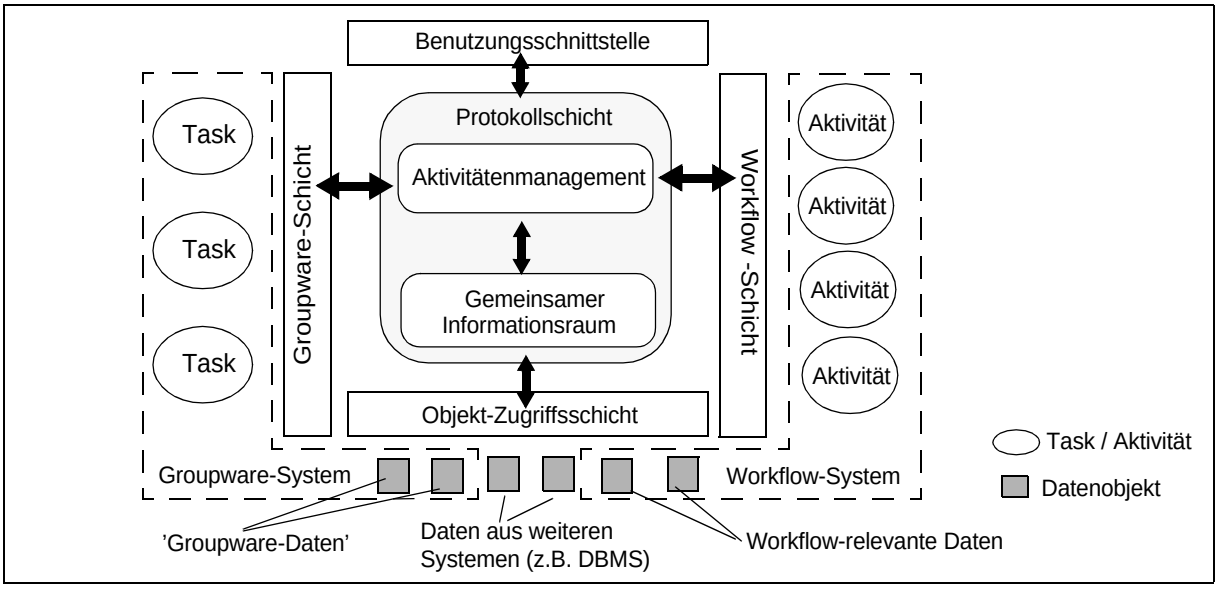


Abbildung 111: Die Verbindung von Informationsraum und Aktivitäten-Management mittels der Protokollschicht

Ein weiteres verbindendes Merkmal ist das Aktivitäten-Management selbst und darin die Event Engine, welche die Strukturierung von Arbeitsschritten und die Beschreibung deren Abhängigkeiten erlaubt. Wir werden diese Aspekte in den folgenden Abschnitten zusammenfassen. Bevor wir in Abschnitt 5.10 einige beispielhafte Aspekte bei einer Designflow-Durchführung aufgreifen.

5.9.1 Das Aktivitäten-Management

Die Verwaltung der in einem Designflow vorhandenen Aktivitäten stellt sich zunächst sehr einfach dar, weil diese lediglich in einem CORBA-Name-Service eingetragen werden und die Überwachung der Ausführung durch den Aktivitäten-Manager zusammen mit der Event Engine geleistet wird (siehe Abschnitt 5.4). Die Designflow-Definition selbst wird durch den Designflow-Editor verwaltet. Dieser erzeugt auch die Aktivitätsobjekte zum Start eines Designflows

(siehe Abschnitt 5.8.1). Aktivitäten, die zur Laufzeit eines Designflows gestartet werden sollen, werden durch einfache Instantiierungen von Aktivitätsobjekten eingebracht. Durch die Event Engine werden die ggf. neu hinzugekommenen Constraints überprüft.

5.9.2 Der Informationsraum

Auch der Informationsraum wurde weitgehend dezentral aufgebaut. Die nutzbaren Objekte (`BasicResourceAdapter`-Interface von CASSY und abgeleitete Interfaces) werden - wie auch die Aktivitäten - in einen weiteren Bereich des Name-Service eingetragen. Somit stellt der Name-Service das Verzeichnis des Informationsraums dar.

Im Gegensatz zu der Implementierung der Groupware-Facility und speziell der dort eingeführten Workspaces, wird der Zugriff auf die Objekte nicht durch eine übergreifende Instanz geregelt, sondern durch die Implementierung der einzelnen Objekte. Die Gründe für diese Entscheidung wurden bereits in Abschnitt 4.6 ausgiebig diskutiert. Kurz zusammengefaßt sind dies unterschiedliche Zugriffsprotokolle für verschiedene Objekttypen und die Unterstützung der Delegationsbeziehung und von Objektabhängigkeiten. Die Strukturierung über Objektabhängigkeiten erlaubt es, komplexe Entwurfsobjekte zu repräsentieren und deren Abhängigkeiten bei der Bearbeitung miteinzubeziehen. Damit ist der Informationsraum die Basis für die Kooperation auf gemeinsamen Daten. Die Akteure können über sie betreffende Änderungen benachrichtigt werden und entsprechend eingreifen. Hierzu dienen letztlich die Interaktionsprotokolle, wie im folgenden Abschnitt beschrieben.

5.9.3 Interaktionsprotokolle

Die Interaktionsprotokolle liegen als einzelne Java-Klassen vor und können dynamisch geladen werden. Dafür ist es derzeit notwendig, daß diese auf den jeweiligen Rechnern vorliegen, um sie dort bei Bedarf laden zu können. Es ist aber auch möglich Java-Klassen über Netzwerk-Protokolle zu senden: dies ist jedoch noch nicht in CASSY realisiert. Eine Protokollmaschine kennt die von ihr verstandenen Protokolle und lädt diese ein, falls sie noch nicht vorhanden sind.

Durch diese Realisierung ist die Durchführung verschiedener an die Anwendung angepasster Protokolle einfach durchführbar. Wie oben beschrieben bilden die Protokolle einen integrativen Bestandteil für das ADM. Besonders hervorzuheben ist die Kooperationsfunktionalität im Zusammenhang mit den Objekten im Informationsraum, die CASSY als CSCW-System qualifiziert, das gleichwertig sowohl die Koordination des Aktivitätenmodells als auch die Kooperation von Akteuren unterstützt. Aktionen, die im Rahmen dieser Kooperationen ausgeführt werden, unterliegen somit der Systemkontrolle.

5.10 Durchführung eines Designflows

In diesem Abschnitt wird ein Einblick in das Laufzeitverhalten von CASSY gegeben. Anhand des Standortplanungsbeispiel aus Kapitel 4 besprechen wir einige interessante Aspekte der Ausführung eines Designflows.

Die Definition des Designflows Standortplanung wurde bereits in Kapitel 4 gezeigt. Mit dem vorgestellten Designflow-Editor kann diese erstellt und in das System eingebracht werden (siehe Abschnitt 5.8.2). Der Designflow-Editor erzeugt dafür einen Aktivitätenbaum. Die hierarchischen Beziehungen der Struktur werden zum einen durch die direkten Referenzen in den Aktivitätsobjekten dargestellt und zum anderen in den CORBA Name-Service eingetragen.

Die Ressourcen werden nicht von dem Designflow-Editor erzeugt. Diese sind entsprechend einzubringen, damit diese auch von den Aktivitäten genutzt werden können. Wir betrachten für den Designflow Standortplanung nun die folgenden Objekte:

- Standortbericht
  Dies ist ein Text-Dokument, das lediglich durch seinen Pfad als einziges Attribut spezifiziert ist (siehe Abschnitt 4.5).
- Standortliste und Standorteinträge
  Die Standortliste wird durch eine Datenbanktabelle realisiert, deren Tupel die Einträge für die einzelnen Standorte sind (siehe Abschnitt 4.6.4). Die Attribute dieser Tabelle sind Arealname, Ort, PLZ, Straße, Ansprechpartner, Preis, Beschreibung, Verzeichnis erstellter Dateien (z.B. Verträge) usw.

Der Standortbericht wird durch eine einfache Spezialisierung von `BasicResourceAdapter` realisiert. Es wird lediglich der Pfadname für das Dokument verwaltet. Die Standortliste kann man durch eine generische DB-Ressource ausdrücken oder wie in unserem Fall durch die Spezialisierung `DBTableStandortliste`, welche eine Suchfunktion über die einzelnen Einträge sowie eine Erzeugungsfunktion für neue Einträge anbietet. Die Einträge sind ebenfalls ADM-Objekte, die Funktionen zur Abfrage und Änderung der einzelnen Attribute realisieren. Diese Ressourcen werden als CORBA-Objekte instantiiert. Änderungen und Abfragen der Daten werden in entsprechende SQL-Ausdrücke umgesetzt und an die Datenbank weitergeleitet. Der Designflow kann die Einträge nun anhand deren textueller ID (Arealname) referenzieren. Wie in Abschnitt 4.6.4 beschrieben, sind die Standorteinträge über ein Dependency-Objekt mit der Standortliste verbunden.

Wird nun der Designflow Standortsuche zur Bewertung verschiedener in Frage kommender Areale gestartet, so wird sich dieser auf das Entwurfsobjekt Planungsbericht registrieren. Dazu wird der Name Service nach der entsprechenden ID “Standortbericht” durchsucht und die resultierende CORBA-Referenz extrahiert. Nun kann der Designflow die Nutzungsanfrage (use) starten, wobei die CORBA-ID der Ressource das Empfängerobjekt der Nutzungskonversation spezifiziert. Analog bauen die verschiedenen Bewertungsaktivitäten Nutzungsbeziehungen mit den einzelnen Standorteinträgen auf. Jetzt können die Objekte auch konkurrierend genutzt werden. In Abschnitt 4.6.4 wurde bereits ein Beispiel für die Anwendung solcher konkurrierender Objektzugriffe gezeigt.

Wird ein neuer Standort gefunden, so ist auch der entsprechende Eintrag zu erzeugen. Dies geschieht über die `create`-Operation des Standortlisten-Objekts, das in diesem Fall auch als Factory für Einträge fungiert.

Ist nun der Bearbeiter einer Bewertung an den Resultaten anderer Bewertungsaktivitäten interessiert, so kann er entweder diese nach einer erfolgten Registrierung mit Lesezugriff direkt abfragen oder aber über die gesamte Standortliste (d.h. das Objekt, das die DB-Tabelle repräsentiert) eine Abfrage starten. Dadurch ist ein Vergleich, etwa für Verhandlungen mit Maklern oder Gemeinden, realisierbar. Wurde ein Standort als vielsversprechend erkannt, so kann er durch den Designflow Kostenabschätzung bewertet werden. Wurde die Abschätzung geändert, so wird auch die entsprechende Bewertungsaktivität notifiziert. Die Aktivität Listenerstellung wird gestartet, wenn eine Änderung eines Eintrags erfolgt. Die Listenerstellung ruft ein Programm auf (REPGEN.EXE, siehe Abschnitt 4.5), das alle Einträge der Standortliste traversiert und in einem Dokument zusammenfaßt. Dieses Dokument dient der Geschäftsführung zur Begutachtung des Fortschritts der Standortbewertungen.

5.11 Fazit

Wie bereits an obigem Beispiel zu sehen ist, wird die Durchführung der von uns geforderten Funktionalität zur Unterstützung von Entwurfsprozessen weitgehend unterstützt. Die verschiedenen Mechanismen und Technologien ergänzen sich und erlauben die Beschreibung strukturierter Elemente im Prozeß sowie die Durchführung verschiedener Kooperationsmuster:

- Ablaufunterstützung, welche die einzelnen Arbeitsschritte vorantreibt und damit den Gesamtablauf beherrschbar macht,
- Delegationsunterstützung, die sich auf das Aktivitätsmodell und das Entwurfsobjektmodell auswirkt, da auf der einen Seite die Abläufe organisiert werden müssen und auf der anderen Seite ein entsprechendes Datenmodell zur Modellierung der Zerlegung eines Entwurfsobjekts bestehen muß,
- Regelung gleichzeitiger Arbeiten, wie zum Beispiel gemeinsame Datennutzung und der frühzeitige Austausch von Arbeitsergebnissen,
- direkte Kommunikations- und Kooperationsunterstützung für die Abstimmung der beteiligten Personen und
- integrative Bestandteile zur Einbindung von existierenden Systemen und Werkzeugen.

Die technische Umsetzung war in einigen Bereichen sehr aufwendig wie z.B. die Entwicklung der umfassenden Groupware-Facility. Der Einsatz von CORBA hat sich jedoch bewährt, da eine Modularisierung einfach erreicht wurde, insbesondere durch die entsprechende Spezifikation von Interfaces. Damit konnte auch ein Wrapping erfolgen. Desweiteren erlaubt die objektorientierte IDL eine gut geeignete Methodik, um durch abgeleitete Klassen einen hohen Anpassungsgrad zu erreichen.

Es soll aber nicht verschwiegen werden, daß CASSY lediglich einen Prototypen aus einem zeitlich und insbesondere personell begrenzten Projekt darstellt. So konnten einige Aspekte nicht oder nur teilweise umgesetzt werden, wie etwa die durchgängige Behandlung der Akteurszuordnung, insbesondere bezüglich deren Verwendung als Ressourcen im *X_NEEDS*-Constraint oder eine weitgehend automatisierte Objekterzeugung.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde das ASCEND Designflow Model (ADM) zur Unterstützung kooperativer und koordinativer Aspekte kooperativer Prozesse, die wir als Entwurfprozesse bezeichnen, vorgestellt. Der Kern dieses Modells ist die Fokussierung auf die kooperative Nutzung gemeinsamer Daten in Verbindung mit einem Aktivitätenmodell. Die Konzeption konnte durch die geschickte Integration von Technologien vor allem aus den Bereichen Groupware, Workflow-Management und Interaktionsprotokolle bewerkstelligt und prototypisch realisiert werden.

6.1 Überblick

Dieses Kapitel faßt die Ergebnisse der vorliegenden Arbeit zusammen. Dazu werden in Abschnitt 6.2 die wesentlichen Aspekte des ADM und dessen Implementierung präsentiert und anschließend gegenüber existierenden Ansätzen in Abschnitt 6.3 abgegrenzt. Abschließend folgt in Abschnitt 6.4 ein Ausblick auf weitergehende Forschungs- oder Implementierungsarbeiten.

6.2 Zusammenfassung

Es wurde eine neue Klasse von kooperativen Prozessen bestimmt und durch Beispiele betrachtet, deren Unterstützung durch das ADM erfolgen soll. Diesen Prozessen ist der Bedarf nach Interaktion, Kooperation, kooperativer Nutzung gemeinsamer Ressourcen, Delegation von Teilarbeiten, strukturierten und weniger strukturierten Teilprozessen, Integration von Arbeitsergebnissen und Abstimmung von Aktionen gemein. Diese Bedürfnisse wurden an den drei Szenarien Chipentwurf, Software-Engineering und Standortplanung beispielhaft dargestellt und belegt. Daraus wurde die Forderung an eine geeignete Benutzerunterstützung abgeleitet, die anstatt einer festen Vorgehensbeschreibung den Nutzern die geeignete Unterstützung in Form entsprechend konfigurierbarer Dienste zur Verfügung stellt. Dies schließt auch, wo es geeignet ist, die Durchführung wohl-strukturierter Teilprozesse mit ein.

Danach wurden Technologien vorgestellt und bewertet, die Teile der aufgestellten Forderungen erfüllen können. Der Schwerpunkt dieser Untersuchung betraf CSCW und Workflow-Management, deren explizites Ziel die allgemeine Unterstützung von Arbeit ist. Zwar wird Workflow-Management dem Bereich CSCW zugeordnet, jedoch zeichnet sich das Workflow-Management durch die Fokussierung auf strukturierte Prozesse ohne wesentliche Kooperationsmuster aus. CSCW-Systeme beruhen vor allem auf Kommunikations- und Kooperationstechniken. Eine weitere Klasse von Systemen zur Durchführung von Arbeiten sind CAD-Frameworks, die spezialisierte Dienste für den technischen Entwurf anbieten. Moderne Systeme umfassen die Prozeß-Steuerung und zentrale Datenversorgung. Dies wird auch unter dem Begriff Designflow-Management zusammengefaßt. Für die Realisierung der von uns gewünschten flexiblen Zugriffsregelung wurden außerdem einige Aspekte der Agententechnologie betrachtet, insbesondere Verhandlungsprotokolle. Als weitere interessante Technologien wurde ein Prozeßmodell für das Software-Engineering, der TOGA-Ansatz und die Prozeßplanung mittels einer erweiterten Netzplantechnik untersucht. Aufgrund der so gewonnenen Erkenntnisse wurde ein Lösungsansatz präsentiert, der auf einer geeigneten Integration einer Vielzahl von Technologien basiert.

Dieser Lösungsansatz wird durch das ASCEND Designflow Model (ADM) umgesetzt. Dieses Modell verwendet drei wesentliche Aspekte: ein Aktivitätenmodell, einen Informationsraum und Interaktionsprotokolle. Workflow-Management stellt eine ideale Technologie für die Automatisierung der Steuerung von strukturierten Teilprozessen dar. Das Aktivitätenkonzept ist

eine geeignete Basis zur Repräsentation von abhängigen Arbeitsschritten. Daher wurden diese Konzepte weitgehend in das ADM integriert. Das Aktivitätenkonzept zur Modellierung und Durchführung abgegrenzter Arbeitsschritte hilft die Aufgabenverteilung und Vorgehensweise von Entwurfsprozessen, soweit möglich, zu strukturieren. Bspw. nutzt die Delegations-Beziehung des ADM Aktivitäten zur Spezifikation verschiedener Unteraufträge. Außerdem wurden sog. Workflow-Aktivitäten eingeführt, die alle Eigenschaften eines Workflows übernehmen und innerhalb eines Entwurfsprozesses ausgeführt werden können. Dadurch wurde eine geeignete Unterstützung gut strukturierter Teilprozesse erreicht. Weiterhin wurden primitive Aktivitäten zum Kapseln von Werkzeuganwendungen und Groupware-Aktivitäten zur Durchführung von wenig strukturierten Teilarbeiten eingeführt. Eine Besonderheit stellen die Designflow-Aktivitäten dar, die durch sogenannte Design-Primitive eine erweiterte Funktionalität realisieren. So können anpaßbare Constraints angewendet werden, welche die Abhängigkeiten zwischen den in einer Designflow-Aktivität enthaltenen Ressourcen und Aktivitäten beschreiben. Durch die weitgehende Definierbarkeit solcher Constraints, besteht die Möglichkeit anwendungsspezifische Abhängigkeiten einzuführen und eine flexible Ablaufunterstützung zu erreichen.

Wesentliche Aspekte von Entwurfsprozessen sind der frühe Austausch von gemeinsamen Ergebnissen, die Bearbeitung gemeinsamer Daten und die Abhängigkeiten bezüglich Daten und Ergebnissen, die in verschiedenen Teilprozessen erarbeitet werden. Daher ist eine Abstimmung zwischen den am Prozeß teilnehmenden Personen notwendig. Da eine Ablaufmodellierung zur Repräsentation von Ablaufbeschreibungen nicht ausreichend ist, bzw. zu einschränkend ist, wurde die gemeinsame Nutzung von Ressourcen im Rahmen des gemeinsamen Informationsraums eingeführt. Dadurch können unvorherbestimmte Abläufe über die Objektzugriffe koordiniert werden. Durch die Erweiterung um Objektabhängigkeiten im Informationsraum können auch die Zusammenhänge der Entwurfsobjekte und weiteren Ressourcen modelliert und Änderungen in verbundenen Objekten verfolgt werden.

Der gemeinsame Zugriff erfordert eine weitergehende Unterstützung, angefangen von der Bereitstellung entsprechender zugreifbarer Objekte, der Behandlung von Zugriffsanfragen bis hin zu der Propagierung von Änderungen. Zur Durchführung und Abstimmung der Nutzung werden Protokolle in Konversationsmustern angewendet, die zum einen eine gewisse Weise des Zugriffs vorschreiben, aber auch die Möglichkeit zur Verhandlung anbieten. Diese Verhandlung, wie sie bei konkurrierenden Zugriffen oder bei der Durchführung des Delegationsprotokolls auftreten, stellen ein mächtiges Werkzeug zur Interaktion zwischen allen Entitäten des ADM dar, d.h. zwischen Akteuren, Objekten und Aktivitäten. In den Protokollen werden Aktionen durchgeführt, welche die Operationen der ADM-Entitäten benutzen. Dadurch sind die Effekte der Interaktionen komplett durch das zugrunde liegende System unterstützt und damit ist eine konsistente Behandlung ermöglicht. Die Interaktion wurde von der Objektnutzung bis in das Aktivitätenmodell des ADM erweitert, wie es insbesondere an der Delegation ersichtlich ist. Durch die Einbindung der Akteure wird eine direkte Kooperation erzielt. Die flexible Einsetzbarkeit, die Anpaßbarkeit und die Erweiterbarkeit der Protokolle ermöglicht einen hohen Grad der Anpassung des ADM an verschiedenste kooperative Prozesse.

Im Rahmen der Implementierung wurden die wesentlichen Bestandteile des ADM prototypisch umgesetzt. Es konnte gezeigt werden, daß eine Integration der vorgeschlagenen Technologien erreicht wurde, jedoch teils mit Einschränkungen insbesondere aufgrund gegebener Standards oder verwendeter Systeme. Dies wurde ausführlich am Beispiel der OMG Workflow Facility JFlow in Kombination mit dem am Markt erfolgreich eingesetzten Workflow-Management-System FlowMark beschrieben, etwa bezüglich der Möglichkeiten zur Definition von Workflows oder den typischen Auswirkungen des Wrapping-Ansatzes (persistente vs. Proxy-Objekte).

Damit unterstützt das ADM zum einen Entwurfsprozesse, die teilweise gut strukturiert sind. So können bei Teilprozessen Workflows eingesetzt werden. Zum anderen erlauben die eingeführten Entwurfskonstrukte (bspw. Delegation, Objektzugriffe und Constraints), auch schwächer strukturierte Teilprozesse und damit ein wesentliches Merkmal des Entwurfs zu unterstützen. Genauso ist es möglich, Groupware gewinnbringend für die Teile eines Prozesses einzusetzen, die keine inhärente Struktur mehr besitzen, wobei es zudem möglich ist, über den gemeinsamen Informationsraum zu kooperieren. Somit wird erreicht, daß die passendste, unterstützende Technologie für den jeweiligen Teilprozeß verwendet werden kann. Dadurch werden die verschiedenen Anforderungen bezüglich koordinativer, wie auch kooperativer Zusammenarbeit erfüllt.

6.3 Abgrenzung

Wir haben bereits die wesentlichen Aspekte der Unterstützung von Entwurfs- und Planungsprozessen angesprochen. Wie es bereits an den erläuterten Grundlagen in Kapitel 3 ersichtlich ist, bedient sich der hier vorgestellte Ansatz vieler, verschiedener Technologien. Es wurde auch dargelegt, in welchen Beziehungen die vorgestellten Technologien Schwächen und Unzulänglichkeiten besitzen. Durch die Integration der Stärken einzelner Technologien in dem ADM, wurde eine umfassende Unterstützung der betrachteten Entwurfscharakteristika ermöglicht. In diesem Abschnitt werden wir unseren Ansatz in die existierenden Technologien einordnen und die Besonderheiten unseres Modells hervorheben, die es von den bisherigen Ansätzen unterscheidet.

Im wesentlich handelt es sich bei dem ADM und dessen Implementierung um ein CSCW-System, das ein Kooperationsmodell und ein Koordinationsmodell verbindet. Die Übergänge sind dabei fließend, so dienen die Interaktionsprotokolle sowohl der Koordination, z.B. zur Regelung von Zugriffen, als auch der direkten Kooperation mittels Verhandlungsprotokollen. Durch diese Kombination ist es möglich, alle gewünschten Aktionen innerhalb eines Entwurfsprozesses systemgestützt durchzuführen. Somit wird der Blackbox-Charakter von “klassischen” Workflow-Modellen durchbrochen. Andererseits soll im ADM nicht auf die grossen Vorteile des Workflow-Managements verzichtet werden, weshalb zum einen Workflow-Aktivitäten integriert wurden und zum anderen das Aktivitäten-Management auf Workflow-verwandten Ansätzen beruht.

Durch die Betrachtungen in dieser Arbeit und insbesondere durch die nachfolgende Diskussion in diesem Abschnitt wird deutlich, daß es sich bei dem ASCEND-Modell und dessen Implementierung um ein System zur umfassenden Unterstützung der zuvor beschriebenen kooperativen Prozesse handelt. Existierende Systeme betrachten häufig nur einen Teil der von uns verlangten Funktionalitäten, obwohl die Ansätze von PlanKo (siehe unten) und moderner CAD-Frameworks sehr fortgeschritten sind. ASCEND geht hier jedoch vor allem durch die Realisierung von mächtigen Kooperationsprotokollen darüber hinaus.

Groupware

Die in der Forschung entstandenen CSCW-Systeme betrachten vor allem die direkte und indirekte Kooperation und weniger die Koordination und Strukturierung von Arbeitsschritten. Wie am Beispiel der in Kapitel 2 geschilderten Prozesse aufgezeigt, besteht aber der Bedarf sowohl Koordination als auch Kooperation für eine gesamtheitlich Unterstützung anzubieten. Groupware kann über die drei Mechanismen Koordination, Kooperation und Kommunikation eingeteilt werden, wie in Abschnitt 3.3.7 beschrieben. Gemeinsame Daten sind häufig ein wesentliches Konzept in der Realisierung dieser Systeme. Jedoch bietet

Groupware i.a. keine so ausgereiften und sicheren Aktivitätskonzepte, wie das im Workflow-Management üblich ist. Es fehlt generell die Strukturierungsmöglichkeit für Prozesse. Hier werden im ADM zum einen die Delegation als Strukturierungsmöglichkeit und zum anderen Aktivitäten als Arbeitseinheiten verwendet. Diese Verbindung erlaubt die Realisierung einer Strukturierung von Entwurfs- und Planungsprozessen.

Workflow

Wesentliche Anforderungen von Entwurfsprozessen sind der frühe Austausch von gemeinsamen Ergebnissen, die Bearbeitung gemeinsamer Daten und die Abhängigkeiten bezüglich Daten und Ergebnissen, die in verschiedenen Teilprozessen erarbeitet werden. Daher ist eine Abstimmung zwischen den am Prozeß teilnehmenden Personen notwendig. Eine mögliche Lösung besteht in der Auflösung von Abhängigkeiten durch eine weitgehende 'Sequentialisierung' von Arbeitsschritten und eine entsprechende Datenversorgung der so definierten Aktivitäten, wie es typischerweise durch Workflow-Management möglich ist. Leider ist dies für Entwurfsprozesse nur in einzelnen Teilprozessen eingeschränkt möglich, da der gesamte Prozeß durch seinen ingenieursmäßigen Charakter gewisse Freiheitsgrade in der Steuerung der Prozeß-Schritte verlangt.

Ein weiterer Faktor, der durch existierende Workflow-Systeme nur ungenügend berücksichtigt wird, ist die Behandlung der Arbeitsdaten innerhalb eines Prozesses. Workflow-Aktivitäten, wie sie typischerweise durch ein Workflow-Management-System unterstützt werden [WFMC 1995], unterscheiden sich von den Designflow-Aktivitäten des ADM vor allem durch ihren 'Black-Box-Charakter'. So wird kein allgemeiner Informationsraum verwendet, vielmehr liest eine Workflow-Aktivität zu Beginn ein feste Menge von Eingabedaten und schreibt am Ende eine Menge von Ausgabedaten. Zwischen diesen beiden Zeitpunkten besteht keine systemunterstützte Interaktionsmöglichkeit zwischen Aktivitäten. D.h., daß Kooperationen und deren Subjekte nur außerhalb des Systems verwendet werden und somit bezüglich dieser keine Konsistenz- oder Historienfunktionalität erreicht werden kann. Durch die Festlegung von Ein- und Ausgabedaten einer Aktivität, können systemgestützt keine anderen Daten behandelt werden. Dies vermeidet unser Ansatz explizit über das Konzept des Informationsraums und mittels spezieller Kooperationsfunktionen. Durch die Verwendung von unseren kooperativen Konzepten stellt eine Designflow-Aktivität eine spezialisierte, kooperative Aktivität dar, die anwendungsspezifische und entwurfsspezifische Funktionalität bereitstellt.

Integration von Groupware- und Workflow-Technologie

Die meisten Ansätze für die Integration von Workflow- und CSCW-Systemen beziehen sich auf den Aufruf von CSCW-Anwendungen in Workflow-Aktivitäten. Somit finden die Abläufe eines CSCW-System außerhalb einer gemeinsamen Kontrolle statt und es erfolgt keine Abstimmung bezüglich der Ablaufsynchronisation. Das PlanKo-Projekt [LUDWIG 1995] verwendet hingegen eine Wartebeziehung als zentrales Koordinationskonzept zwischen Workflow- und CSCW-Anwendungen. Die Zustände werden in hierarchischen State Charts verwaltet, wodurch sich mehr Flexibilität in der Ablaufsteuerung zwischen Workflow- und Groupware-Aktivitäten ergibt. Die Kooperationen in PlanKo erfolgen Kooperationsobjekt-bezogen. Damit bestehen einige Parallelen zum ADM, hinsichtlich der Kooperation über gemeinsam genutzten Objekte. Eine Forderung zur Weiterentwicklung des PlanKo-Projekts lautet wie folgt:

"*Man kann sich beispielsweise vorstellen, den Kontrollfluß von Aktivitäten bzw. den Koordinationsmechanismus zur Objektkoordination aus den einzelnen Anwendungen zu separieren und als Basisdienst den einzelnen CSCW-Anwendungen zur Verfügung zu stellen.*" [SCHALLER UND SCHWAB 1997]

Diese Forderung wird umfassend durch das ADM und dessen Implementierung umgesetzt. Damit wurde eine weitgehendere Integration von Workflow- und Groupware-Anwendungen erreicht, die PlanKo nicht in diesem Umfang bieten kann. So fehlt beispielsweise die Interaktion zwischen Konzepten wie Informationsraum und Aktivitäten.

CAD-Frameworks

In Abschnitt 3.6 wurden CAD-Frameworks diskutiert. Das ADM folgt auch einem Framework-Ansatz und unterstützt die dort beschriebenen Dienste, lediglich die Versionierung wurde im ADM nicht berücksichtigt. Hier kann man auf das fortgeschrittene CONCORD-Modell in [RITTER 1997B] verweisen. In diesem Modell wurde auch ein Kooperationsmodell eingeführt (siehe auch Abschnitt 3.6.3), das zum einen auf einem erweiterten Transaktionsansatz basiert, der das Übertragen von Zugriffsrechten auf Daten und dem Übermitteln von Daten innerhalb von ausgezeichneten Kooperationstransaktionen unterstützt. Dieser Ansatz ist gegenüber dem ADM insofern beschränkt, da er sich i.w. auf einfache Datentypen und nicht auf die Operationen spezifischer Datentypen bezieht, wie etwa Transformationen von CAD-Objekten. Zudem wird dadurch lediglich eine indirekte Kooperation erreicht. Im ADM werden die in [RITTER 1997B] vorgeschlagenen Erweiterungen des CONCORD-Modells durch die Einführung einer expliziten Verhandlung und die Integration synchroner Gruppenkommunikation, bereits abgedeckt.

Wie ebenfalls in Abschnitt 3.6.2 beschrieben, bietet das Modell aus [ROLLER UND ECK 1999] eine mächtige Beschreibung von Objekten und ein Kooperationsmodell, das auf dem Erkennen von Konflikten und deren Meldung mittels eines Notifikationsmechanismus basiert. Einfach ausgedrückt, bedeutet dies, daß CONCORD Zugriffskonflikte vorab vermeidet und das Modell von [ROLLER UND ECK 1999] hingegen Konflikte zuläßt, sie jedoch erkennt und eine Auflösung unterstützt. Das ADM erlaubt hier beide Szenarien zu unterstützen. Zum einen kann ein ADM-Objekt ein rechtebasierte Zugriffsverwaltung realisieren und zum anderen mittels geeigneter Protokolle Zugriffskonflikte erkennen und weitermelden. Durch die Schnittstelle der Protokollmaschine ist auch die Einbindung von menschlichen Akteuren in die Konfliktauflösung einfach.

6.4 Ausblick

Weiterführende Arbeiten zur Verfeinerung und Erweiterung des ADM bieten sich in den nachfolgenden Bereichen an.

Verfeinerung von Workflow-Diensten

Zu Beginn des ASCEND-Projekts wurde eine fein-granulare Integration von Workflow- und CSCW-Diensten verfolgt, die eine dynamische Zusammensetzung von ADM-Funktionalitäten erreicht. In Kapitel 3 wurden funktionale Dienste des Workflow-Managements und der Groupware bestimmt und untersucht. In einem ersten Entwurf der Architektur (siehe Abbildung 112) wurden daher entsprechende Dienstgruppen wie Workflow-Enactment oder Gruppenverwaltung extrahiert, die dynamisch über eine CORBA-Middleware kombinierbar sind [FRANK 1997].

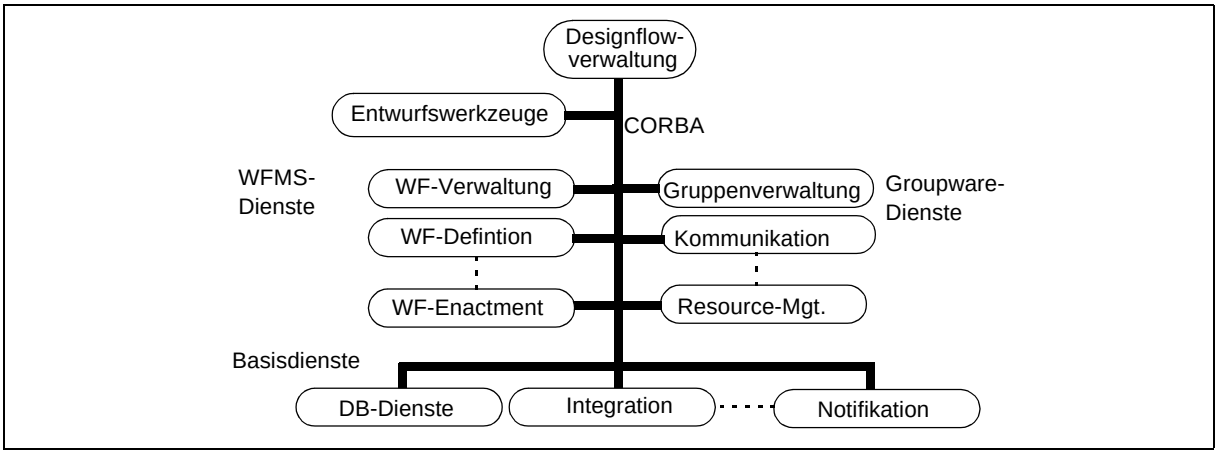


Abbildung 112: Ursprüngliches Dienstekonzept

Dieses Dienstekonzept ermöglicht eine weitestgehende Verzahnung der verschiedenen Technologien. Beispielsweise könnte die Event Engine durch eine geeignete Workflow-Enactment-Komponente ersetzt werden. Zudem war als einzige Aktivität des ADM eine Designflow-Aktivität geplant, die beliebig aus den Primitiven der verschiedenen funktionalen Bereiche definiert werden kann. Da die verschiedenen Dienstgruppen für uns nicht kombinierbar waren, wurden Workflow- und Groupware-Aktivitäten im ADM eingeführt. Diese Vorgehensweise wurde durch folgende Punkte erzwungen:

- monolithische Architektur der meisten Workflow-Systeme,
- nur in begrenztem Umfang von außen zugängliche Funktionalität
- und auch die ’halbherzig’ entworfene JFlow-Schnittstelle der OMG.

Dagegen stellt das Konzept unserer Groupware-Facility ein komponentenorientiertes System dar, das durchaus in eine modulare Architektur übernommen werden kann. Daher sollte in zukünftigen Arbeiten auch untersucht werden, wie ein komponentenbasiertes Workflow-System zu erreichen und entsprechend zu integrieren ist. Dies stellt eine schwieriges Problem dar, weil wegen hohen Anforderungen an die konsistente und fehlertolerante Durchführung von Workflows auch entsprechende Schnittstellen zwischen den einzelnen Komponenten etabliert werden müssen, die eine übergeordnete bzw. durchgängige Durchsetzung der dafür notwendigen Funktionalitäten erlauben, wie z.B. Logging, Recovery, transaktionale Behandlung etc. Der Ansatz der WfMC geht uns hier nicht weit genug, da er zu kompromißgeprägt ist und somit nur eine kleine gemeinsame Untermenge an Funktionalität liefert.

Leistungsaspekte

Der in dieser Arbeit vorgestellte Prototyp wurde genutzt, um die Umsetzbarkeit der vorgestellten Konzepte darzulegen. Die prototypische Implementierung auf der einen Seite und die aufwendige Darstellung der CORBA-Objekte auf der anderen Seite lassen eine schwache Performanz erwarten. Um hier Abhilfe zu schaffen, sind insbesondere folgende Bereiche zu optimieren:

Ein Objekt des Informationsraumes wird derzeit durch ein einzelnes CORBA-Objekt repräsentiert. Abhängig von der Granularität dieser Objekte kann hier ein Flaschenhals entstehen. Als Beispiel mag eine CAD-Datei dienen, die durch die einzelnen Elemente (also Linien, Kreisbögen etc.) in das System eingebracht wird. Zum einen entsteht dadurch eine schwer überschaubare Menge von Objekten, zum anderen müssen häufig Zugriffanfragen abgehandelt werden. Dies kann auf zwei Weisen umgangen werden. Ohne Änderung ist ein geeigneteres Granulat zu

wählen, wie ausführlich in [FRANK ET AL. 2000] beschrieben. Eine zweite Möglichkeit ist, ein einziges Proxy-Objekt für eine Menge gleichartiger Objekte einzuführen, das alle Anfragen erhält und entsprechend behandelt.

Die Protokollmaschine wurde auf Grundlage von JATLite implementiert. Es hat sich gezeigt, daß deren Engine mit zunehmender Anzahl von Protokollteilnehmern ineffizient wird. Es lohnt sich hier also Alternativen zu suchen.

Anwendung auf weitere Szenarien

Wir haben bereits einige Bereiche von Arbeitsformen anhand entsprechender Szenarien beschrieben, die durch das ADM unterstützt werden. Es ist von Interesse, welche weiteren Prozesse sich durch die vorgestellten Konzepte unterstützen lassen. Dazu sind weitere Untersuchungen notwendig.

Literatur

ALONSO 1997

Alonso, G.: Processes + Transactions = Distributed Applications; HPTS'97, Pacific Grove, California, USA,1997

ALONSO ET AL. 1995

Alonso, G., Günthör, R., Kamath, M., Agrawal, D., El Abbadi, A., Mohan, C.: Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System; in Proc. 3rd Intern. Conf. on Cooperative Information Systems, Wien, 1995

ALONSO ET AL. 1997

Alonso, G., Agrawal, D., El Abbadi, A., Mohan, C.: Functionalitiy and Limitations of Current Workflow Management Systems; IEEE-Expert (Spezialausgabe über kooperative Systeme), 1997

ALONSO UND HAGEN 1997

Alonso, G., Hagen, C.: Geo-Opera: Workflow Concepts for Spatial Processes. In: 5th Int. Symp. on Spatial Databases (SSD'97), Berlin, Germany, July 1997

ALMASI ET AL. 1994

G. Almasi et al.: Functional Specifications for Collaborative Services; CERC Technical Report Series, CERC-TR-TM-94-002, Proc. of the IEEE Third Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, April 1994, IEEE Computer Society Press, Los Alamitos, CA, U.S.A., 1994

ANTREICH 2000

Antreich, G.: Unterlagensammlung zur Vorlesung Rechnergestützte Entwurfsverfahren, Stand 11/00, TU München, http://www.eda.ei.tum.de/lehre/REV/index.html, 2000

ARIS 1996

IDS Prof. Scheer GmbH: ARIS, 1996

ATTIE ET AL. 1993

Attie, P.C., Singh, M.P., Sheth, A., Rusinkiewicz, M.: Specifying and Enforcing Intertask Dependencies, Proc. 19th VLDB, Dublin, Irland, S. 134ff, Aug., 1993

ATTIE ET AL. 1996

Attie, P.C., Singh, M.P., Emerson, E. A., Sheth, A., Rusinkiewicz, M.: Scheduling Workflows by Enforcing Intertask Dependencies; Distributed Systems Engineering Journal, 3(4): 222-238, Dec. 1996, U.S.A., 1996

BALZERT 1998

Balzert, H.: Lehrbuch der Software Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung, Heidelberg, Spektrum, Akad. Verlag, 1998

BARBUCEANU UND LO 1999

Barbuceanu, M., Lo, W.-K.: Conversation Oriented Programming in COOL: Current State and Future Directions. Workshop on Specifying and Implementing Conversation Policies of the Autonomous Agents '99, Seattle, Washington, May, 1999

BECKER UND ZUR MÜHLEN 1999

Becker, J., zur Mühlen, M.: Towards a Classification Framework for Application Granularity in Workflow Management Systems; CAiSE*99, LNCS, Springer-Verlag, Berlin,

1999

BENTLEY ET AL. 1997
Bentley, R., Appelt, W., Busbach. U., Hinrichs, E., Kerr, D., Sikkel, S., Trevor, J. and Woetzel, G., : Basic Support for Cooperative Work on the World Wide Web; in International Journal of Human-Computer Studies: Special issue on Innovative Applications of the World Wide Web, Academic Press, Spring 1997

BINGLEY ET AL. 1992
P. Bingley, K.O. ten Bosch, and P. van der Wolf: Incorporating Design Flow Management in a Framework Based CAD System; Proc. IEEE/ACM International Conference on CAD - 92, pp. 538-545, Nov 1992

BINGLEY UND VAN DER WOLF 1990
P. Bingley and P. van der Wolf: A Design Platform for the NELSIS CAD Framework; Proc. 27th ACM/IEEE Design Automation Conference, pp. 146-149, June 1990

BLOTT ET AL. 1996
Blott, S., Relly, L., Schek, H.-J.: An open abstract-object storage system; ACM SIGMOD Internationl Conference on Management of Data, Montreal, Canada, 1996

BORGHOFF UND SCHLICHTER 1995
Borghoff, U., Schlichter, H.-J.: Rechnergestützte Gruppenarbeit; Springer-Verlag, Berlin, 1995

BORGHOFF UND SCHLICHTER 2000
Borghoff, U., Schlichter, H.-J.: Computer-Supported Cooperative Work: Introduction to Distributed Algorithms; Springer-Verlag, Berlin, 2000

BOSCH ET AL. 1991
K.O. ten Bosch, P. Bingley, and P. van der Wolf: Design Flow Management in the NELSIS CAD Framework; Proc. 28th ACM/IEEE Design Automation Conference, pp.711-716, June 1991

BOSCH ET AL. 1994
K.O. ten Bosch, P. van der Wolf and A.J. van der Hoeven: Design Flow Management: More than Convenient Tool Invocation; Proceedings 4th Int. IFIP 10.5 Working Conf. on Electronic Design Automation Frameworks, 1994

BOSCH 1995
Bosch, O. ten: Design Flow Management in CAD Frameworks; Delft university of Technology, also thesis, Delft, 1995

BOZAK 2000
Bozak, E.: Integration von autonomen Agenten in das Designflow-System CASSY; Studienarbeit 1769, Univ. Stuttgart, Abt. Anwendersoftware, 2000

BRETSCHNEIDER 1993
Bretschneider, F.: A Process Model for Design Flow Management and Planning; Fortschr. Ber. VDI Reihe 9 Nr. 157, VDI-Verlag, Düsseldorf, 1993

BROCKMAN ET AL. 1992
Brockman, J.B., Cobourn J.B., Jacome M.F., Director S.W.: The Odyssey CAD Framework; IEEE DATC Newsletter on Design Automation, 1992

BURGER UND SEMBACH 1994
Cora Burger, Frank Sembach: Ein Überblick über Verfahren zur rechnergestützten Ko-

operation; Fakultätsbericht 7/1994, Institut für Parallele und Verteilte Hochleistungsrechner, Univ. Stuttgart, 1994

BURGER 1997
Cora Burger: Groupware: Kooperationsunterstützung für verteilte Anwendungen; dpunkt-Verlag, Heidelberg, 1997

CFI 1991
CAD Framework Initiative: Framework Architecture Reference, Version 0.81, CFI Doc. No. 91, Austin, Texas, 1991

COCREATE SOFTWARE ET AL. 1997
CoCreate Software et al.: Workflow Management Facility, 1997 OMG Document: bom/97-08-05, 1997

CONKLIN UND BEGEMAN 1988
J. Conklin, M. L. Begeman: gIBIS: A Hypertext Tool for Exploratory Policy Discussion; Proc. 2nd International Conference on CSCW, Portland, 1988, in SIGCHI/SIGOIS ACM, New York, U.S.A., 1988

DART 1992
Dart, S. A.: Parallels in Computer-Aided Design Framework and Software Development Environment Efforts; in Rhyne, T.(Hrsg.): Electronic Design Automation Framework; Elsevier Science Publishers B.V., Holland, Seite 175ff, 1992

DEMARCO UND LISTER 1987
DeMarco, T., Lister, T.: Peopleware; Dorset House Publishing Co., New York, 1987

DEITERS ET AL. 1996A
Deiters, W., Jablonski, S., Klöckner, K., Schwab, K.: Paneldiskussion Workflow und CSCW; auf der GI Informatik'96, Klagenfurt, Östereich, 1996

DEITERS ET AL. 1996B
Deiters W., Herrmann T., Löffeler T., Striemer R.: Identifikation und Unterstützung semistrukturierter Prozesse in Prozeßorientierten Telekooperationssystemen; in Krcmar H., Schwabe H. (Hrsg.): Herausforderung Telekooperation - Einsatzerfahrungen und Lösungsansätze für ökonomische und ökologische, technische und soziale Fragen unserer Gesellschaft; Fachtagung DCSCW'96 Stuttgart, Springer-Verlag, Berlin, 1996

FRANK 1997
Aiko Frank: Integration von CSCW- und Workflow-Aspekten in Entwurfsumgebungen; 8. Workshop "Transaktionskonzepte", 29.-31. Jan. 1997, Weisendorf, in Datenbank Rundbrief der GI-Fachgruppe Datenbanken (FG 2.5.1), Ausg. 19, Mai, 1997

FRANK 1998
Aiko Frank: CSCW- und Workflow-Dienste unter CORBA für Entwurfsumgebungen; 9. Workshop "Transaktionen und ihre Anwendungen", 29.-30. Jan. 1998, Warnemünde, Datenbank Rundbrief der GI-Fachgruppe Datenbanken (FG 2.5.1), Ausg. 21, 1998

FRANK ET AL. 1992
Frank, A., Glaser, F., Hartmann, H., Klein, G.: Abschlußbericht ESPRIT-Projekt ARIADNE, IFT, München, 1992

FRANK 1999
Frank, A.: Towards an Activity Model for Design Applications. Proc. of the ISCA 14th

Intern. Conf. on Computers and Their Applications, April 7-9, Cancun, Mexico, 1999

FRANK 2001
Frank, A.: Agent Protocols for Integration and Cooperation in a Design Application Framework. Workshop on Software Agents and Workflows for Systems Interoperability of the 6th International Conference CSCW in Design, London, ON, Canada, July 14th, 2001

FRANK ET AL. 2000
Frank, A., Sellentin, J., Mitschang, B.: TOGA-A Customizable Service for Data-Centric Collaboration; Inform. Systems Vol. 25, No. 2, S. 157-176, Elsevier Science Ltd, GB, 2000

FRANK UND MITSCHANG 2001
Frank, A., Mitschang, B.: On Sharing of Objects in Concurrent Design; Proc. of the 6th International Conference on CSCW in Design, London, ON, Canada, 2001

GEORGAKOPOULOS ET AL. 1995
Georgakopoulos D., Hornick M., Sheth A.: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure; in Distributed and Parallel Databases, 3, Kluwer Academic Publishers, Boston MA, 1995

GEORGAKOPOULOS UND RUSINKIEWICZ 1996
Georgakopoulos, D., Rusinkiewicz, M.: From Transactions to Transactional Workflows; Tutorium bei der ICDE-12, New Orleans, U.S.A., 1996

GRADY 1992
Grady, R. B.: Practical Software Metrics for Project Management and Process Improvement; Prentice Hall, Englewood, 1992

GREENBERG UND ROSEMAN 96
Greenberg, S., Roseman, M.: Groupware Toolkits for Synchronous Work; in Trends in CSCW, John Wiley & Sons, 1996

GRUDIN 1994
Grudin, J.:Computer Supported Cooperative Work: History and Focus; Computer, May 1994, Vol. 27, No. 5, IEEE Computer Society, Los Vaqueros Circle, PO, U.S.A., 1994

GÜNTHÖR 1996
Günthör, R.: Ein Basisdienst für die zuverlässige Abwicklung langdauernder Aktivitäten; Dissertation an der Univeristät Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1996

HÄRDER UND REUTER 1983
Härder T., Reuter A.: Principles of Transaction-Oriented Database Recovery; ACM Computing Surveys, Vol. 15, No. 4, 1983, S. 287-248

HARRISON ET AL. 1990
Harrison, D., Newton, R., Spickelmier, R., Barnes, T.: Electronic CAD Frameworks; Proc. of the IEEE, Vol. 78, No. 2, S. 393-417, 1990

HASENKAMP ET AL. 1994
Ulrich Hasenkamp, Stefan Kirn, Michael Syrin (Hrsg.): CSCW-Computer Supported Cooperative Work; Addison-Wesley GmbH, Bonn 1994

HAUGENEDER 1994
Haugeneder, H. (Hrsg.): IMAGINE: Final Project Report; IMAGINE Technical Report

series, ESPRIT Project 5362. München, 1994

HEINL UND STEIN 1996
Petra Heinl, Katrin Stein: Behandlung semantischer Fehler im Workflow Management; GI Datenbank Rundbrief 17. Mai 1996

HEIß 1998
Heiß, G.: Evaluierung der Vorschläge für eine CORBA Workflow Facility und die Implementierung eines solchen Dienstes; Diplomarbeit, TU München, Februar 1998

HÜBNER 1998
Hübner, M.: Entwurf und Implementierung eines standortunabhängigen und plattformübergreifenden Prozeßplanungssystem; Diplomarbeit, Institut für Informatik, Lehstuhl für Wissens- und Datenbanken, TU München, Februar 1998

HUMPHREY 1989
Humphrey, W.S.: *Managing the Software Process*, Addison-Wesley Publ. Co., Reading Massachusetts, 1989

HUMPHREY 1995
Humphrey, W. S.: A Discipline for Software Engineering, Addison Wesley Publ. Co., Reading Massachusetts, 1995

IABG 1999
IABG: *Prominand*, http://www.iabg.de, 1999

IBM 1995
IBM Corporation: *IBM Flowmark Programming Guide*, IBM Corp., 1995

IBM 2000
IBM Corp.: *MQ Series Workflow*, www.ibm.com/software/ts/mqseries/workflow, 2000

IBM 2001
IBM Corp.: Lotus home page, http://www.lotus.com, 2001

JABLONSKI 1995
Stefan Jablonski: Workflow-Management-Systeme: Modellierung und Architektur; Intern. Thomson Publishing, Bonn, 1995

JABLONSKI ET AL. 1997
Jablonski, S., Böhm, M, Schulze, W. (Hrsgg.): Workflow-Management: Entwicklung von Anwendungen und Systemen; Facetten einer neuen Technologie; dpunkt-Verlag, Heidelberg, 1997

JABLONSKI UND BUßLER 1996
Jablonski, S., Bußler, C.: Workflow Management - Modeling Concepts, Architecture and Implementation, Int. Thomson Computer Press, London, 1996

JAESCHKE 1996
Jaeschke, P.: Geschäftsprozeßmodellierung mit INCOME; in G. Vossen, J. Becker: Geschäftsprozeßmodellierung und Workflow-Management, Intern. Thomson Publ., Bonn, 1996

JCF 1991
JCF Task Force Architecture: Jessy-Common-Framework Architecture Specification;

Jessi-Common-Framework project, ESPRIT 5082/7364, 1991

JÖRIS 1998

Joeris, G.: Aspekte und Konzepte der Flexibilität in Workflow-Managementsystemen, Proc. of the D-CSCW'98, Workshop: Flexibilität und Kooperation in Workflow-Management-Systemen; Tech. Report Angewandte Mathematik und Informatik 18/98-I, University of Münster, 1998

JOHANSEN 1991

R. Johansen et al.: Groupware: Computer Support for Business Teams; Free Press, New York, 1991

KAMATH UND RAMAMRITHAM 1996

Mohan Kamath, Krithi Ramamritham: Bridging the gap between Transaction Management and Workflow Management; URL http://www-ccs.cs.umass.edu/~kamath/nsf-wf.html, Univ. of Massachusetts, Amherst MA 01003, U.S.A., 1996

KATZENBACH UND SMITH 1993

Katzenbach, J. R., Smith, D. K.: The Discipline of Teams, Harvard Business Review, Nr. 3, S. 111-120, USA, 1993

KIM UND UNLAND 1994

S. Kirn, R. Unland: Workflow Management mit kooperativen Softwaresystemen: State of the Art und Problemabriß; Arbeitsbericht Nr. 29 am Inst. für Wirtschaftsinformatik, Universität Münster, 1994

KLEEDÖRFER 1998

Kleedörfer, R.: Prozeß- und Änderungsmanagement der integrierten Produktentwicklung; Dissertation, TU München, Fakultät für Maschinenwesen, 1998

KLEIN 1991

Klein, J.: Advanced Rule Driven Transaction Management; 36th IEEE Comp. Soc. Int'l. Conf. (CompCon) Spring 1991, S. 562ff., U.S.A., 1991

KOCH 1995

Michael Koch: The Collaborative Multi-User Editor IRIS; Technischer Bericht TUM-9524, Technische Universität München, 1995

KRISHNAKUMAR UND SHETH 1995

Krishnakumar, S., Sheth, A.: Managing Heterogenous Multi-System Tasks to Support Enterprise-wide Operations; in Distributed and Parallel Databases, 3, Kluwer Academic Publishers, Boston, 1995

LAURENCE UND LINK 1999

Laurence, R. P., Link, H. E.:.:The Role of Conversation Policy in Carrying Out Agent Conversations. Workshop on Specifying and Implementing Conversation Policies of the Autonomous Agents '99, Seattle, Washington, May, 1999

LEVI UND HAHNDEL 1993

Levi, P., Hahndel S.: Kooperative Systeme in der Fertigung; Kapitel 10 aus MÜLLER 1993, BI Verlag, Mannheim, 1993

LEYMANN 1995A

Frank Leymann: Supporting Business Transactions Via Partial Backward Recovery In

Workflow Management Systems; Proceedings BTW95, Dresden, 1995

LEYMANN 1995B
Frank Leymann: Workflows Make Objects Really Useful; in Proceedings 6th Intl. Workshop on High Performance Transaction Systems (HPTS), Asilomar CA, U.S.A., 1995

LEYMANN 1997A
Leymann, F.: Tutorium: Workflow-Management und Prdouktionsworkflows; anläßlich der BTW'97, Ulm, 1997

LEYMANN 1997B
Leymann, F.: Transaktionsunterstützung von Workflows; in Informatik Forschung und Entwicklung, Themenheft Workflow-Management, Band 12, Heft 2, Springer-Verlag, 1997

LUDWIG 1995
Ludwig, H.:Modellierung computerunterstuetzter Kooperationen im Bereich der betrieblichen Planung. In: Augsburger, W., Ludwig, H., Schwab, K. (Hrsg.): Koordinationsmethoden und -werkzeuge bei der computergestützten kooperativen Arbeit. Proceedings of the workshop Koordinationsmechanismen bei der computergestützten kooperativen Arbeit of the GI-Fachgruppe 5.5.1, Bamberg, 7.7.1995

NIIIP 1997
NIIIP Consortium, Task and Session Objects, Response to Business Object Task Force RFP-1, 1997

MARIUCCI 1998
Mariucci, M.: Untersuchung kooperativer Abläufe anhand von Beispielen und Entwicklung eines kooperativen Entwurfsvorgangsmodells für die Integration von Workflow- und CSCW-Aspekten, Diplomarbeit, TU München, Lehrstuhl für Datenbanksysteme, Wissensbasen, 1998

MARK UND FUCHS 1997
Mark, G., Fuchs, L., Sohlenkamp, M.: Supporting Groupware Conventions through Contextual Awareness, in Proc. of 5th E-CSCW, 253-268, Kluwer Academic Publisher, Netherlands, 1997

MAXWELL ET AL. 1996
Maxwell, K. D., Wassenhove, L. V., Dutta, S.: *Software Development Productivity of European Space, Military, and Industrial Application*s, in: IEEE Transactions on Software Engineering, Oct. 1996

MENTOR 2000
Mentor Graphics Corp.: *FPGA Advantage for HDL Design*; Werbebroschüre und Produktpräsentationsvideo, http://www.mentor.com, 2000

MICROSOFT 2001
Microsoft Corp.: *Exchange Server Home Page*, http://www.microsoft.com/exchange/default.asp, 2001

MINTZBERG 1992
Mintzberg, H.: Die Mintzberg-Struktur: Organisationen effektiver gestalten; Verlag Moderne Industrie (amerik. Originalausg.: Prentice Hall 1983), Landsberg,1992

MITSCHANG ET AL. 1996
B. Mitschang, T. Härder, N. Ritter: Design Management in CONCORD: Combining Transaction Management, Workflow Management, and Cooperation Control; Proc. 6th

Int. Workshop on Research Issues in Data Engineering: Interoperability of Nontraditional Database Systems (RIDE-NDS), New Orleans, 1996, pp160-168

MUCHITSCH 1998
Muchitsch, M.: Entwurf und prototypische Implementierung einer CORBA CSCW Facility; Diplomarbeit, TU München, Lehrstuhl für Datenbanksysteme, Wissensbasen, 1998

MÜLLER 1993
Müller, J. (Hrsg.): Verteilte Künstliche Intelligenz: Methoden und Anwendungen; BI-Wiss.-Verlag, Mannheim, 1993

NIIIP 1997
NIIIP Consorium: Task and Session Objects; Response to OMG Business Object Task Forces RFP-1, 1997

NORTHERN TELECOM 1997
Northern Telecom, Univ. of Newcastle upon Tyne: *Workflow Management Facility Specification*; OMG Proposal, August 1997, ftp://ftp.omg.org/pub/docs/bom/97-08-04.pdf

OMG 1996
Object Management Group: The Common Object Request Broker Architecture: Architecture and Specification; http://www.omg.org/corba/, Revision 2.0, OMG, July 1996

OMG 1997
Object Management Group: Workflow Management Facility, Request for Proposal; OMG Document cf/97-05-06, August 1997

OMG 2000
Object Management Group: Workflow Management Facility Specification, V1.2; OMG Document 2000-05-02, April 2000

ORFALI ET AL. 1997
Orfali, R., Harkey, D., Edwards, J.: Instant CORBA; John Wiley & Sons Inc., New York, 1997

ORFALI UND HARKEY 1998
Orfali, R., Harkey, D.: Client/Server Programming with Java(tm) and CORBA; 2nd Ed., John Wiley & Sons Inc., New York, 1998

OSETERWEIL UND SUTTON 1996
Leon J. Osterweil, Stanley M. Sutton Jr.: Using Software Technology to Define Workflow Processes; in Proc. of the NSF Workshop on Workflow and Process Automation, Athens GA, U.S.A., 1996

POMBERGER UND BLASCHEK 1996
Pomberger, G., Blaschek, G.: Software Engineering: Prototyping und objektorientierte Software-Entwicklung; 2. Auflage, Carl Hanser Verlag, München, Wien, 1996

PRINZ ET AL. 1998
Prinz, W., Mark, G., Pankoke-Babatz, U.: Designing Groupware for Congruency in Use; Proc. of the CSCW98, Seatte, Washington, U.S.A., ACM Publishing, pp. 373-382, 1998

RATHGEB 1994
Michael Rathgeb : Einführung von Workflow-Management-Systemen; in U. Hasenkamp, S. Kirn, M. Syring (Hrsg.): CSCW - Computer Supported Cooperative Work; Addison

Wesley, Bonn, 1994

REIN UND ELLIS 1991
G. J. Rein, C. Ellis: rIBIS: A Real-Time Group Hypertext System; Int. J. Man-Machine Studies 34, 1991

REINWALD 1995
Berthold Reinwald: Workflow-Management in verteilten Systemen; 2. Aufl., (entspr. Dissertation an FAU Erlangen), Teubner, 1995

REINWALD UND MOHAN 1996
Reinwald, B., Mohan, C.: Structured Workflow Management with Lotus Notes Release 4; Proc. 41st IEEE Comp. Soc. Int'l Conference, digest of papers, pp. 451-457, Santa Clara, California, 1996

REUTER UND SCHWENKREIS 1995
Andreas Reuter, Friedemann Schwenkreis: ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems; in Data Engineering March 1995, Vol. 18 No. 1, IEEE Computer Society, Washington D.C., U.S.A., 1995

RITTEL UND WEBBER 1973
H. Rittel, M. Webber: Dilemmas in a general Theory of Planning; Policies Sciences 4, 1973

RITTER ET AL. 1994
Ritter, N., Mitschang, B., Härder, T., Gesmann, M., Schöning, H.: Capturing Design Dynamics - The CONCORD Approach, Proc. 10th Int. IEEE Data Engineering Conf., Houston, US, 1994, pp. 440-451.

RITTER 1997
Norbert Ritter: Group-Authoring in CONCORD - A DB-based Approach; 12th Annual Symposium on Applied Computing (SAC'97), San Jose, U.S.A., 1997

RITTER 1997B
Ritter, N.: DB-gestützte Kooperationsdienste fuer technische Entwurfsanwendungen, (auch Dissertation), DISDBIS 33, 1997

RITTER UND MITSCHANG 1997
N. Ritter, B. Mitschang: Konzepte für die Assistenzfunktion kooperativer Designflows; Informatik Forschung und Entwicklung, Springer Verlag, 1997

ROLLER UND ECK 1999
Roller, D., Eck, O.: Knowledge-based techniques for product databases; Int. J. of Vehicle Design, Vol. 21, Nos. 2/3, Seite 243ff, 1999

ROSEMAN UND GREENBERG 1996
M. Roseman, S. Greenberg: Building Real Time Groupware with GroupKits, A Groupware Toolkit; ACM Transactions on Computer Human Interaction, 1996

RYBA 1996
Ryba, M.: Planung und Durchführung methodischer Entwurfsaktivitäten; (zzgl. Diss. Univ. Stuttgart 1995), Verlag Dr. Kovac, Hamburg, 1996

SAYEGH 1997
Sayegh, M.: Corba: Standard, Spezifikationen, Entwicklung; O'Reilly Verl., Köln, 1997

SCHALLER UND SCHWAB 1997
Schaller, K., Schwab, T.: Integration von asynchronen CSCW-Anwendungen; Kapitel

16.2 in [JABLONSKI ET AL. 1997], 1997

SCHILL 1996
Schill, A.: Rechernergetützte Gruppenarbeit in verteilten Systemen;Printice Hall, München, 1996

SCHLICHTER ET AL. 1997
J. Schlichter, M. Koch, M. Bürger: Workspace Awareness for Distributed Teams; in Proc. Workshop Coordination Technology for Collaborative Applications, Singapore, 1997

SCHMIDT UND RODDEN 1996
Schmidt, Kjeld, and Tom Rodden: Putting it all together: Requirements for a CSCW platform; in D. Shapiro, M. Tauber, and R. Traünmuller (eds.): The Design of Computer Supported Cooperative Work and Groupware Systems, North Holland, Amsterdam, 1996, pp. 157-176

SCHNEIDER 2001
Schneider, K.: Die zuverlässige Ausführung von Workflows. Eine kompensationsbasierte Fehler- und Ausnahmebehandlung für Workflow-Managment-Systeme; Dissertation, Univ. Stuttgart, 2001

SCHULZE 1997
Schulze, W.: Evaluation of the Submission to the Workflow Management Facility RFP; OMG Document bom/97-09-02, 1997

SCHULZE UND BÖHM 1996
Schulze, W., Böhm, M.: Klassifikation von Vorgangsverwaltungssystemen; in VOSSEN UND BECKER 1996, ITP, Bonn, 1996

SCHWENKREIS 1993
Friedemann Schwenkreis: APRICOTS - A Prototype Implementation of a ConTract System - Management of the Control Flow and the Communication System; in Proc. of the 12th Symposium on Reliable Database Systems, Princeton NJ, U.S.A., 1993

SCHWAB 1995
Klaus Schwab: Workflow Management Systeme: aktuelle Trends und Perspektiven; Festschrift zum 60. Geburtstag von Walter Augsburger, Univ. Bamberg, 1995

SELLENTIN ET AL. 1999
Sellentin, J., Frank, A., Mitschang, B.: TOGA-A Customizable Service for Data-Centric Collaboration; Proc. CAiSE*99, Springer Verlag, Heidelberg, Germany, 1999

SELLENTIN 2000
Sellentin, J.: Datenversorgung komponentenbasierter Informationssysteme; Springer-Verlag, Berlin, 2000

SENKEL 1998
Senkel, C.: Bewertung von Lotus Notes für einen Entwurf von CSCW-CORBA-Services; Diplomarbeit, TU München, Lehrstuhl für Datenbanksysteme, Wissensbasen, 1998

SHETH ET AL. 1996
Sheth, A., Georgakopoulos, D., Joosten, S., Rusinkiewicz, M., Scacchi, W., Wileden, J., Wolf, A.: Report from NSF Workshop on Workflow and Process Automation in Information Systems, Univ. of Georgia, Athens, Georgia, 1996

SHETH UND RUSINKIEWICZ 1993
Sheth, A., Rusinkiewicz, M.: On Transactional Workflows; Bulletin of the Technical

Committee on Data Engineering, 16(2), June 1993, IEEE Computer Society, U.S.A., 1993

SINGH 1997A

Singh, M. P.: Formal ASpects of Workflow Management - Part: Distributed Scheduling; Tech. Bericht TR-97-05, Dep. of Comp. Sc., North Carolina State Univ., Juni 1997

SINGH 1997B

Singh, M. P.: Formal ASpects of Workflow Management - Part: Semantics; Tech. Bericht TR-97-04, Dep. of Comp. Sc., North Carolina State Univ., Juni 1997

SPURR ET AL. 1994

K. Spurr, P. Layzell, L. Jennison, N. Richards (Hrsgg.): Software Assistance for Business Re-Engineering; John Wiley & Sons, Chichester, U.K., 1994

STAFFWARE 1997

Staffware, http://www.staffware.com, 1997

STREITZ ET AL. 1994

Streitz N. A.; Geiler J.; Haake J. M.; Hol J., DOLPHIN: Integrated Meeting Support across LiveBoards, Local and Remote Desktop Environments, in Proceedings of the ACM '94 Conference on Computer Supported Cooperative Work (CSCW '94), Chapel Hill, North Carolina, Okt. 22-26, 1994, 345-358, 1994

STUFFER 1993

Stuffer, R.: Planung und Steuerung der integrierten Produktentwicklung; Dissertation Technische Univ.München, Fakultät für Maschinenwesen, 1993

SUTTON ET AL. 93

Sutton, P., Brockman, J., Director, S.: Design Management Using Dynamically Defined Flows, in 30th Design Automation Conference, 1993

TANENBAUM 1992

Andrew S. Tanenbaum: Computer-Netzwerke; 2. Aufl., Wolfram’s Fachverlag, Attenkirchen, 1992

TAUDES ET AL. 1996

Alfred Taudes, Peter Cilek, Martin Natter: Ein Ansatz zur Optimierung von Geschäftsprozessen; in [VOSSEN UND BECKER 1996], International Thomson Publishing GmbH, Bonn, 1996

TEEGE 1993

Gunnar Teege: The Activity Support System TACTS; Tech. Report TUM-I9306, TU München, 1993

TEEGE 1996

Teege, G.: CSCW-Systeme: Unterstützung von Gruppenarbeit; Skript zur Vorlesung an der TU München, Wintersemester 96/97, München, 1996

TEEGE 1996A

Teege, G.: Object-Oriented Activity Support: A Model for Integrated CSCW Systems; Computer Supported Cooperative Work - The Journal of Collaborative Computing 5(1) 1996, pp. 93-124

TEUFEL ET AL. 1995

S. Teufel, C. Sauter, T. Mühlherr, K. Bauknecht: Computerunterstützung für die Grup-

penarbeit; Addison-Wesley, Bonn, 1995

THOMAS 1995
Peter Thomas (Ed.): CSCW Requirements and Evaluation; Springer-Verlag, London, U. K., 1995

VLADIMIRESCU UND LIU 1980
Vladimirescu, A., Liu, S.: The Simulation of MOS Integrated Circuits Using SPICE2, ERL Memo No. ERL M80/7, Electronics Research Laboratory, University of California, Berkeley, October 1980

VOSSEN 1987
K. Voss: Nets in Office Automation; in W. Brauer, W. Reisig, G. Rozenberg (Hrsg.): Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986 Part II; LNCS Vol. 255, Springer-Verlag, Berlin, 1987

VOSSEN 1995
Gottfried Vossen: The Subject; in F. Leymann, H.J. Schek, G. Vossen (Hrsg.): Transactional Workflows - Dagstuhl-Seminar-Report; 152; Schloss Dagstuhl, 1995

VOSSEN UND BECKER 1996
Vossen, G., Becker, J.: Geschäftsprozeßmodellierung und Workflow-Management; International Thomson Publishing GmbH, Bonn, 1996

WAGNER 2001
Wagner, F.: Erstellung eines Contraints-basierten Aktivitätenmanagers für CASSY; Diplomarbeit Nr. 1896, Fak. f. Informatik, Univ. Stuttgart, Mai 2001

WAINER ET AL. 1996
J. Wainer, M. Weske, G. Vossen, C. B. Medeiros: Scientific Workflow Systems; NSF Workshop on Workflow and Process Automation: State-of-the-art and Future Directions, Athens, GA, May 1996

WFMC 1995
The Workflow Management Coalition: The Workflow Reference Model, http://www.wfmc.org, WFMC-TC-1003, 1995

WFMC 1996
The Workflow Management Coalition: *Workflow Client Application (Interface 2) Application Interface (WAPI)*; Version 1.2, http://www.wfmc.org/, 1. Oct. 1996

WILSON 1991
P. Wilson: Computer Supported Cooperative Work; Oxford, U.K.: Intellect Books, 1991

WINDRIVER 2001
WindRiver Inc.: SNiFF+, http://www.windriver.com/products/html/sniff.html, 2001

WINOGRAD UND FLORES 1986
Winograd, T., Flores, F.: Understanding Computers and Cognition: A New Foundation for Design; Ablex, Norwood NJ, 1986

WODTKE ET AL. 1995
Wodtke, D., Kotz-Dittrich, A., Muth, P., Sinnwell, M., Weikum, G.: Mentor: Entwurf einer Workflow-Management-Umgebung basieren auf State- und Activitycharts; BTW95, Dresden, 1995

VAN DER WOLF ET AL. 1990
P. van der Wolf, P. Bingley, and P. Dewilde: On the Architecture of a CAD Framework:

The NELSIS Approach; Proc. European Design Automation Conference, pp. 29-33, March 1990

Anhang A: Glossar

Hier werden einige Begriffe beschrieben, wie sie im Rahmen der vorliegenden Arbeit verwendet werden.

Akteur
Ein Akteur ist für die Durchführung einer Aktivität verantwortlich. Dieser kann ein Programm, eine Maschine oder eine menschliche Person sein.

Aktivität
Eine Aktivität ist ein logischer Arbeitsschritt innerhalb eines automatisierten Prozesses (z.B. Workflow).

Arbeit
Unter Arbeit verstehen wir einen zielorientierten Vorgang, z.B. zur Erstellung eines Produkts oder der Erbringung einer Dienstleistung (Seite 18).

Arbeitsvorgang
Der Arbeitsvorgang stellt die Art der Durchführung einer Arbeit insbesondere deren Ablauf dar.

Arbeitsschritt
Ein Arbeitsvorgang kann in verschiedene Arbeitsschritte aufgeteilt werden.

Awareness
Die Awareness beschreibt das gegenseitige Existenzbewußtsein verschiedener an einer Arbeit beteiligter Personen. Sie bildet häufig die Grundlage für kooperatives Arbeiten (Seite 38).

CSCW, CSCW-System
Siehe Definitionen auf Seite 37.

Constraint
Ein Constraint des ADM beschreibt Abhängigkeiten für die Durchführung von Aktivitäten.

Designflow
Ein Designflow ist die Beschreibung eines Entwurfsprozesses, siehe Definitionen ab Seite 80.

Designflow-Management
Siehe Definitionen ab Seite 80.

Entität des ADM
Objekte und Aktivitäten des ADM werden Entitäten genannt.

Entwurfsprozeß
Ein Entwurfsprozeß im eigentlichen Sinne ist in Abschnitt 1.2 und 2.2.4 beschrieben. In dieser Arbeit bezeichnen wir jedoch alle kooperativen Prozesse mit ähnlichen Anforderungen an eine umfassende Systemunterstützung als Entwurfsprozeß.

Entwurfsobjekt
Im Rahmen eines Entwurfsprozesses wird ein Entwurfsobjekt erstellt (siehe auch Definition auf Seite 80).

Groupware
Groupware sind CSCW-Systeme für die Unterstützung von Gruppenarbeit.

Informationsraum
In dem Informationsraum des ADM stehen Objekte zur Verfügung, die kooperativ genutzt werden können (siehe auch allg. Seite 45).

Interaktion
Interaktion bedeutet eine Kommunikation zwischen mindestens zwei Interaktionspartnern. Die Interaktion wird im ADM protokollbasiert durchgeführt.

Interaktionsprotokoll
Ein Interaktionsprotokoll beschreibt den Ablauf einer Interaktion. Im ADM existieren dazu Initiatorprotokolle, die von den jeweils initiierenden Interaktionspartnern verwendet werden. Die Komplementärprotokolle vervollständigen auf der ’Empfängerseite’ dieses Protokoll.

Kollaboration
Unter Kollaboration verstehen wir hier die Zusammenarbeit verschiedener Personen, die nicht in einer Konkurrenzsituation stehen. Die Kollaboration muß jedoch nicht kooperativ in unserem Sinne sein.

Konversation
Eine Konversation ist die konkrete Durchführung eines Interaktionsprotokolls.

Kooperation
Kooperation ist eine Kollaboration, die sich durch das bewußte Zusammenarbeiten und Miteinander auszeichnet. Persönliche Ziele sind zweitrangig und eine kooperative Arbeitsgruppe trifft zusammen Entscheidungen (Seite 38).

Koordination
Koordination bedeutet Regelung. Im Zusammenhang mit Arbeitsvorgängen ist dies etwa die Regelung der Durchführung von Arbeitsschritten und Ressourcennutzung.

Notifikation
Die Notifikation im ADM dient vor allem der Propagierung von Änderungen der Objekte. Die Notifikation kann aktiv (*push*) oder passiv (*pull*) aus Sicht der Objekte erfolgen.

Nutzungsbeziehung
Die Nutzungsbeziehung des ADM erlaubt die Registrierung von Aktivitäten bei Objekten, um auf diese zuzugreifen bzw. über Änderungen der Objekte notifiziert zu werden. Die Nutzungsbeziehung wird durch das sog. Nutzungsprotokoll aufgebaut.

Objekt des ADM
Ein Objekt ist eine Ressource, die im Informationsraum verwaltet werden kann.Typischerweise sind dies allgemeine Daten und Entwurfsobjekte.

Workflow
Siehe Definition auf Seite 49.

Workflow-Management
Siehe Definition auf Seite 49.

Workspace
Ein Workspace enthält alle für die Durchführung einer Aufgabe notwendigen Ressourcen (Abschnitt 3.3.5 auf Seite 45). Damit ist ein Workspace ein Überbegriff des Informationsraums.

Zerlegung des Entwurfsobjekts
Um ein Entwurfsobjekt zu erstellen, wird häufig das Prinzip ’divide et impera’ angewendet. So wird das Entwurfsobjekt (rekursiv) in kleinere Teile zerlegt, bis es von einzelnen Personen geeignet bearbeitet werden kann. Typischerweise fallen die Zerlegung und die Vergabe von Unteraufträgen (d.h. Delegation) innerhalb eines Entwurfsprozesses zusammen.

Anhang B: Syntax der Designflow-Spezifikationssprache

DesignflowSpecification ::= **DESIGNFLOW (** (DesignflowActivity) (ActivityDefintion)* (ResourceDefinition)* (ProgramDefinition)* (ActorDefinition)***)**

ActivityDefinition ::= DesignflowActivity | WorkflowActivity | GroupwareActivity | PrimitiveActivity |

DesignflowActivity ::= **DESIGNFLOW-ACTIVITY:** UniqueName **(GOALS:** [GoalDefinitions] **DESIGN-OBJECTS:** [DesignObjects] **ACTORS:** [Actors] **RESOURCES:** [Resources] **ACTIVITIES:** [SubActivities] **DESIGN-PRIMITIVES:** [DesignPrimitives] **PROGRAM:** [ProgramDefinition] **)**

WorkflowActivity ::= **WORKFLOW-ACTIVITY:** UniqueName **(DESIGN-OBJECTS:** [DesignObjects*] **ACTORS:** [Actor*] **RESOURCES:** [Resource*] **ACTIVITIES:** [SubActivity*] **WF-PRIMITIVES:** [WFPrimitive*] **APPLICATION:** [Facility]**)**

GroupwareActivity ::= **GROUPWARE-ACTIVITY:** UniqueName **(GOALS:** [GoalDefinitions] **DESIGN-OBJECTS:** [DesignObjects*] **ACTORS:** [Actor*] **RESOURCES:** [Resource*] **APPLICATION:** [Facility]**)**

PrimitiveActivity ::= **PRIMITIVE-ACTIVITY:** UniqueName **(DESIGN-OBJECTS:** [DesignObjects*] **ACTORS:** [Actor*] **RESOURCES:** [Resource*] **)**

GoalDefinitions ::= EMPTY | GoalDefinition (**,** GoalDefinition)*

GoalDefinition ::= String | OQL-Expression

DesignObjects ::= EMPTY | ObjectType UniqueName (**,** ObjectType UniqueName)*

ObjectDefinition ::= **OBJECT:** UniqueName**(** AbstractType Identifier [**=** value] (**,** (Type | AbstractType) Identifier [**=** value])* **)**

Actors ::= UniqueName (**,** UniqueName)*

ActorDefinition ::= **ACTOR** UniqueName **(** String **)**

Resources ::= UniqueName (**,** UniqueName)*

Sub-Activities ::= UniqueName (**,** UniqueName)*

DF-Primitives ::= DF-Primitive (**,** DF-Primitive)*

DF-Primitive ::= UniqueName **USES** UniqueName | UniqueName **DELEGATES** UniqueName | Constraint

DF-Primitives ::= WF-Primitive (**,** WF-Primitive)*

WF-Primitive ::= String //dies wird abhängig von der verwendeten Workflow Facility bestimmt

ProgramDefinition ::= **PROGRAM** UniqueName **(** String (, String)* **)**

ResourceDefinition ::= **RESOURCE** UniqueName **(** (AbstractType Identifier [**=** value] (**,** (Type | AbstractType) Identifier [**=** value])*) | **EXTERNAL** String **)**

SubActivities ::= UniqueName Indentifier (, UniqueName Indentifier)*

Facility ::= UniqueName (ParamList)

AbstractTypeDefinition ::= **TYPE: (** AbstractType Identifier [**=** value] (**,** Type Identifier [**=** value])* **)** TypeName

AbstractType ::= Type | Identifier

Type ::= **integer** | **string** | **float**

ObjectType ::= Letter Char*

UniqueName ::= Letter Char*

ParamList ::= Char* | Char***,** ParamList

TypeName ::= Letter+

Integer ::= { (Digit)+ }

RealNumber ::= { Integer **.** (Digit)* [(**e**|**E**)[**-**|**+**]Integer] }

String ::= **“**(Char)***”**

Char ::= ’ ’ .. ’**~**’

Letter ::= **a** .. **z** | **A** .. **Z**

Digit ::= **0** .. **9**

Anhang C: Designflow für die Standortplanung

In einfacher Schreibweise:

```
designflow(
groupware-activity: GESCHÄFTSFÜHRUNG(
   ACTORS:
      Geschäftsführung
   DESIGN-OBJECTS:
       Planungsbericht
   RESOURCES:
      Standortliste
   GOAL:
      “Festlegung eines neuen Filialstandorts und dessen Planung”
)

designflow-activity: STANDORTPLANUNG(
   RESOURCES:
      Standortliste
   DESIGN-OBJECTS:
       Planungsbericht
   ACTORS:
      Planungsgruppe, Geschäftsführung
   GOAL:
      Planungsbericht.fertig ==  TRUE
   ACTIVITIES:
      MARKTANALYSE:          M,
      STANDORTSUCHE:         S,
      KOSTENABSCHÄTZUNG:  K,
      GESCHÄFTSFÜHRUNG:   G,
   DESIGN_PRIMITIVES:
      M starts_before {S, K},
      G delegates {M, S, K},
      G uses Standortliste,
      G uses Planungsbericht,
      K uses Planungsbericht,
      K uses Standortliste,
      S uses Standortliste
)

workflow-activity: MARKTANALYSE(
   DESIGN-OBJECTS:
      Standortliste
   ACTORS:
      Marktforschungsleiter
   GOAL:
      Standortliste_fertig=TRUE
   ACTIVITIES:
      DATENSPEZIFIKATION:    D,
      REGIONALDATEN:         R,
      KONKURRENZDATEN:       K,
      ZUSAMMENFASSEN:        Z,
      AUSWAHL:               A
   WF_PRIMITIVES:
      Z uses Standortliste,
      A uses Standortliste
   APPLICATION:
   JFLOW_FLOWMARK( MARKTANALYSE,
   Standortliste)
)

designflow-activity: STANDORTSUCHE(
   DESIGN-OBJECTS:
      Standortliste
   ACTORS:
```

```
        Leiter_Filialprojekt
    GOAL:
         “Begutachtung der Immobilien und
        Grundstücke.”
    ACTIVITIES:
        STANDORTAUSWAHL      S,
        BEWERTUNG[i]:        B[i],
        LISTENERSTELLG:      L,
    DESIGN_PRIMITIVES:
        {S,L,B[i]} uses Standortliste,
        S delegates B[i],
        L starts_after S
)

designflow-activity: BEWERTUNG(
    DESIGN-OBJECTS: Standorteintrag
    ACTORS: Gutachter
    GOAL: Standorteintrag.ERLEDIGT == true
    APPLICATION: StandortMaske(Standorteintrag)
)
PROGRAM: StandortMaske( netscape, “app_serv.uni.edu/app.html?standortmaske%20$Standorteintrag$”)

primitive-activity: LISTENERSTELLUNG(
    RESOURCES:
        Standortliste,
        Standortbericht
    ACTORS:
        automatic
    APPLICATION:
        Berichtgenerator( Standortliste, Standortbericht)
)

designflow-activity: KOSTENABSCHÄTZUNG(
    ACTORS:
        Bauleitung,
        Bauplanungsberechner
    GOAL:
        “Berechnung der Kosten einer Filiale”
    ACTIVITIES:
        BERECHNUNG:              B,   //Berechnung der Gesamtkosten + “Gutachten”, wird nicht detailliert
        VERSORGUNGSPLANUNG       V,   //Abschätzung der logistischen Versorgung, wird nicht detailliert
        FILIALPLANUNG            F    //Geschätzte Baukosten, wird nicht detailliert
    DESIGN_PRIMITIVES:
        B delegates V,
        B delegates F,
        V uses Standortliste,
        F uses Standortliste
)

OBJECT: Standortbericht( Report = “reportserver.uni.edu/projects/new_branch/report.txt” )
PROGRAM: Berichtgenarator( REPGEN.EXE, $Standortliste.table_name$ + “-path ” + Standortbericht.Report )

DESIGN-OBJECT: document Planungsbericht(
    string goal,
    list of ( text Standorte, value Kosten ),
    text Bericht,
    text Gewählter_Standort,
    bool fertig
)
)
```

Anhang D: Constraints in CASSY

XML-Defintion:

```
<?xml version=”1.0” encoding=”ISO-8859-1”?>
<!DOCTYPE CONSTRAINT_DEFINITIONS SYSTEM “ActivityManagement.dtd”>

<CONSTRAINT_DEFINITIONS>

<!-- Die erste Aktivitt darf erst starten, wenn die zweite
     fertig (terminate) oder abgebrochen (cancel) wurde. -->
<DEPENDENCY NAME=”after” VERSION=”0.1”>
<OR>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
    <EVENT NAME=”terminate”>
        <PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </SEQUENCE>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
    <EVENT NAME=”cancel”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Sobald die erste Aktivitt erfolgreich beendet wurde, darf die
     zweite Aktivitt nicht mehr gestartet werden. -->
<DEPENDENCY NAME=”disables” VERSION=”0.1”>
<OR>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
    <EVENT NAME=”terminate”>
        <PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </SEQUENCE>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”cancel”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </COMPLEMENT>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Dieser Constraint wird zwischen einer Activity und einem Actor
     eingefügt und stellt sicher, dass die Ressource exklusiv für die
     Aktiviätt zur Verfügung steht -->
<DEPENDENCY NAME=”exclusively_needs” VERSION=”0.1”>
  <OR>
    <SEQUENCE>
        <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
        <EVENT NAME=”busy”>
       <PARAMETER NAME=”resource” VALUE=”$1.id”/>
       <PARAMETER NAME=”activity” VALUE=”$j”/>
        </EVENT>
    </SEQUENCE>
    <SEQUENCE>
        <EVENT NAME=”ready”>
       <PARAMETER NAME=”resource” VALUE=”$1.id”/>
       <PARAMETER NAME=”activity” VALUE=”$j”/>
        </EVENT>
        <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    </SEQUENCE>
    <COMPLEMENT>
```

```
        <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    </COMPLEMENT>
  </OR>
</DEPENDENCY>

<!-- Die erste Aktivitt erzeugt Daten, die die zweite Aktivitt
     benötigt. Daher darf die zweite Aktivitt nur starten, wenn die
      erste erfolgreich beendet wurde (terminate) -->
<DEPENDENCY NAME=”feeds” VERSION=”0.1”>
<OR>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”terminate”>
        <PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Wenn die erste Aktivitt erfolgreich beendet wurde, dann darf
     die zweite nicht auch terminieren. -->
<DEPENDENCY NAME=”forbids” VERSION=”0.1”>
<OR>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”cancel”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
    <EVENT NAME=”cancel”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </COMPLEMENT>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Wenn die erste Aktivitt terminiert, dann muss auch die
     zweite Aktivitt terminieren. -->
<DEPENDENCY NAME=”implies” VERSION=”0.1”>
<OR>
  <EVENT NAME=”terminate”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  <COMPLEMENT>
    <EVENT NAME=”terminate”>
        <PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Eine der beiden Aktivitten muss terminieren. -->
<DEPENDENCY NAME=”or” VERSION=”0.1”>
<OR>
  <EVENT NAME=”terminate”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  <EVENT NAME=”terminate”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
</OR>
</DEPENDENCY>

<!-- Wenn die erste Aktivitt terminiert oder gecanceled wurde, darf
     und muss die zweite Aktivitt starten. -->
<DEPENDENCY NAME=”sequence” VERSION=”0.1”>
<OR>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”terminate”>
        <PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”cancel”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
```

```
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
</OR>
</DEPENDENCY>

<!-- Die erste Aktivitt muss terminieren und danach muss die
     zweite Aktivitt starten. -->
<DEPENDENCY NAME=”sequence_terminate” VERSION=”0.1”>
<SEQUENCE>
  <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  <EVENT NAME=”terminate”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
</SEQUENCE>
</DEPENDENCY>

<!-- Wenn die erste Aktivitt abgebrochen wurde, muss die zweite
     (Kompensations-)Aktivitt starten. Zwischen den Ereignissen kann
     aber eine lange Zeit liegen.  -->
<DEPENDENCY NAME=”sequence_cancel” VERSION=”0.1”>
<OR>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”cancel”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME=”cancel”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>


<!-- Sobald die erste Aktivitt gestartet wurde, kann auch die
     zweite starten -->
<DEPENDENCY NAME=”start_enables” VERSION=”0.1”>
<OR>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>


<!-- Die Aktivitt muss starten. -->
<DEPENDENCY NAME=”starts” VERSION=”0.1”>
<EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
</DEPENDENCY>

<!-- Wenn beide Aktivitten starten, dann muss die erste Aktivitt
     vor der zweiten starten. -->
<DEPENDENCY NAME=”starts_before” VERSION=”0.1”>
<OR>
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </COMPLEMENT>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Die Aktivitt muss erfolgreich beendet werden. -->
<DEPENDENCY NAME=”terminates” VERSION=”0.1”>
<EVENT NAME=”terminate”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
</DEPENDENCY>

<!-- Die erste Aktivitt darf nur während der zweiten laufen. -->
<DEPENDENCY NAME=”while” VERSION=”0.1”>
<OR>
```

```
  <SEQUENCE>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”terminate”>
        <PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”terminate”>
        <PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </SEQUENCE>
  <COMPLEMENT>
    <EVENT NAME=”start”><PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
  </COMPLEMENT>
</OR>
</DEPENDENCY>

<!-- Genau eine der beiden Aktivitten muss terminieren. -->
<DEPENDENCY NAME=”xor” VERSION=”0.1”>
<AND>
  <OR>
    <EVENT NAME=”terminate”>
        <PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    <EVENT NAME=”terminate”>
        <PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
  </OR>
  <OR>
    <COMPLEMENT>
      <EVENT NAME=”terminate”>
          <PARAMETER NAME=”id” VALUE=”$0.id”/></EVENT>
    </COMPLEMENT>
    <COMPLEMENT>
      <EVENT NAME=”terminate”>
          <PARAMETER NAME=”id” VALUE=”$1.id”/></EVENT>
    </COMPLEMENT>
  </OR>
</AND>
</DEPENDENCY>
</CONSTRAINT_DEFINITIONS>
```

Anhang E: IDL: CASSY

Anhang E.1 ActorInterface.idl

```
#include "CSCW.idl"

module ActorInterface
{
//Vorwaertsdefinitionen der einzelnen Interfaces
interface User;
interface Group;
interface BasicResourceAdapter;
interface ResourceDependency;
interface PartOfDependency;
interface IsADependency;

//gemeinsamer Datentyp
enum ActorType { USER, GROUP, RESOURCE};
enum NotificationType{  NONE,
                        ALL,     //Weiterleitung jeder Änderung (inklusive Objekte und Änderung)
                        SPECIAL, //für zusätzliche Modi
                        PASSIVE};//Nur Mitteilung, daß eine Änderung stattgefunden hat

typedef sequence<User>          UserList;
typedef sequence<Group>         GroupList;
typedef sequence<BasicResourceAdapter> BasicResourceAdapterList;

/*************************    Actor     **********************************/
interface Actor  {
//Oberklasse von Group,User und BasicRessourceAdapter
   string getName();
   boolean setName(in string name);
   string getID();
   ActorType isa();
   boolean load();
   boolean save();

   boolean AssignActivity(in Activity act, in unsigned long mode);
   boolean RemoveActivity(in Activity act);
   boolean ReadAssignedActivities(out ActvityList acts);
};

/*************************    Group     **********************************/
interface Group :Actor{
// Attribute
  /*

   attribute string Groupname;// Gruppenname

   readonly attribute string GroupID;// Gruppen-ID
   jetzt in der Oberklasse mit getName(),setName(),getId()
  */
   /*
     attribute string Description; Allg. Beschreibung der
                  Gruppe z.b. Gruppenziel
     wird durch Methoden getDescription, setDescription ersetzt
   */

// Methoden
   boolean SaveGroupData();// Speichern des Gruppenzustandes
   boolean getDescription(out string description);
   boolean setDescription(in string description);
   // liefert Group-Objekt der CSCW-Facility zurück
   Object getImplementationObject();
   // Beziehungen zur Group-Klasse
   boolean is_SubGroup(in Group grp);// Überprüft, ob die angegebene Gruppe eine Untergruppe ist
// Beziehungen zur User-Klasse
   // Aufnahme eines neuen Gruppenmitglieder
   boolean AddGroupmember(in User usr, in unsigned long mode);
   // Entfernen eines Gruppenmitglieser
   boolean RemoveGroupmember(in string userid, in unsigned long mode);
   // Auflisten der Gruppenmitglieder
   boolean ListGroupmembers(out UserList members);
   // Überprüft, ob der angegeben User ein Gruppenmitglied ist
   boolean is_Groupmember(in string userid);
   // Speichern der Gruppenmitglieder
   boolean SaveGroupmembers();
   // Lesen der Gruppenmitglieder aus einer, mit SaveGroupMembers
   // erstellten Sicherung
   boolean ReadGroupmembers(out UserIDList uidl);
   // Liefert den User, der innerhalb der Gruppe eine bestimmte Role besitzt
   UserGetRoleOwner(in string rolename);
```

```
   // Entfernt alle Gruppenmitglieder
   boolean RemoveAllGroupmembers();

// Beziehungen zur Role-Klasse
   // Ordnet dem User eine neue Rolle innerhalb der Gruppe zu
   boolean AddRoleToGroupmember(in string userid, in Role r, in unsigned long mode);
   // Entzieht dem User eine Rolle innerhalb der Gruppe
   boolean RemoveRoleFromGroupmember(in string userid, in string rolename,
                                     in unsigned long mode);
   // Auflisten der Rollen eines Gruppenmitgliedes
   boolean ListGroupmemberRoles(in string userid, out RoleList rl);
   // Speichern der Rollen der Gruppenmitgliedes
   boolean SaveGroupmemberRoles(in string userid);
   // Lesen der Rollen des Gruppenmitgliedes aus einer, mit
   // SaveGroupmemberRoles erstellten Sicherung
   boolean ReadGroupmemberRoles(in string userid, out RoleNameList rnl);

   void dump();
   // Löschen der Sicherungsdateien beim Entfernen des Group-Objektes
   void removeFiles();
};// Group-Interface

/********************************** User ******************************/
interface User : Actor {
   enum ConnectionState {
         Connected,
         Disconnected
   };
   /*       User-Daten                */
   // liefert User-Objekt der CSCW-Facility zurück
   Object getImplementationObject();
   boolean getConnectionState(out ConnectionState connState);
   boolean ChangePassword(in string OldPwd, in string NewPwd);
   boolean CheckPassword(in string Pwd);
   string Connect() ;
   void Disconnect();
   boolean SaveUserData();
   boolean SendMessage(in Message msg);
   boolean ReceiveMessage(out Message msg);

   // Manipulation der Beziehungen zur Role-Klasse
   boolean AddRole(in Role r, in string groupid, in unsigned long mode);
   boolean RemoveRole(in string rolename, in string groupid, in unsigned long mode);
   boolean SaveRoles(in string groupid);
   boolean ReadRoles(in string groupid,out RoleNameList rnl);
   boolean ListRoles(in string groupid, out RoleList rl);
   void dumpRoles();

   boolean AddGroup(in Group g, in unsigned long mode);
   boolean RemoveGroup(in string groupid, in unsigned long mode);
   boolean RemoveAllGroups();
   boolean SaveGroups();
   boolean ReadGroups(out GroupIDList gidl);
   boolean ListGroups(out GroupList gl);

   void removeFiles();
};

/********************************** BasicResourceAdapter ******************************/
interface BasicResourceAdapter : Actor {
   enum ResourceType { DOCUMENT, PERSISTENT_OBJECT, CLIENT_APPLICATION, SERVER_APPLICATION,
                   EVENT_CHANNEL, REALTIME_CHANNEL, SERVICE };
   enum ResourceAdapterType {TASK_RA, APPLICATION_RA, DATA_RA, SERVICE_RA, SYSTEM_RA };

   // liefert “original” Objekt einer Facility oder eines externen Systems zurück
   Object getImplementationObject();
   ResourceType GetResourceType();
   ResourceAdapterType GetResourceAdapterType();
   boolean AssignResourceObject(in Object obj);
   Object GetRessourceObject();

   boolean SaveAdapterInformation(in ResourceAdapterType adapter_type);

   //Informationsraum
   boolean RegisterInInfoSpace();
   boolean RemoveFromInfoSpace();

   //Interaktion
   boolean receiveMessage( in any msg);
   boolean sendMessage( in CORBA::Object receiver, any msg );

   void removeFiles();
};

/********************************** ResourceDependency ******************************/
interface ResourceDependency {
   readonly ResourceList Objects;
```

```
   readonly NotificationType notification;
   readonly string description;

   boolean register( in BasicResourceAdapter object);
   boolean unregister( in BasicResourceAdapter object);
   boolean receiveMessage( in any msg);
   boolean sendMessage( in CORBA::Object receiver, any msg );

};
/********************************** PartOfDependency ******************************/
interface PartOfDependency : PartOfDependency {
   readonly BasicResourceAdapter SuperObject;
   boolean registerSuperObject( in BasicResourceAdapter object);
};
/********************************** IsADependency *********************************/
interface PartOfDependency : IsADependency {
   readonly BasicResourceAdapter BaseObject;
   boolean registerBaseObject( in BasicResourceAdapter object);
};

/*********************** ActorObjectFactory ********************************/
interface ActorObjectFactory {

   Actor create_actor_object(in string name, in string id, in ActorType type);
   Actor create_empty_actor_object(in string id);
 };
/*********************** UserObjectFactory ********************************/
interface UserObjectFactory {

   User create_user_object(in string name, in string id);
   User create_empty_user_object(in string id);
 };
/*********************** GroupObjectFactory ********************************/
interface GroupObjectFactory {

   Group create_group_object(in string name, in string id);
   Group create_empty_group_object(in string id);
 };
/*********************** ResourceAdapterObjectFactory ********************************/
interface ResourceAdapterObjectFactory {

   BasicResourceAdapter create_resourceadapter_object(in string name, in string id);
   BasicResourceAdapter  create_empty_resourceadapter_object(in string id);
 };

};
/*********************** DependencyFactories ********************************/
interface ResourceDependencyObjectFactory {

   User create_dep_object(in string name, in string id);
   User create_empty_dep_object(in string id);
 };
interface PartOfDependencyObjectFactory : ResourceDependencyObjectFactory {};
interface IsADependencyObjectFactory: {};
```

Anhang E.2 ActivityInterface.idl

```
#include "ActorInterface.idl"
#include “ActivityManagement,idl”

module ActivityInterface
{

interface Activity;
interface PrimitiveActivity;
interface BasicActivity;
interface ToplevelActivity;
interface GroupwareActivity;
interface WorkflowActivity;
interface DesignflowActivity;
interface Constraint {

// Type definitions

// ActivitySequence
typedef sequence<Activity> ActivityList;
typedef sequence<ActorInterface::BasicResourceAdapter> ResourceList;
typedef sequence<Constraint> ConstraintList;

// Activity states
```

```
enum stateType {initialized,
                started,
                stopped,
                canceled,
                not_initialized,
                finished };

enum ActivityType { ACTIVITY,
                    PRIMITIVEACTIVITY,
                    BASICACTIVITY,
                    TOPLEVELACTIVITY };

enum BasicActivityType { WORKFLOWACTIVITY,
                         GROUPWAREACTIVITY,
                         DESIGNFLOWACTIVITY };

//*****************************************************************
* Ein Constraint ist ein Beziehung zwischen Objekten
* (Sub-Aktivitäten, Ressourcen, Zeiten, ...) mit der die
* Folge der Ereignisse eingeschränkt wird. Die
* DesignFlowActivity ist die Factory für Constraints.
//*****************************************************************
interface Constraint {
   readonly attribute string name;
   readonly attribute ObjectSeq parameters;
};
typedef sequence<Constraint> ConstraintList;

//*****************************************************************
//**                                                             **
//**                    Activities                               **
//**                                                             **
//*****************************************************************


//*****************************************************************
//                     Activity
//*****************************************************************
interface Activity {
//****** Attributes ******
attribute unsigned long resources_length;
   ActivityType is_a();

   //state management
    boolean getstate(out stateType state);
    boolean init();
    boolean start();
    boolean stop();
    boolean terminate();
    boolean reset();
    boolean cancel();
    boolean resume();

   //actor management
    boolean getActor(out ActorInterface::Actor actor);
    boolean setActor(in ActorInterface::Actor actor);
    boolean getName(out string name);
    boolean setName(in string name);

   //ID
    boolean getID(out string id);
    boolean setID(in string name);

   //hierarchy (parents only)
    boolean getParentActivity(out BasicActivity activity );
    boolean setParentActivity(in BasicActivity activity);

   //resource management
    boolean getResources( out ResourceList resources);
    boolean setResourceList(in ResourceList resources, in long len);
    boolean getResource(in string id,
                        out ActorInterface::BasicResourceAdapter bra);
    boolean addResource(in ActorInterface::BasicResourceAdapter bra);
    boolean removeResource(in string id);
    boolean testResource(in string id);

   //persistency
    boolean saveActivity();
    boolean loadActivity(in ToplevelActivity top);

   boolean receiveMessage( in any msg);
   boolean sendMessage( in CORBA::Object receiver, any msg );
};

//*****************************************************************
//                     PrimitiveActivity
//*****************************************************************
```

```
interface PrimitiveActivity:Activity { };

//*****************************************************************
//                       BasicActivity
//*****************************************************************
interface BasicActivity:Activity {

   //****** Attributes ******

    attribute unsigned long childActivities_length;
    attribute unsigned long childActivities_free;

   //****** Methods *********

    // hierarchy (children)
    // parents are already implmented via Activity interface
    boolean getChildActivities(out ActivityList activities);
    boolean getChildActivity(out Activity activity, in string id);
    boolean addChildActivity(in Activity childActivity);
    boolean removeChildActivity(in string id);
    boolean testChildActivity(in string id);
};

//*****************************************************************
//                       ToplevelAktivity
//*****************************************************************
interface ToplevelActivity : BasicActivity {

   //****** Attributes ******

    attribute unsigned long activities_length;
    attribute unsigned long activities_free;
    attribute unsigned long actual_activities_length;

   //****** Methods *********

    boolean showAllActivities (out ActivityList activities);
    boolean getActivity(out Activity activity, in string id);
    boolean removeActivity(in string id);
    boolean addActivity(in Activity activity);
    boolean testActivity(in string id);
};

//*****************************************************************
//                       Workflow Activity
//*****************************************************************
interface WorkflowActivity : BasicActivity {
   boolean SetProcessDefinition( in any Process );
   void getProcessDefinition(out any Process);
};

//*****************************************************************
//                       Groupware Activity
//*****************************************************************
interface GroupwareActivity : BasicActivity {
//Die Schnittstelle der GroupwareActivity ist bereits komplett in der BasicActivity vorhanden
};

//*****************************************************************
//                       Designflow Activity
//*****************************************************************
interface DesignflowActivity : BasicActivity {
   attribute ConstraintList constraints;
   attribute DesignPrimitiveList designPrimitives;
   Constraint createConstraint(in string name,in ObjectSeq designItems);
   void removeConstraint(in Constraint c);
};
/*****************************************************************
*                         Manager
*
* Der Manager ist die zentrale Komponente, die zum Ändern
* des Zustandes verwendet werden muss. Er prüft das
* jeweilige Ereignis zunächst bei der EventEngine
* (requestEvent), ruft dann die entsprechende Mehtode der
* Aktivität auf und meldet der EventEngine am Ende, dass
* das Ereignis eingetreten ist.
*****************************************************************/
interface Manager {
   boolean init(in Activity activity);
   boolean start(in Activity activity);
   boolean stop(in Activity activity);
   boolean terminate(in Activity activity);
   boolean reset(in Activity activity);
   boolean cancel(in Activity activity);
   boolean resume(in Activity activity);
};
```

```
//*****************************************************************
//**                                                             **
//**                    Factories                                **
//**                                                             **
//*****************************************************************

//*****************************************************************
// ActivityObjectFactory
//*****************************************************************
interface ActivityObjectFactory {
   Activity createActivity(in BasicActivity parent,
   in ActorInterface::Actor actor,
   in string name);
   Activity getActivity(in string id);
};

//*****************************************************************
// PrimitiveActivityObjectFactory
//*****************************************************************
interface PrimitiveActivityObjectFactory : ActivityObjectFactory {

    PrimitiveActivity create_PrimitiveActivity_object(in string name,
                           in string id,
                           in ActorInterface::Actor actor,
                           in BasicActivity parent,
                           in ResourceList resources,
                           in unsigned long resources_len);

   PrimitiveActivity create_empty_PrimitiveActivity_object(in string id);
};

//*****************************************************************
// BasicActivityObjectFactory
//*****************************************************************
interface BasicActivityObjectFactory : ActivityObjectFactory {

    BasicActivity create_BasicActivity_object(in string name,
                           in string id,
                           in ActorInterface::Actor actor,
                           in BasicActivity parent,
                           in ResourceList resources,
                           in unsigned long resources_len,
                           in ActivityList childs,
                           in unsigned long childs_len);

    BasicActivity create_empty_BasicActivity_object(in string id);
};

//*****************************************************************
// ToplevelActivityObjectFactory
//*****************************************************************
interface ToplevelActivityObjectFactory : ActivityObjectFactory {

    ToplevelActivity create_ToplevelActivity_object(in string name,
                           in string id,
                           in ActorInterface::Actor actor,
                           in ResourceList resources,
                           in unsigned long resources_len,
                           in ActivityList childs,
                           in unsigned long childs_len,
                           in ActivityList activities,
                           in unsigned long activities_len);

   ToplevelActivity create_empty_ToplevelActivity_object(in string id);
};

//*****************************************************************
// WorkflowActivityObjectFactory
//*****************************************************************
interface WorkflowActivityObjectFactory : ActivityObjectFactory {

    WorkflowActivity create_WorkflowActivity_object(in string name,
                           in string id,
                           in ActorInterface::Actor actor,
                           in ResourceList resources,
                           in unsigned long resources_len,
                           in ActivityList childs,
                           in unsigned long childs_len,
                           in ActivityList activities,
                           in unsigned long activities_len);

    WorkflowActivity create_empty_WorkflowActivity_object(in string id);
};

//*****************************************************************
// DesignflowActivityObjectFactory
//*****************************************************************
```

```
interface DesignFlowActivityObjectFactory : ActivityObjectFactory {
   DesignFlowActivity createDesignFlowActivity(
      in string name,
      in string id,
      in ActorInterface::Actor actor,
      in ActivityInterface::BasicActivity parent,
      in ActorInterface::BasicResourceAdapterList resources);
   DesignFlowActivity createEmptyDesignFlowActivity(in string id);
   StringSeq getConstraintNames();
};

};
```

Anhang E.3 EventEngine.idl

```
module EventEngine {

interface Engine;
interface Listener;
interface Manager;

struct NameValue {string name; string value};
typedef sequence<NameValue> NameValueSequence;

// Event
// =====
enum EventState { UNKNOWN, POSSIBLE, BLOCKED, IMPOSSIBLE, REQUIRED, REQUESTED, HAPPENED };
typedef sequence<EventState> EventStateList;

/* Steht dieses Zeichen am Anfang einer value eines
*  Parameters eines Events, dann wird diese value als
*  eine Variable interpretiert.*/
const char VARIABLE_PREFIX = '$';

/* Ein Ereignis hat einen Namen und eine Folge
*  von Parametern */
struct Event { string name; NameValueSequence parameters;};
typedef sequence<Event> EventList;

// Dependencies
// ============
enum DependencyOperator { AND_DEP, OR_DEP, SEQ_DEP, EVENT, COMPLEMENT, CONSTANT };

/* Eine DependencyDefintion ist ein Ausdruck der die
*  gültigen Folgen von Events einschränkt. */
union DependencyDefinition switch (DependencyOperator) {
    case AND_DEP:
    case OR_DEP:
    case SEQ_DEP: sequence<DependencyDefinition> operands;
    case EVENT:
    case COMPLEMENT: Event event;
    case CONSTANT: boolean value;
};

/* Eine Dependency ist eine bei der EventEngine registrierte DependencyDefinition. Sie wird bei
*  eintreffenden Events (query, request, signal) berücksichtigt.*/
typedef long long Dependency;
typedef sequence<Dependency> DependencyList;

// Engine
// ======
exception DependencyViolation { DependencyList dependencies; };

/* Die Schnittstelle der EventEngine. Sie legt fest, wie Ereignissen ausgelöst und Abhängigkeiten
*  eingefügt und wieder entfernt werden. */
interface Engine {
   /* Erfragt den Zustand (EventState) des Ereignisses.*/
   EventState queryEvent(in Event e);
   EventState queryEvent2(in Event e, out Dependency blockingDependency);
   /* Fordert ein Ereignis an und blockiert es damit für andere.*/
   boolean requestEvent(in Event e);
   boolean requestEvent2(in Event e, out Dependency blockingDependency);
   boolean cancelRequestEvent(in Event e);
   /* Das Ereignis gilt mit dem Aufruf dieser Methode als eingetreten, auch wenn es nicht hätte
    * eintreten sollen. Die Exception wird erst nach dem Einarbeiten des Events geschmissen. */
   void signalEvent(in Event e) raises (DependencyViolation);
   /* Hier können sich Listener registrieren lassen, die dann nach dem Eintreten des Ereignisses
   * informiert werden (notification). Die Listener werden nicht persistent gespeichert.*/
   void addEventListener (in Event pattern, in EventStateList states, in Listener listener);
   void removeEventListener(in Event e, in Listener l);
   /* Die Methode nimmt die DependencyDefinition in die Menge der zu beachtenden Regeln auf.*/
   Dependency createDependency(in DependencyDefinition definition);
   boolean removeDependency(in Dependency dependency);
   /*Gibt eine Liste aller aktuell registrierten Dependencies zurück.*/
   DependencyList dependencies();
```

```
   /* Hiemit kann die Definition einer registrierten Dependency abgefragt werden.*/
   DependencyDefinition definition (in Dependency dependency);
   /* Die Methode liefert den Rest einer registrierten Dependency zurück. Dieser Rest ist aus der
    * Definition der Dependency im Laufe der eingetroffenen Events entstanden und gibt den
    * noch zu betrachtenden Teil an.*/
   DependencyDefinition residual(in Dependency dependency);
   /* Gibt true zurück wenn die Dependency erfüllt ist.*/
   boolean isSatisfied(in Dependency dependency);
   /* Gibt true zurück wenn die Dependency nicht mehr erfüllbar ist.*/
   boolean isUnsatisfiable(in Dependency dependency );
};

/* Ist ein Listener bei der EventEngine regisitriert, wird er über das Eintreten von Ereignissen
*  informiert (callback).*/
interface Listener {
   void event(in Event event, in EventState state);
};
/*****************************************************************/
*                          Manager
*
* Der Manager ist die zentrale Komponente, die zum Ändern
* des Zustandes verwendet werden muss. Er prüft das
* jeweilige Ereignis zunächst bei der EventEngine
* (requestEvent), ruft dann die entsprechende Mehtode der
* Aktivität auf und meldet der EventEngine am Ende, dass
* das Ereignis eingetreten ist.
*****************************************************************/
interface Manager {
   boolean init(in ActivityInterface::Activity activity);
   boolean start(in ActivityInterface::Activity activity);
   boolean stop(in ActivityInterface::Activity activity);
   boolean terminate(in ActivityInterface::Activity activity);
   boolean reset(in ActivityInterface::Activity activity);
   boolean cancel(in ActivityInterface::Activity activity);
   boolean resume(in ActivityInterface::Activity activity);
};
};
```

Anhang F: IDL: Workflow Facility

Anhang F.1 BoWorkflow.idl

```
module BoWorkflow {
typedef string WfName;
typedef sequence<WfName> WfNameList;
typedef string State;
typedef sequence<State> StateList;
typedef string WfKey;

// typedef CosNaming :: Name WfMKey;
enum element_type{process_instance, activity_instance, work_item};
struct DataElement {
   WfName name;
   any value;
};
typedef sequence<DataElement> DataElementList;

struct Filter {
   long type;
   string filter_specification;
};

struct Transition {
   WfName name;
   State start;
   StateList end;
   boolean controllable;
};
typedef sequence<Transition> TransitionList;

//not used
struct EventProperty {
   WfName name;
   any value;
};
typedef sequence<EventProperty> EventPropertyList;

struct WfEvent {
   long event_code;
   string event_type;
   WfKey producer_key;
   WfName producer_name;
   TimeBase :: UtcT timestamp;
   WfKey responsible;
   EventPropertyList event_header;
   EventPropertyList event_header_extension;
   EventPropertyList event_data;
   any event_body;
};
typedef sequence<WfEvent> WfEventList;


// Forward Deklarations
interface WorkflowManagerFactory;
interface ProcessDefinition;
interface Participant;
typedef sequence<Participant> ParticipantList;
interface WorkflowManager;
interface DefinitionIterator;
interface ExecutionElement;
interface ElementIterator;
interface ProcessData;
interface ProcessInstance;
interface ActivityInstance;
typedef sequence<ActivityInstance> ActivityInstanceList;
interface WorkItem;
typedef sequence<WorkItem> WorkItemList;

// Exception definitions
exception NoLogon {};
exception InvalidFilter{Filter filter;};
exception StateChangeNotAllowed{State state;};
exception StateChangeNotPerformed{State state;};
exception InvalidState{State state;};
exception InvalidKey{WfKey key;};
exception TransitionNotAllowed{WfName transition_name;};
exception TransitionNotPerformed{WfName transition_name;};
exception InvalidTransition{WfName transition_name;};
exception DataElementAssignmentFailed{ WfNameList data_elements;};
exception InvalidType{ DataElementList data_elements;};
```

```
exception ParticipantAssignmentFailed{Participant participant;};
exception InvalidName{WfNameList names;};
exception HistoryNotAvailable{};
exception UpdateParticipantNotAllowed{ParticipantList participants;};
exception DataElementUndefined{DataElementList data_elements;};

interface WorkflowManagerFactory {
   WorkflowManager open(in string srv, in string db, in string user, in string password )
      raises (NoLogon);
   void close(in WorkflowManager wfm);
};

interface ExecutionElement {
   attribute WfName name;
   attribute unsigned long priority;
   readonly attribute WfKey key;
   attribute string description;
   readonly attribute ProcessData input;
   readonly attribute ProcessData output;

   void destroy() raises(TransitionNotAllowed);
   StateList list_valid_states();
   TransitionList list_valid_transitions();
   State get_state();
   void change_state(in State new_state) raises(InvalidState, StateChangeNotAllowed,
      StateChangeNotPerformed);
   State make_transition (in WfName transition_name) raises (InvalidTransition,
      TransitionNotAllowed, TransitionNotPerformed);
//    HistoryIterator get_history(in Filter filter) raises(HistoryNotAvailable);
   ParticipantList get_responsibles();
   void set_responsibles(in ParticipantList responsibles) raises(UpdateParticipantNotAllowed,
      ParticipantAssignmentFailed);
   ParticipantList get_performers();
   void set_performers(in ParticipantList performers) raises(UpdateParticipantNotAllowed,
      ParticipantAssignmentFailed);
};

typedef sequence<ExecutionElement> ExecutionElementList;

interface WorkflowManager {
   readonly attribute WfMKey key;
   ElementIterator list_elements(in element_type type, in Filter filter) raises (InvalidFilter);
   ExecutionElement get_element(in element_type type, in WfKey key) raises (InvalidKey);
// HistoryIterator list_history(in element_type type, in Filter filter) raises (InvalidFilter);
   DefinitionIterator list_definitions(in Filter filter) raises (InvalidFilter);
   ProcessDefinition get_definition(in WfKey key) raises (InvalidKey);
};

interface ProcessDefinition /*: CosLifeCycle :: GenericFactory*/ {
   attribute WfName name;
   attribute string description;
   readonly attribute WfKey key;
   ProcessInstance create_process_instance(in WfName instance_name) raises(InvalidName);
// ProcessInstance create_sub_process_instance(in WfName instance_name,
// in WfMKey parent_workflow_manager in WfKey parent) raises(InvalidName);
   ElementIterator list_process_instances(in Filter filter) raises(InvalidFilter);
};
typedef sequence<ProcessDefinition> ProcessDefinitionList;


interface DefinitionIterator {
   readonly attribute Filter filter;
   boolean get_next(out ProcessDefinition definition);
   boolean get_next_n(in long how_many, out ProcessDefinitionList definitions);
   void dispose();
};

interface ElementIterator {
   readonly attribute Filter filter;
   readonly attribute element_type type;
   boolean get_next(out ExecutionElement element);
   boolean get_next_n(in long how_many, out ExecutionElementList elements);
   void dispose();
};

interface ProcessData {
   attribute WfName name;
   DataElement get_data_element(in WfName name)
      raises (InvalidName, DataElementUndefined);
   DataElementList get_data_element_list(in WfNameList name_list)
      raises (InvalidName, DataElementUndefined);
   void assign_data_element(in DataElement data_element)
      raises (InvalidName, InvalidType, DataElementAssignmentFailed);
   void assign_data_element_list(in DataElementList data_element_list)
      raises (InvalidName, InvalidType, DataElementAssignmentFailed);
};
```

```
interface ProcessInstance : ExecutionElement {
   readonly attribute ProcessDefinition implements;
   readonly attribute WfKey realizes_work_item;
   readonly attribute WfKey realizes_activity_instance;

// ActivityInstanceList list_activities(in Filter filter);
// HistoryIterator list_full_history(in Filter filter) raises(HistoryNotAvailable,
// InvalidFilter);
   void start () raises(TransitionNotAllowed, TransitionNotPerformed);
   void terminate() raises(TransitionNotAllowed, TransitionNotPerformed);
   void abort() raises(TransitionNotAllowed, TransitionNotPerformed);
};
typedef sequence<ProcessInstance> ProcessInstanceList;

interface ActivityInstance : ExecutionElement {
   readonly attribute ProcessInstance contained_in;
   WorkItemList list_work_items(in Filter filter) raises(InvalidFilter);
};

interface WorkItem : ExecutionElement {
   readonly attribute WfKey realized_by;
   readonly attribute ActivityInstance contained_in;
// void reassign (in Participant new_owner) raises (ParticipantAssignmentFailed);
   string start() raises(TransitionNotAllowed, TransitionNotPerformed);
   void complete() raises(TransitionNotAllowed, TransitionNotPerformed);
};

interface Participant {
   readonly attribute WfKey identification;
   attribute WfName name;
};
};
```

Anhang G: IDL: CSCW-Facility (Ausschnitte)

Anhang G.1 CSCW.idl:

```
// Das Modul beinhaltet die Schnittstellendefinitionen der Objekte des
// CSCW Facility Objekt Modells.
//
// Diese sind:
// - User   : Definition von Benutzern der CSCW Facility
// - Group    : Definition von Gruppen
// - Role   : Definition von Rollen der Benutzer innerhalb der Gruppen
// - Property : Definition von Rechten/Eigenschaften, die mit den Rollen verbunden sind
// - Workspace: Definition von Arbeitsbereichen
// - MultiUserTask : Definition von Tasks
// - Basic Resource Adapter : Oberklasse zur Einbindung von Ressourcen in die Facility
//
// Weitere Schnittstellen:
// - MessageQueue : Nachrichtenwarteschlange

// Vorwärtsdefinitionen der einzelnen Interfaces
interface User;
interface Role;
interface Property;
interface Workspace;
interface Group;
interface BasicResourceAdapter;
interface MultiUserTask;
interface MessageQueue;

// Definitionen von Objekt-Listen
typedef sequence<User> UserList;
typedef sequence<Role> RoleList;
typedef sequence<Property> PropertyList;
typedef sequence<Workspace> WorkspaceList;
typedef sequence<Group>GroupList;
typedef sequence<BasicResourceAdapter> BasicResourceAdapterList;
typedef sequence<MultiUserTask> MultiUserTaskList;

//
// Weitere Datentypen
//

// Nachricht als Folge von Strings
typedef sequence<string> Message;

// Informationen über den Anmeldestaus des Benutzers an der CSCW Facility
struct UserInfo { string userid;// User-ID
        boolean participate; };// TRUE : User angemeldet, FALSE: sonst

// Angaben über einzelne Applikationen

struct AppInfo  { string name;// Applikationsname
       boolean client_app;// TRUE  : Client-Applikation
              // FALSE : Server-Applikation
      };

// Listen der einzelnen Objekt-Identifikatoren
typedef sequence<UserInfo> UserIDList;
typedef sequence<string> UserNameList;
typedef sequence<string> GroupIDList;
typedef sequence<string> RoleNameList;
typedef sequence<string> PropertyNameList;
typedef sequence<string> WorkspaceIDList;
typedef sequence<string> TaskIDList;

typedef sequence<AppInfo> ApplicationNameList;


// Exceptions
exception FacilityException {};


//
// Definitionen der einzelnen Interfaces
//

/*************************    Group     **********************************/
interface Group {
// Attribute
   attribute string Groupname;// Gruppenname
   readonly attribute string GroupID;// Gruppen-ID
   attribute string Description;// Allg. Beschreibung der Gruppe z.b. Gruppenziel
```

```
   boolean SaveGroupData();// Speichern des Gruppenzustandes
// Beziehungen zur Group-Klasse
   boolean is_SubGroup(in Group grp); //Überprüft, ob die Gruppe eine Untergruppe ist

// Beziehungen zur User-Klasse

   // Aufnahme eines neuen Gruppenmitglieder
   boolean AddGroupmember(in User usr, in unsigned long mode);
   // Entfernen eines Gruppenmitglieser
   boolean RemoveGroupmember(in string userid, in unsigned long mode);
   // Auflisten der Gruppenmitglieder
   boolean ListGroupmembers(out UserList members);
// Überprüft, ob der angegeben User ein Gruppenmitglied ist
   boolean is_Groupmember(in string userid);
   // Speichern der Gruppenmitglieder
      boolean SaveGroupmembers();
   // Lesen der Gruppenmitglieder aus einer, mit SaveGroupMembers erstellten Sicherung
   boolean ReadGroupmembers(out UserIDList uidl);
   // Liefert den User, der innerhalb der Gruppe eine bestimmte Role besitzt
   UserGetRoleOwner(in string rolename);
   // Entfernt alle Gruppenmitglieder
   boolean RemoveAllGroupmembers();

// Beziehungen zur Role-Klasse

   // Ordnet dem User eine neue Rolle innerhalb der Gruppe zu
   boolean AddRoleToGroupmember(in string userid, in Role r, in unsigned long mode);
   // Entzieht dem User eine Rolle innerhalb der Gruppe
   boolean RemoveRoleFromGroupmember(in string userid, in string rolename, in unsigned long mo-
de);
   // Auflisten der Rollen eines Gruppenmitgliedes
   boolean ListGroupmemberRoles(in string userid, out RoleList rl);

   // Speichern der Rollen der Gruppenmitgliedes
   boolean SaveGroupmemberRoles(in string userid);
   // Lesen der Rollen des Gruppenmitgliedes aus einer, mit
   // SaveGroupmemberRoles erstellten Sicherung
   boolean ReadGroupmemberRoles(in string userid, out RoleNameList rnl);

// Beziehungen zur Workspace-Klasse

   // Zuweisen eines Workspaces an die Gruppe
   boolean AddWorkspace( in Workspace ws, in unsigned long mode);
   // Aufheben der Zuordnung eines Workspaces an die Gruppe
   boolean RemoveWorkspace (in string workspaceid, in unsigned long mode);
   // Aufheben aller Workspacezuordnungen
   boolean RemoveAllWorkspaces();
   // Auflisten der zugeordneten Workspaces
   boolean ListWorkspaces(out WorkspaceList workspaces);
   // Speichern der zugeordneten Workspaces
   boolean SaveWorkspaces();
   // Lesen der zugeordneten Workspaces aus einer, mit SaveWorkspaces
   // erstellten Sicherung
   boolean ReadWorkspaces(out WorkspaceIDList wsidlist);

   // Hilfsfunktion zur Ausgabe der Gruppen-Informationen
   void dump();

   // Löschen der Sicherungsdateien beim Entfernen des Group-Objektes
   void removeFiles();
}; // Group-Interface


// Iterator zum Auflisten der im Group Management Service definierten Gruppen

interface GroupIterator {
   boolean next_one(out Group g);
   boolean next_n(in unsigned long how_many, out GroupList gl);
   void destroy();
};


// Object Factory für Group-Objekte
interface GroupObjectFactory {
   Group create_group_object(in string groupname,in string groupid);
};

/*****************************   Property   ******************************/
interface Property {
   readonly attribute string Propertyname;// Bezeichner des Properties

// Beziehungen zur Role-Klasse

   // Zuweisen des Properties an eine Rolle
   boolean AddRole(in Role u, in unsigned long mode);
   // Entziehen des Properties von einer Rolle
   boolean RemoveRole(in string rolename, in unsigned long mode);
```

```
   // Aufheben der Zuordnungen des Properties an alle Rollen
   boolean RemoveAllRoles();
   // Auflisten der Rollen, denen das Property zugeordnet ist
   voidListRoles(out RoleList ul);
   // Speichern der Rollen, denen das Property zugeordnet ist
   boolean SaveRoles();
   // Lesen der mit SaveRoles gesicherten Rollen
   boolean ReadRoles(out RoleNameList unl);

   // Ausgeben der Objektdaten
   void dump();

   // Löschen der Sicherungsdateien
   void removeFiles();
};


interface PropertyIterator {
   boolean next_one(out Property p);
   boolean next_n(in unsigned long how_many, out PropertyList pl);
   void destroy();
};



interface PropertyObjectFactory {
   Property create_property_object(in string propertyname);
};

interface Role {
   readonly attribute string Rolename;

   /*********************************************************/
   boolean AddUserRole(in User u, in Group grp, in unsigned long mode);
   boolean RemoveUserRole(in string userid, in string groupid, in unsigned long mode);
   boolean RemoveAllUser();
   boolean SaveUser();
   boolean ReadUser(out UserNameList unl);

   boolean SaveGroups(in string userid);
   boolean ReadGroups(in string userid, out GroupIDList groups);

   /*********************************************************/
   boolean AddProperty(in Property p, in unsigned long mode);
   boolean RemoveProperty(in string Propertyname, in unsigned long mode);
   boolean RemoveAllProperties();
   boolean ListProperties(out PropertyList pl);
   boolean SaveProperties();
   boolean ReadProperties(out PropertyNameList unl);
   boolean HasProperty(in string propertyname);

   void dump();

   void removeFiles();
};


interface RoleIterator {
   boolean next_one(out Role r);
   boolean next_n(in unsigned long how_many, out RoleList rl);
   void destroy();
};

interface RoleObjectFactory {
   Role create_role_object(in string rolename);
};


interface MessageQueue {
   boolean SendMessage(in Message msg, in boolean auto_save);
   boolean ReceiveMessage(out Message msg);
   boolean SaveMessages();
   boolean ReadMessages();

   void dumpMessageQueue();
};

interface User {
   /*********************************************************/
   /* Gibt an, ob der User momentan an der Facility */
   /* angemeldet ist             */
   /*********************************************************/
      enum ConnectionState { Connected, Disconnected };

   /*********************************************************/
   /*       User-Daten                    */
   /*********************************************************/
```

```
            attribute stringName;
   readonlyattribute stringUserID;
   readonlyattribute ConnectionStateConnState;

   boolean ChangePassword(in string OldPwd, in string NewPwd);
   boolean CheckPassword(in string Pwd);

   stringConnect() ;
   voidDisconnect();

   boolean SaveUserData();
   boolean SendMessage(in Message msg);
   boolean ReceiveMessage(out Message msg);

   // Manipulation der Beziehungen zur Role-Klasse
   booleanAddRole(in Role r, in string groupid, in unsigned long mode);
   booleanRemoveRole(in string rolename, in string groupid, in unsigned long mode);
   boolean SaveRoles(in string groupid);
   boolean ReadRoles(in string groupid,out RoleNameList rnl);
   booleanListRoles(in string groupid, out RoleList rl);
   voiddumpRoles();

   booleanAddGroup(in Group g, in unsigned long mode);
   booleanRemoveGroup(in string groupid, in unsigned long mode);
   booleanRemoveAllGroups();
   boolean SaveGroups();
   boolean ReadGroups(out GroupIDList gidl);
   booleanListGroups(out GroupList gl);

   boolean AssignTask(in MultiUserTask task, in unsigned long mode);
   boolean RemoveTask(in string taskid, in unsigned long mode);
   boolean RemoveAllTasks();
   boolean SaveAssignedTasks();
   boolean ReadAssignedTasks(out TaskIDList tasks, out WorkspaceIDList workspaces);
   boolean ListAssignedTasks(out MultiUserTaskList tasks);

   void removeFiles();
};

interface UserIterator {
   boolean next_one(out User us);
   boolean next_n(in unsigned long how_many, out UserList ul);
   void destroy();
};

interface UserObjectFactory {
   User create_user_object(in string name, in string userid, in string pwd);
};

interface BasicResourceAdapter {
   enum ResourceType {  DOCUMENT,                    PERSISTENT_OBJECT,
CLIENT_APPLICATION,
SERVER_APPLICATION,  EVENT_CHANNEL,       REALTIME_CHANNEL,    SERVICE };
   enum ResourceAdapterType { TASK_RA,       APPLICATION_RA,      DATA_RA,
SERVICE_RA,          SYSTEM_RA};

   string   GetResourceName();
      ResourceTypeGetResourceType();
   ResourceAdapterType GetResourceAdapterType();
   boolean  AssignResourceObject(in Object obj);
   Object   GetRessourceObject();


   boolean  AssignWorkspace(in Workspace ws, in unsigned long mode);
   boolean  Move(in Workspace ws);
   WorkspaceGetWorkspace();
   string   GetWorkspaceID();

   boolean  SaveAdapterInformation(in ResourceAdapterType adapter_type);
   boolean  AddTask(in MultiUserTask task, in unsigned long mode);
   boolean  RemoveTask(in string taskid, in unsigned long mode);

   void removeFiles();
};


interface TaskResourceAdapter : BasicResourceAdapter {
   MultiUserTask GetTask();
   string GetTaskID();
};

interface ApplicationResourceAdapter : TaskResourceAdapter {
   string GetUserID();
   User   GetUser();
   boolean   SetUser(in User usr);
};
```

```
interface GeneralResourceAdapter: BasicResourceAdapter {

   enum MigrationPolicy{No_Migration, Upward_Migration, Downward_Migration, Migration };

   readonly attribute boolean MultipleTasks;
   readonly attribute MigrationPolicy Policy;

   void GetTaskList(out MultiUserTaskList tasks);
   void GetTaskIDList(out TaskIDList tasks);
};

interface SystemResourceAdapter : GeneralResourceAdapter {};

interface DataResourceAdapter : GeneralResourceAdapter {};

interface ServiceResourceAdapter : GeneralResourceAdapter {};

typedef sequence<BasicResourceAdapter::ResourceType> ResourceTypeList;

struct ResourceAdapterInformationElement { string resourcename;
            BasicResourceAdapter::ResourceType resource_type; } ;

typedef sequence <ResourceAdapterInformationElement> ResourceAdapterInformationList;

interface ResourceAdapterFactory {
   TaskResourceAdapter create_task_adapter(
                        in stringname,
                        in BasicResourceAdapter::ResourceType  type,
                        in Workspacews,
                        in MultiUserTask task,
                        in Objectobj);
   ApplicationResourceAdapter create_application_adapter(
                        in stringname,
                        in BasicResourceAdapter::ResourceType  type,
                        in Workspacews,
                        in MultiUserTask task,
                        in Objectobj,
                        in Userusr);
   DataResourceAdapter create_data_adapter(
                        in string name,
                        in BasicResourceAdapter::ResourceType  type,
                        in boolean multipletasks,
                        in GeneralResourceAdapter::MigrationPolicy policy,
                        in Workspace ws,
                        in Object    obj) ;
   SystemResourceAdapter create_system_adapter(
                        in string name,
                        in BasicResourceAdapter::ResourceType  type,
                        in boolean multipletasks,
                        in GeneralResourceAdapter::MigrationPolicy policy,
                        in Workspace ws,
                        in Object    obj) ;
   ServiceResourceAdapter create_service_adapter(
                        in string name,
                        in BasicResourceAdapter::ResourceType  type,
                        in boolean multipletasks,
                        in GeneralResourceAdapter::MigrationPolicy policy,
                        in Workspace ws,
                        in Object    obj) ;
};


interface MultiUserTask {

   enum TaskState {NotStarted, Running,   Suspended,        Terminated };
   enum TaskAccessPolicy {Open,        Closed };

   readonly attribute stringTaskID;
readonly attribute TaskStateState;
   readonly attribute TaskAccessPolicy AccessPolicy;
   readonly attribute stringResponsibleUserID;
   readonly attribute UserResponsibleUser;
   readonly attribute stringTaskdescription;
   readonly attribute stringWorkspaceID;
   readonly attribute WorkspaceAssignedWorkspace;
   readonly attribute booleanSupportSuspend;

   boolean AssignNewResponsibleUser(in User usr, in boolean auto_save);

   boolean AssignWorkspace(in Workspace ws);
   boolean Move(in Workspace ws);
   boolean SaveTaskData();

   boolean AssignResourceAdapter(in BasicResourceAdapter adapter, in unsigned long mode, in boo-
lean auto_save);
   boolean RemoveResourceAdapter(in string resourcename,
```

```
                        in BasicResourceAdapter::ResourceType resource_type,
                        in unsigned long mode,
                        in boolean auto_save);
   void GetAssignedResourceAdapter(in string resourcename,
                            in BasicResourceAdapter::ResourceType resource_type,
                            out BasicResourceAdapterList adapter_list);
   void GetAllAssignedResourceAdapter ( out BasicResourceAdapterList adapter_list);
   BasicResourceAdapter DetermineResourceAdapter(in string resourcename,
                           in BasicResourceAdapter::ResourceType resource_type,
                           in boolean create_task_ra,
                           in boolean create_resource,
                           in string par1,
                           in string par2,
                           in string par3);
   boolean FreeAllResources();
   boolean ReadResourceAdapter(out ResourceAdapterInformationList adapters);

   boolean AssignUser(in User usr, in unsigned long mode, in boolean auto_save);
   boolean RemoveUser(in string userid, in unsigned long mode);
   voidGetAssignedUsers(out UserList users);
   boolean ReadAssignedUsers(out UserIDList users);
   boolean Join(in string userid,in boolean auto_save);
   boolean Leave(in string userid);
   boolean is_Participate(in string userid);
   boolean Connect(in string userid);
   boolean Disconnect(in string userid);
   boolean is_connected(in string userid);

   boolean AddApplication(in string application_name,
                    in string paramater1,
                    in string parameter2,
                    in string parameter3,
                    in boolean client_application,
                    in boolean auto_save);
   boolean StartApplication(in string application_name,in boolean auto_save);
   boolean SuspendApplication(in string application_name);
   boolean RestartApplication(in string application_name);
   boolean TerminateApplication(in string application_name);
   boolean RemoveApplication(in string application_name);
   voidListApplications(out ApplicationNameList apps);
   boolean RestoreApplications();

   boolean Start();
   boolean Suspend();
   boolean Restart();
   boolean Terminate();

   boolean ApplicationTerminated(in string application, in string userid);

   boolean SetCheckpoint(out unsigned long checkpoint_id);
   boolean RecoverToCheckpoint(in unsigned long checkpoint_id);

   void removeFiles();
};

interface Workspace {
   enum WorkspaceAccessPolicy { Open, Closed };
   enum MigrationDirection{ Upward, Downward };

   readonly attribute stringWorkspaceID;
   attribute stringWorkspaceName;
   attribute WorkspaceAccessPolicy  Policy;
   readonly attribute WorkspaceParentWorkspace;

   GroupUserGroup();
   stringUserGroupID();

   boolean ChangeParent(in Workspace parent);
   boolean AssignUserGroup(in Group grp, in unsigned long mode);
   boolean ReplaceUserGroup(in Group newgroup, in unsigned long mode);
   boolean RemoveUserGroup(in unsigned long mode);

   boolean AddWorkspaceUser(in User newUser);
   boolean RemoveWorkspaceUser(in string userid);
   boolean is_WorkspaceUser(in string userid);
   boolean EnterWorkspace(in User usr);
   boolean SaveWorkspaceData();
   WorkspaceCreateSubWorkspace(in string wsname, in WorkspaceAccessPolicy wspolicy,
         in boolean auto_save);
   booleanAddSubWorkspace(in Workspace ws);
   booleanMigrateSubWorkspace(in string ws, in Workspace target);
   booleanMigrateActualWorkspace (in Workspace ws, in MigrationDirection direction);
   booleanRemoveSubWorkspace(in string wsid, in boolean delete_sub);
   booleanRemoveSubWorkspaceEntry(in string wsid);
   WorkspaceLookupSubWorkspace(in string wsid, in unsigned long depth);
   void  ListSubWorkspaces(out WorkspaceList workspaces);
   booleanReadSubWorkspaces(out WorkspaceIDList workspaces);
```

```
   booleanLoadSubWorkspaces(in boolean auto_save, in boolean ref_actions);
   booleanis_SubWorkspace(in Workspace ws, in unsigned long level);
   booleanis_ParentWorkspace(in string wsid, in unsigned long level);
   booleanMerge      (in Workspace ws);

   boolean AddResourceAdapter(in BasicResourceAdapter adapter,in unsigned long mode);
   boolean RemoveResourceAdapter(in string resourcename,
      in BasicResourceAdapter::ResourceType resource_type,in unsigned long mode);
   BasicResourceAdapter LookupResourceAdapter(in string resourcename,
      in BasicResourceAdapter::ResourceType resource_type);
   void ListResourceAdapter (out BasicResourceAdapterList adapters);
   boolean SaveResourceAdapter();
   boolean ReadResourceAdapter(out ResourceAdapterInformationList list);

   MultiUserTask CreateTask(in MultiUserTask::TaskAccessPolicy taskpolicy,
      in User responsible_user, in string description);
   boolean AddTask  (in MultiUserTask task);
   boolean RemoveTask(in string taskid);
   MultiUserTaskLookupTask(in string taskid);
   boolean ListTasks(out MultiUserTaskList tasks);
   boolean SaveTasks();
   boolean LoadTasks(in boolean referential_actions);
   boolean ActiveTasks();

   voiddump();
   voidremoveFiles();
};

interface WorkspaceIterator {
   boolean next_one(out Workspace ws);
   boolean next_n(in unsigned long how_many, out WorkspaceList wl);
   void destroy();
};

interface WorkspaceObjectFactory {
   Workspace create_workspace_object(in string workspacename,in string workspaceid, in Workspace
parent,
                              in Workspace::WorkspaceAccessPolicy policy, in boolean auto_save);
};
```

Anhang G.2 CSCWInterfaces.idl

```
#include "CSCW.idl"

interface Application {
   boolean Start();
   boolean Terminate();
   boolean Suspend();
   boolean Restart();
   boolean SupportTaskInterface();
   boolean TaskOID(in MultiUserTask task);
};
```

Anhang G.3 CSCWAccessLayer.idl

```
#include "CSCW.idl"

interface CSCWAccessLayer {
   User Login(in string username,
              in string userid,
              in string password);
   boolean Logout(in string userid);
};
```

Anhang G.4 CSCWManagementServices.idl

```
// Das Modul beinhaltet die Schnittstellendefinitionen der Services
// CSCW Facility Management Service Ebene.
//
// Diese sind:
//
// - User Management Service : Verwaltung der User Objekte
// - Group Management Service :Verwaltung der Group-Objekte
// - Role Management Service :Verwaltung der Role-Objekte
// - Property : Management Service :Verwaltung der Property-Objekte
// - Workspace:Management Service :Verwaltung der Top Level Workspace-Objekte
// - Resource Management Service :  Verwaltung der Resource Adapter und
//             Erzeugung von Resourcen für best. Resource-Klassen

#include "CSCW.idl"
```

```
interface GroupManagementService {
   // Erzeugen einer neuen Gruppe
   GroupCreateGroup(in string name, in string id,  in boolean auto_save);
   // Entfernen einer Gruppe
   boolean RemoveGroup(in string id);
   // Suchen einer Gruppe an Hand der Gruppen-ID
   GroupLookupGroupByID(in string id);
   //Auflisten aller definierten Gruppen
   voidListGroups(in unsigned long how_many, out GroupList gl, out GroupIterator gi);
   // Sichern der Gruppen
   boolean Save();
   // Laden der Gruppen aus Sicherung
   boolean Load();
   // Laden einer einzelnen Gruppe
   GroupLoadGroup(in string id);
   // Hilfsfkt. zur Ausgabe der definierten Gruppen
   voiddumplist();
}; // GroupMgmtService


interface PropertyManagementService {
   // Erzeugen eines neuen Property-Objektes
   Property CreateProperty(in string propertyname, in boolean auto_save);
   // Entferne Property
   boolean RemoveProperty(in string propertyname);
   // Suche Property an Hand der Propertybezeichnung
   Property LookupProperty(in string propertyname);
   // Auflisten aller definierten Properties
   void ListProperties(in unsigned long how_many, out PropertyList rl, out PropertyIterator ri);
   // Sichern der Property-Objekte
   boolean Save();
   // Laden der Objekte aus Sicherung
   boolean Load();
   // Laden eines einzelnen Property-Objektes
   Property LoadProperty(in string propertyname);
   // Hilfsfkt. zur Ausgabe der definierten Properties
   void dumplist();
}; // PropertyMgmtService


interface ResourceManagementService {
   // Erzeugen eines neuen Resource Adapters
   BasicResourceAdapter CreateResourceAdapter(
                  in string resourcename,
                  in BasicResourceAdapter::ResourceType resource_type,
                  in boolean create_task_ra,
                  in Workspace ws,
                  in Object obj,
                  in boolean   create_resource,
                  in string    par1,
                  in string    par2,
                  in string    par3,
                  in User usr,
                  in boolean multipletasks,
                  in GeneralResourceAdapter::MigrationPolicy policy,
                  in boolean auto_save);

   // Entfernen eines Resource Adapters
   booleanRemoveResourceAdapter(in string resourcename, in   BasicResourceAdapter::ResourceType
resource_type);
   // Suchen eines Resource Adapters nach Resourcename und Resourcetyp
   BasicResourceAdapter LookupResourceAdapter (in string resourcename, in    BasicResourceAdap-
ter::ResourceType resource_type);
   // Auflisten aller Resource Adapter des gegebenen Typs
   boolean ListResourceAdaptersByType( in BasicResourceAdapter::ResourceType resource_type,
out BasicResourceAdapterList resourcelist);

   // Sichern der Resource Adapter
   boolean Save();
   // Laden der Resource Adapter aus Sicherung
   boolean Load();
   // Laden eines einzelnen Resource Adapters
   BasicResourceAdapter LoadAdapter(in string resourcename, in  BasicResourceAdapter::Resource-
Type resource_type);


   // Erzeugen einer Resourcen-Instanz über die entsprechende Factory
   Object CreateResource(in string resourcename,
                     in BasicResourceAdapter::ResourceType resource_type,
                     in string par1,
                     in string par2,
                     in string par3);

   // Registrieren einer neuen Object Factory für den angegebenen Resourcentypen unter dem an-
gegebenen Namen
   booleanAddResourceFactory( in BasicResourceAdapter::ResourceType resource_type, in   string
object_factory_name );
```

```
   // Löschen der Registrierung einer Resource Factory
   boolean RemoveResourceFactory(in BasicResourceAdapter::ResourceType resource_type);

   // Auflisten der Resourcetypen, für welche Object Factories beim Resource Management
   // Service registriert sind
   void ListSupportedResources(out ResourceTypeList resources);

   // Liefert den Registriernamen der Factory für den angegebenen Resourcentypen
   string GetFactory(in BasicResourceAdapter::ResourceType resource_type);

   // Sichern Registriernamen der Object Factories
   boolean SaveResourceFactories();

   // Laden der Registriernamen aus Sicherung
   boolean LoadResourceFactories();

   // Festlegen welcher Resourcentype über welche(n) Spezialisierungen der
   // Resource Adapter-Klasse eingebunden werden
   boolean AssignResourceType_To_AdapterType(
                in BasicResourceAdapter::ResourceAdapterType adapter_type,
                in BasicResourceAdapter::ResourceType         resource_type);

   // Sichern der Zuordnung Resource Adapter Type  ->  ResourceType
   boolean SaveRT_to_AT();
      // Laden der Sicherung
   boolean LoadRT_to_AT();

   // Hilfsfkt. zur Ausgabe der Resource Adapter
   void dumplist();
};

interface RoleManagementService {
   // Erzeuge neue Rolle
   RoleCreateRole(in string rolename, in boolean auto_save);
   // Entferne Rolle
   boolean RemoveRole(in string rolename);
   // Suche Rolle an Hand der Bezeichnung
   RoleLookupRole(in string rolename);
   // Auflisten der definierten Rollen
   voidListRoles(in unsigned long how_many, out RoleList rl, out RoleIterator ri);
   // Sichern der Rollen
   boolean Save();
   // Laden der Rollen aus Sicherung
   boolean Load();
   // Laden einer einzelnen Rolle
   RoleLoadRole(in string rolename);
   // Hilfsfkt. zur Ausgabe der definierten Rollen
   voiddumplist();
}; // RoleMgmtService

interface UserManagementService {
   // Erzeugene eines neuen User-Objektes
   UserCreateUser(in  string  username,  in  string  userid,  in  string  password,  in  boolean
auto_save);
   // Entfernen eines User-Objektes
   boolean RemoveUser(in string userid);
   // Suche User an Hand des Namens
   UserLookupUserByName(in string name);
   // Suche User an Hand der User-ID
   UserLookupUserByID(in string userid);
   // Auflisten aller definierten User
   voidListUser(in unsigned long how_many, out UserList ul, out UserIterator ui);
   // Sichern der User
   boolean Save();
   // Laden der User aus Sicherung
   boolean Load();
   // Lade einzelnen User aus Sicherung
   UserLoadUser(in string userid);
   // Hilfsfkt. zur Ausgabe der definierten User
   voiddumplist();
}; // UserMgmtService

interface WorkspaceManagementService {
   // Erzeuge neuen Top Level Workspace
   WorkspaceCreateWorkspace(in string wsid,
               in string wsname,
               in Workspace::WorkspaceAccessPolicy policy,
               in boolean auto_save);

   // Entferne Top-Level Workspace
   booleanRemoveWorkspace(in string wsid);
   // Suche Top-Level Workspace an Hand der Workspace-ID
   WorkspaceLookupWorkspace(in string wsid);
   // Auflisten aller Top-Level Workspaces
   void  ListWorkspaces(in unsigned long how_many,
               out WorkspaceList workspaces,
```

```
                out WorkspaceIterator iterator);

   // Sichern der Top-Level Workspaces
   booleanSave ();
   // Laden der Top-Level Workspaces
   booleanLoad ();
   // Lade einzelnen Top-Level Workspace
   WorkspaceLoadWorkspace(in string wsid);

   // Hikfsfkt. zur Ausgabe der Top-Level Workspaces
   void  dumplist();
}; // WorkspaceMgmtService
```