Ein Beitrag zur Realisierung von verteilten Workflow-Management-Systemen durch Entwicklung verfahrensunabhängiger Basiskomponenten

Zur Erlangung des akademischen Grades
eines
Doktors der Ingenieurwissenschaften
von der Fakultät für Maschinenbau der
Universität Karlsruhe

genehmigte
Dissertation

von
Dipl.-Inform. Andreas Schmidt
aus Karlsruhe

Tag der mündlichen Prüfung:	20. Januar 2000
Hauptreferent:	Prof. Dr. Georg Bretthauer
Korreferent:	Prof. Dr. Hartwig Steusloff

Kurzfassung

Workflowsysteme haben im administrativen Geschäfts– und Organisationsbereich und zum Teil auch im Produktionsbereich inzwischen eine weite Verbreitung gefunden. Charakteristische Merkmale der in diesem Bereich durchzuführenden Arbeitsabläufe sind hohe Fall-Stückzahlen, eine einheitliche Vorgehensweise bei der Bearbeitung, eine relativ einfache Struktur sowie keine oder nur seltene Abweichungen vom festgelegten Ablauf. Im Gegensatz dazu weisen Arbeitsabläufe im wissenschaftlich–technischen Umfeld sehr geringe Fall-Stückzahlen, hohe Änderungsraten, einen zumeist komplexeren Aufbau und weitaus flexiblere Ablaufstrukturen auf. Trotz dieser großen Unterschiede in den Abläufen gibt es eine Reihe von Gründen, die auch für den Einsatz von Workflowtechnologie im wissenschaftlich–technischen Umfeld sprechen.

Ziel dieser Disssertation war es, den Einsatz von Workflow–Technologien im wissenschaftlich–technischen Umfeld zu untersuchen und einen Software-Baukasten ($\mathcal{W}*FLOW$) zur Unterstützung des Aufbaus von workflow-basierten Arbeitsumgebungen im wissenschaftlich–technischen Bereich zu konzipieren und anschließend prototypisch zu realisieren.

Hierzu wurde ein komponentenorientierter Ansatz gewählt, der dem Entwickler von Workflowsystemen oder verteilten Arbeitsumgebungen eine Reihe von Basistechnologien aus den Bereichen Workflowsysteme, Applikations-Framework und Datenverwaltung zur Verfügung stellt. Als Bindeglied zwischen den Komponenten wurde eine Skriptsprache gewählt, was insbesondere unter dem Aspekt des *Rapid Application Development* und *Rapid Prototyping* entscheidende Vorteile bringt. Die Offenheit des Systems wurde durch die Verfügbarkeit der Skriptsprachen-Schnittstelle zur Laufzeit erreicht, wodurch eine sehr enge Ankopplung von Anwendungen, bzw. eine einfache Erweiterung eines mittels $\mathcal{W}*FLOW$ realisierten Anwendungssystems möglich wird.

Die Hauptkomponenten des Baukastens sind: *Container* zur Aufnahme, Verwaltung und dem Zugriff auf die Anwendungsdaten, *Aktivitäten* zum Aufbau und zur Steuerung der Arbeitsabläufe und die *Toolservices* zur Anbindung externer Tools.

Neuartig an dem hier vorgestellten Konzept ist vor allem die Integration der verschiedenen Technologien innerhalb eines Baukastens in Form von eigenständigen, mittels einer netzwerktransparenten Skriptsprachenschnittstelle miteinander kombinierbaren, Komponenten. Dies ermöglicht sowohl die Erstellung von flexiblen, leicht änderbaren Arbeitsumgebungen, als auch die Entwicklung kompletter Workflowsysteme auf Basis der vorgestellten Komponenten. Die in dieser Arbeit vorgestellte objektorientierte Interpretation des Aktivitätenbegriffes und die einheitliche Betrachtung interner Aktionen und externer Anwendungen (Methoden), welche auf privaten Ressourcen (Daten) arbeiten, erlauben es, die Aktivitäten als Ausgangspunkt für die Realisierung komplexer, anwendungsorientierter semantischer Transaktionsmodelle einzusetzen.

Abstract

Workflow systems have become widely used in business administration and organization. Characteristic features of workflow in this latter area are large numbers of jobs/pieces, one standard approach to processing, a relatively simple structure, and no, or only infrequent, deviations from the pattern defined. In contrast, workflows in scientific and technological environments are characterized by very small numbers of jobs/pieces, high rates of change, mostly rather complex structures, and by far more flexible arrangements. Despite these major differences in workflow, there are a number of reasons advocating the use of workflow technology also in scientific-technical environments.

This Ph.D. thesis was written to study the use of workflow technologies in a scientific-technical environment, and design, and subsequently implement in a prototype, a modular software system ($\mathcal{W}*FLOW$) supporting the buildup of workflow-based work environments in science and technology.

For this purpose, a component-oriented approach is adopted which offers developers of workflow systems or distributed working environments a number of basic technologies from the fields of workflow systems, application frameworks, and data management. The "glue" combining these components is a scripting language, which offers decisive advantages especially under the aspects of *rapid application development* and *rapid prototyping*. The openness of the system was achieved by the availability of the scripting language interface at runtime, which allows for very close coupling of applications and for easy expansion, respectively, of an application system implemented by means of $\mathcal{W}*FLOW$.

These are the main components of the modular system: *container* for accommodation and administration of, and access to, application data; *activities* for building up and controlling workflows; *toolservices* to connect external tools.

The key novel feature of the concept presented here is the integration of various technologies within one modular system in the form of stand-alone components which can be combined by means of a network-transparent scripting language interface. This allows both flexible, easy-to-change work environments to be established and complete workflow systems to be developed on the basis of the components presented. The object-oriented interpretation of the activities concept outlined in this thesis, and the uniform treatment of internal actions and external applications (methods) operating on private resources (data), allow activities to be used as a starting point for the implementation of complex, application-oriented semantic transaction models.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Bedeutung, Historie und Grundkomponenten von verteilten Workflow-Management-Systemen

Hauptaufgabe eines Workflowsystems ist die Koordination von Arbeitsschritten zwischen Benutzern und Systemkomponenten. Diese erfolgt entsprechend einer zuvor definierten Ausführungsanweisung und umfaßt die Zuordnung von einzelnen Arbeitsschritten/Programmen, Daten und Benutzern [HL91]. Ziel ist es dabei, nicht nur die Abarbeitung der einzelnen Arbeitsschritte, sondern vor allem auch die Koordinierung und Überprüfung gemäß Qualitätskriterien (z. B. im Sinne von ISO 9000ff) rechnerunterstützt zu optimieren.

Im folgenden wird zuerst ein kurzer Abriß über die Historie und die Herkunft von Workflow-Management-Systemen gegeben. Anschließend werden die grundlegende Architektur und das zugrundeliegende Workflow-Modell vorgestellt.

1.1.1 Historie und Überblick

Die Ursprünge von Workflowsystemen können in den Bereichen Büroautomatisierung, Dokumentenverwaltung und Groupeware Produkten gesehen werden [Hol94a, JB96, EN96]. Jeder dieser Bereiche enthält Funktionalitäten, die sich heute in den meisten Workflowsystemen wiederfinden. Im Bereich Büroautomatisierung sind das u. a. Arbeitsplanung, die Funktionsintegration, die Aufgabenverwaltung und die Assistenz durch das System. Eng damit verwandt ist auch der Bereich der elektronischen Dokumentenverwaltung. Aufgaben, wie sie auch im Bereich Workflow-Management vorkommen, sind die Verteilung von Dokumenten an unterschiedliche Personengruppen, die Verwaltung unterschiedlicher Zugriffsrechte und damit verbunden die Anwendung bestimmter Applikationen auf diese Dokumente. Aber auch erweiterte e-Mail Systeme weisen Funktionen auf, die in Workflowsystemen zu finden sind. Die Möglichkeit, Informationen über weitere Bearbeiter, an die eine Nachricht oder ein Dokument nach der Bearbeitung geschickt werden soll, zu integrieren, erlaubt es, Ausführungssequenzen, wie sie auch bei Workflowsytemen üblich sind, zu definieren. Weitere Ursprünge liegen in Groupeware Systemen, welche die Fähigkeit aufweisen, die Zusammenarbeit von Gruppen und Individuen zu unterstützen. Dies ist auch eine gewünschte Eigenschaft von Workflowsystemen.

Abbildung 1.1 zeigt die Einordnung einer Reihe von verbreiteten kommerziellen Workflowsystemen im Umfeld von ausprogrammierten (hardcodierten) Applikationen, Büro-, Groupeware- und Workflowsystemen. Man erkennt, daß die Workflowsysteme, bei gleicher Spezifität der Anwendung und Strukturierung der Arbeitsabläufe, in ihrem Einsatz weitaus flexibler sind als

hardcodierte Systeme. Analog unterscheiden sich die Workflowsysteme von den Groupewareprodukten hauptsächlich durch die stärkere Strukturierung der Abläufe, bei ähnlich starker Aufgabenteilung und hoher Flexibilität. Am anderen Ende des Spektrums liegen Bürosysteme, wie etwa *Office 97* von Microsoft, die sich durch eine geringe Aufgabenteilung, Flexibilität und Strukturierung auszeichnen.

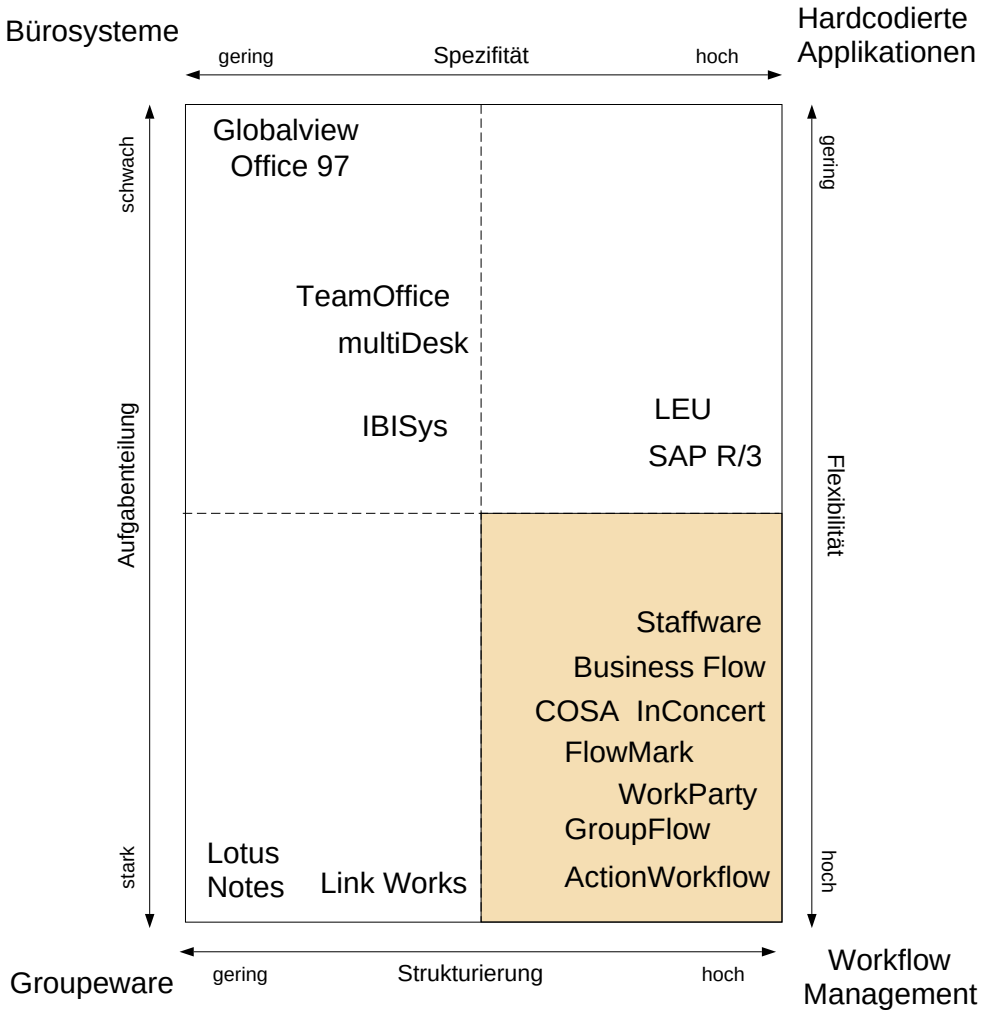


Abbildung 1.1: *Einordnung von kommerziellen Workflowsystemen in das Umfeld von Bürosystemen, hardcodierten Applikationen und Groupewaresystemen (aus [SM96])*

Hauptmerkmal eines Workflowsystems ist die Trennung von Applikation und Ablaufsteuerung. Diese Separation kann analog zur Trennung zwischen Applikation und Datenhaltungskomponente in Datenbankanwendungen gesehen werden und erlaubt bei der Entwicklung von Anwendungen die Konzentration auf die Applikationsaspekte [JB96]. Abbildung 1.2 zeigt, in Anlehnung an [RzM96], die Entwicklung des Aufbaus von Informationssystemen, ausgehend von einem monolithischen Block (a), hin zu den Systemen, die die Datenhaltung an ein externes Datenbanksystem deligieren (b), bzw. im Falle von Workflowsystemen, die Ablaufsteuerung von der eigentlichen Anwendung trennen (c). Als logische Konsequenz aus dieser Entwicklung ergibt sich für Workflowsysteme, als Anwendungen mit hohem Datenaufkommen, die Architektur (d), bei der sowohl die Datenhaltung, als auch die Ablaufsteuerung von der eigentlichen Anwendung getrennt sind.

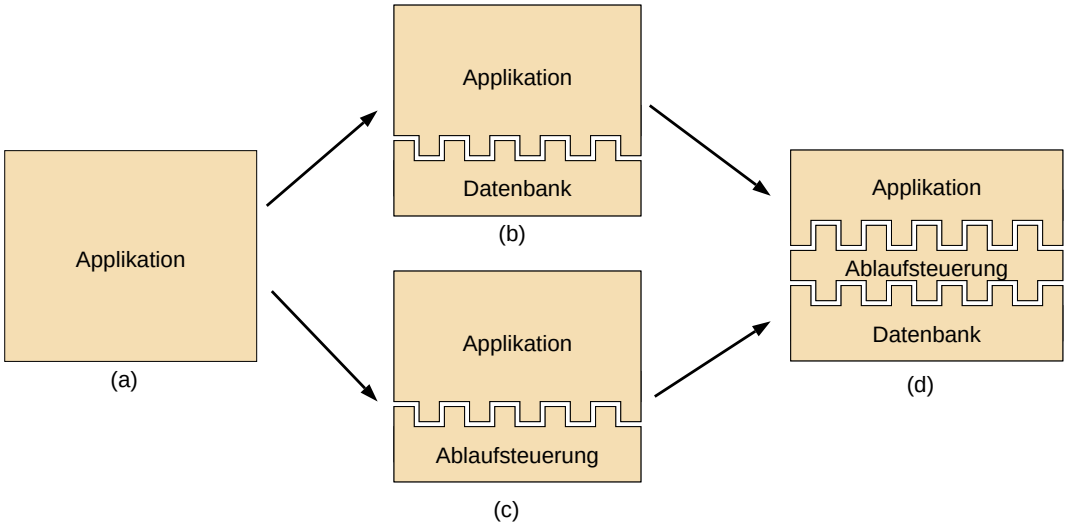


Abbildung 1.2: *Entwicklung des Aufbaus von Informationssystemen*

Workflow-Management-Systeme (WFMS) werden von vielen Experten als sehr erfolgversprechender Markt mit enorm hohem Wachstumspotential angesehen [JB96, Moh97, JBS97]. Aus diesem Grund ist es auch nicht verwunderlich, daß in den letzten Jahren eine Vielzahl von Systemen auf dem internationalen Markt erschienen sind, die für sich alle den Begriff "Workflow" in Anspruch nehmen[1]. Von dieser großen Anzahl an Systemen sind aber nur eine Handvoll als vollwertige Workflowsysteme zu bezeichnen [AS96], von denen die wichtigsten in Abschnitt 1.2.1 vorgestellt werden. Ein Hauptproblem der erhältlichen Systeme ist, daß sie alle zueinander inkompatibel sind. Dies führte bei potentiellen Anwendern zu der Sorge, bei der Entscheidung, ein Workflowsystem einzusetzen, möglicherweise aufs "falsche Pferd" zu setzen und sich in eine Sackgasse bzw. in eine enge Abhängigkeit von einem Hersteller zu begeben. Insbesondere durch die nicht unerheblichen Investitionen, die mit der Einführung von Workflow-Management-Systemen verbunden sind, führte dies dazu, daß viele Firmen auf den Einsatz von Workflowsystemen verzichteten, auch wenn die Vorteile eines solchen Systems klar auf der Hand lagen. Dies führte 1993 zur Gründung eines Interessenverbands, der sich mit der Standardisierung von Workflowsystemen beschäftigt.

Workflow Management Coalition

Die Workflow Management Coalition (WFMC) wurde im August 1993 als Non–Profit Interessenverband von Workflow-Management-System Anbietern und Anwendern gegründet. Ziel der WFMC ist es, einen Standard im Bereich Workflowsysteme zu etablieren, der die Interoperabilität zwischen Produkten verschiedener Hersteller ermöglicht und die Verbreitung der Workflow Technologie fördern soll [WFM96b].

Dies soll durch die Einführung einer Reihe von Schnittstellen erreicht werden, die es den Anwendern erlauben, bestimmte Module eines Herstellers durch die Module anderer Hersteller zu ersetzen oder zu ergänzen. Abbildung 1.3 zeigt fünf von der WFMC definierte Schnittstellen, die jeweils zwischen der zentralen Komponente eines Workflowsystems, der eigentlichen *Engine*, und den anderen Komponenten definiert sind, und die im folgenden kurz vorgestellt werden sollen:

Interface 1: (Workflow Process Definition Read/Write Interface)

[1] So waren bereits 1994 über 150 verschiedene Anbieter von "Workflow Tools" auf der CeBit vertreten [JBS97].

Schnittstelle zum Austausch von Prozeßdefinitionen. Dazu wird in [WFM98c] eine Grammatik vorgeschlagen, welche die WPDL[2] beschreibt. Die WPDL soll die Formulierung der Arbeitsabläufe erlauben [WFM98c, WFM98f].

Interface 2: (Workflow Client Application Programming Interface)
Hierbei handelt es sich um die Schnittstelle zu Workflow Client Anwendungen, die die Realisierung von speziellen Benutzerfrontends erlauben soll [WFM98e].

Interface 3: (Invoked Applications)
Diese Schnittstelle soll eine Standardisierung beim Applikationsaufruf realisieren. In neueren Dokumenten der WFMC ist das Interface 3 mit in das Interface 2 integriert worden [WFM98e].

Interface 4: (Interoperability)
Interface 4 regelt die Interoperabilität zwischen verschiedenen WFMS. Dies beinhaltet primär, daß konforme Workflowsysteme in der Lage sind, die Prozeßdefinitionen auszutauschen und zu interpretieren [WFM96a, WFM98d]. Im Zusatzdokument [WFM98b] wird ein Mechanismus zum Nachrichtenaustausch auf MIME[3] Basis definiert.

Interface 5: (Draft Audit Specification)
Die Draft Audit Specification standardisiert den Zugriff auf Informationen zur Administration von Workflows, um beliebige Administrations– und Monitoringtools einsetzen zu können.

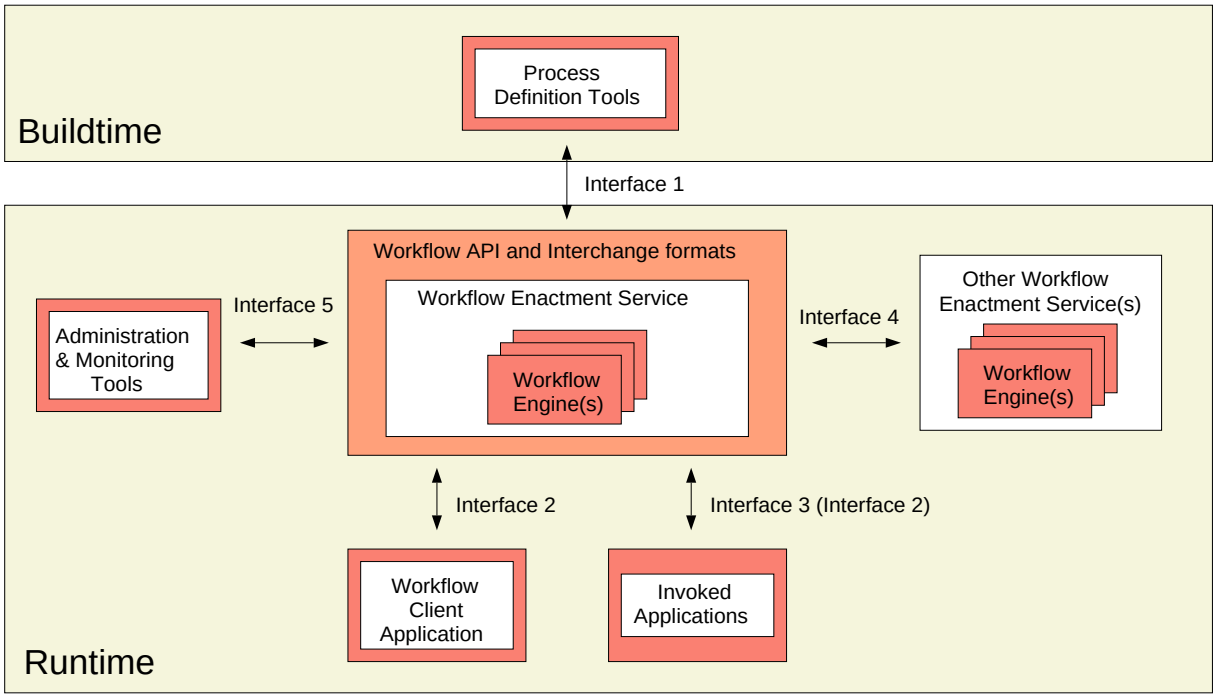


Abbildung 1.3: *WFMC Workflow Referenz Modell [Hol94b]: Komponenten und Schnittstellen*

[2] **W**orkflow **P**rocess **D**efinition **L**anguage
[3] **M**ultipurpose **I**nternet **M**ail **E**xtension (siehe Glossar Seite 182)

Object Management Group

Neben der WFMC gibt es noch ein zweites Konsortium, das sich mit der Standardisierung von Workflows auseinandersetzt. Die Object Management Group (OMG) hat sich zum Ziel gesetzt, die Verbreitung der Objekttechnologie zu fördern und hat mit CORBA[4] eine Referenzarchitektur [OMG95a] vorgestellt. CORBA stellt eine plattform- und sprachunabhängige Infrastruktur für verteilte Objekte bereit. Die OMG hat sich anschließend entschlossen, ihre Standardisierungsbemühungen auf weitere Gebiete auszuweiten und mit den *CORBA Facilities* [OMG95b] Standardisierungsbestrebungen für eine Reihe von weiteren Teilgebieten vorgeschlagen, die als *Dienste*[5] bezeichnet werden. All diesen *Facilities* ist gemeinsam, daß sie auf der von CORBA bereitgestellten Objekttechnologie aufsetzen. Einer der vorgeschlagenen anwendungsabhängigen Dienste ist die *Workflow Facility*[6], für die Anfang 1997 eine Arbeitsgruppe eingerichtet wurde. Die Gruppe veröffentlichte im Mai '97 ein entsprechendes *Request for Proposal* (RFP) [OMG97], als dessen Ziel "die Definition einer Schnittstelle und deren Semantik für die Ausführung und Manipulation von verteilten Workflow Objekten und deren Metadaten" angegeben wird. Die *Workflow Facility* soll hierbei als Plattform zum Aufbau flexibler Management Anwendungen dienen, welche die Verbindung von Objekten und existierenden Applikationen realisiert. Im Juli '98 wurde daraufhin als Reaktion auf das RFP ein überarbeiteter Vorschlag, basierend auf Standards der WFMC, veröffentlicht [OMG98]. Dieser Vorschlag wird von einer Reihe führender Unternehmen getragen und stellt eine Basis für die Integration der Workflow Technologie in das OMG Architekturmodell dar.

1.1.2 Einführung in grundlegende Begriffe und die Technologie eines Workflow-Management-Systems

In [WFM99] wird ein Workflow-Management-System als ein aus Software bestehendes System bezeichnet, das die Erzeugung und Ausführung von Arbeitsabläufen unterstützt, mit den an der Durchführung der Arbeitsabläufe beteiligten Personen (*Workflow-Participant*) interagiert und, falls notwendig, den Aufruf von Applikationen veranlaßt.
Dazu muß das System in der Lage sein, maschinenlesbare Beschreibungen der Arbeitsabläufe (Workflowmodelle) zu speichern, zu interpretieren und daraus einzelne Ablauf-Instanzen (*Workflow-Instanzen*) ableiten zu können. Neben dem Begriff des Workflow-Management-Systems sind auch die Begriffe *Workflow-Automation-System*, *Workflow-Manager* und *Workflow-Computing-System* gebräuchlich.

Ein Workflow-Management-System benötigt zur Ausführung von Workflow-Instanzen *Ressourcen*. Bei *Ressourcen* kann es sich sowohl um Personen, die an der Bearbeitung eines Workflows beteiligt sind, und Maschinen, welche z. B. im Rahmen eines Produktionsworkflows eingesetzt werden, handeln. Daneben stellen auch die benötigten Daten, (z. B. ein CAD-Entwurf oder ein entwickeltes Simulationsmodell) und Programme (z. B. die Verfügbarkeit einer Laufzeitlizenz für ein bestimmtes Programm) *Ressourcen* dar, die zur Ausführung einer Workflow-Instanz notwendig sind.

Ein Workflow besteht in der Regel aus mehreren, z. T. hierarchisch angeordneten Aktivitäten[7]. Zur Ausführung einer konkreten Aktivität werden dieser ein oder mehrere *Workflow-Bearbeitern*

[4] **C**ommon **O**bject **R**equest **B**roker **A**rchitecture

[5] Es werden hierbei horizontale (Anwendungsgebiet unabhängige) und vertikale (auf Anwendungsgebiete spezialisierte) Dienste unterschieden.

[6] hierbei handelt es sich um einen Teil der Task Management Dienste.

[7] andere Bezeichnungen für Aktivitäten sind beispielsweise *Step*, *Node*, *Task*, *Work-Element*, *Process-Element*, *Operation* und *Instruktion*.

zugeordnet. Hierbei handelt es um eine Person oder eventuell auch um eine maschinenbasierte *Ressource*, wie etwa einen *Intelligenten Agenten*. Eine Aktivität wird dem Bearbeiter im allgemeinen in Form von einem oder mehreren Augaben (*Work-Items*) in einer Arbeitsliste (*ToDo-List*) repräsentiert, welche die durchzuführenden Arbeitsschritte festlegen.

Nachdem nun die wichtigsten Begriffe aus dem Bereich Workflow-Management vorgestellt wurden, sollen im folgenden Abschnitt die Kernkomponenten, welche bei den meisten Workflow-Management-Systemen vorhanden sind, vorgestellt werden. Dies soll anhand des Referenzmodells der WFMC [Hol94a] geschehen.

1.1.2.1 Architektur

Prinzipiell läßt sich ein WFMS in zwei große funktionale Einheiten unterteilen. Zum einen ist dies die *Buildtime* Komponente, auch *Prozeßdefinitions-Tool* (Abbildung 1.3 oben) genannt, die zur Erstellung des aus der Realität abzubildenden Prozeßmodells in eine durch einen Computer interpretierbare Beschreibung eingesetzt wird.

Die zweite Einheit wird durch die *Runtime* Komponente gebildet (Abbildung 1.3 unterer Block), welche die Abarbeitung der, mittels des Prozeßdefinitions-Tools definierten, Workflows durchführt und überwacht. Hierzu zählen beispielsweise das Anlegen neuer Workflows, der Aufruf von Applikationen, die Interaktion mit dem Benutzer und das Verwalten der Arbeitslisten.

Die Runtime Komponente läßt sich, wie aus Abbildung 1.3 ersichtlich, noch in Laufzeitkontrolle (mitte) und Laufzeitschnittstellen (unten) unterteilen. Erstere enthält die Workflow-*Engine*, welche für die Ablaufsteuerung sowie die persistente Speicherung der Daten zuständig ist. Die Laufzeitschnittstelle gliedert sich in die Teilbereiche Benutzerschnittstelle (unten, links), Aufrufschnittstelle für Programme (unten, rechts), Administrationsschnittstelle (links) und das Interface zur Kommunikation mit anderen Workflowsystemen (rechts).

Nach der Beschreibung der prinzipiellen Architektur eines Workflowsystems soll im folgenden auf den zentralen Begriff eines Workflowsystems eingegangen werden, den Workflow.

1.1.2.2 Workflow

Die WFMC definiert einen Workflow wie folgt [WFM99]:

> **Definition 1.1 (Workflow)** *[A workflow is] the computerised facilitation or automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.*[8]

Der abzubildende Geschäftsprozeß wird mittels des Prozeßdefinitions-Tools von der realen Welt in eine durch den Computer verarbeitbare Form gebracht. Das Ergebnis dieses Schrittes wird als *Prozeß*– oder *Workflowmodell* bezeichnet.

Ein Modell besteht aus einem Netz von (verteilten) Aktivitäten und den Beziehungen zwischen diesen.

Die WFMC definiert ein Metamodell [Hol94b, WFM99] eines Workflowmodells, das eine Reihe von Komponenten und Konzepten enthält. Hierbei handelt es sich um ein abstraktes Modell,

[8] Übersetzung: *Die computergestütze Unterstützung oder Automatisierung eines Geschäftsprozesses im Ganzen oder in Teilen davon, während deren Dokumente, Informationen oder Aufgaben zur Bearbeitung, entsprechend einer Reihe von prozeduralen Regeln, von einem Beteiligten an den nächsten weitergereicht werden.*

das eine Reihe von alternativen Realisierungen zuläßt[9]. Im folgenden soll eine Realisierung des Modells vorgestellt werden, wie sie im Produkt *FlowMark* von IBM implementiert wurde und in [IBM96] beschrieben wird. Hierbei geht es nicht um eine präzise Definition der verwendeten Konzepte, sondern es soll ein Grundverständnis vermitteln werden, welche Komponenten existieren und wie diese interagieren. Elemente eines Workflowmodells sind:

Aktivität:
Aktivitäten können in Programm- und Prozeßaktivitäten unterteilt werden. Den Programmaktivitäten können Anwendungsprogramme zugeordnet sein, die bei deren Durchführung ausgeführt werden. Prozeßaktivitäten werden als Strukturierungsmittel eingesetzt, um zusammengesetzte Workflows aufzubauen. Eine Prozeßaktivität enthält wiederum eine Reihe von, eventuell hierarchisch geschachtelten, Aktivitäten.

Kontrollfluß:
Der Kontrollfluß legt die Abarbeitungsreihenfolge innerhalb eines Workflows fest und wird durch Pfeile zwischen den Aktivitäten repräsentiert.

Prozeß:
Beschreibung einer Menge von Aktivitäten und ihren Beziehungen untereinander, welche zum Erreichen eines bestimmten Arbeitsziels durchgeführt werden müssen. Bei den Aktivitäten kann es sich sowohl um manuelle– (d. h. nicht durch einen Computer ausführbare), als auch automatische (durch einen Computer ausführbare oder unterstützbare) Aktivitäten handeln.

Eingabecontainer:
Eingabecontainer speichern Daten, die als Eingabedaten für die auszuführenden Applikationen dienen.

Ausgabecontainer:
Ausgabecontainer speichern Daten, die die Ausgabedaten der auszuführenden Applikationen aufnehmen.

Datenfluß:
Der Datenfluß kann als Abbildung zwischen Ausgabe– und Eingabecontainern angesehen werden, und analog zum Kontrollfluß, durch Pfeile zwischen Aktivitäten symbolisiert werden.

Bedingung:
Ein logischer Ausdruck, der von der Workflow-Engine zur Laufzeit ausgewertet wird um zu entscheiden, ob beispielsweise eine bestimmte Aktivität gestartet werden kann (Startbedingung) oder ob die Kontrolle von einer Aktivität an eine weitere Aktivität weitergereicht werden kann (Endbedingung, Transitionsbedingung).

Ein wichtiger Punkt ist hierbei die mögliche Verteilung der einzelnen Aktivitäten und Prozesse auf verschiedene Rechner(systeme). Daher spielt die Kommunikation zwischen den einzelnen Komponenten eines solchen Systems eine zentrale Rolle. In den Abschnitten 2.2.1.2 und 2.2.2.3 wird genauer auf die Anforderungen an ein derart verteiltes System eingegangen.

[9] Dies rührt daher, daß in der WFMC eine Reihe von Workflowherstellern vertreten sind, die alle darauf Wert gelegt haben, daß ihr konkretes Workflowsystem den Standardisierungsbemühungen weitgehend entspricht.

1.1.2.3 Modellierungsmöglichkeiten von Workflows

Nachfolgend sollen, um das breite Spektrum möglicher Realisierungen zu verdeutlichen, eine Reihe von Überlegungen angestellt werden, die bei der Entwicklung eines Workflowsystems eine Rolle spielen. Die Punkte sind aus [Bö97] entnommen und verdeutlichen die Vielfalt der unterschiedlichen möglichen Realisierungsansätze.

Als erstes sollen hier die Möglichkeiten bei der Modellierung der Arbeitsvorgänge betrachtet werden. In [Bö97] werden drei Erscheinungsformen unterschieden. Als einfachste Variante sind Arbeitsvorgänge mit fester Ausführungsreihenfolge genannt, die nur einfache Kontrollflußkonstrukte wie Sequenz, Parallelität und Alternative erfordern. Daneben gibt es Arbeitsvorgänge, die aus bekannten Teilaufgaben bestehen, aber Freiheiten in ihrer Abarbeitung zulassen. Das Workflowsystem muß diesem Punkt Sorge tragen, indem es erlaubt, solche Freiheiten zu modellieren. Die dritte Variante von Arbeitsvorgängen enthält Teilaufgaben, deren Abarbeitung die jeweilige Situation bestimmt, d. h. der Kontrollfluß wird in Abhängigkeit der Laufzeitinformationen bestimmt und nicht wie in den beiden vorherigen Fällen durch einen zuvor explizit definierten Fluß.

Sichtweise von Arbeitsvorgängen

Die Arbeitsvorgänge können aus unterschiedlichen Blickwinkeln betrachtet werden. In der Literatur werden folgende Sichtweisen unterschieden [Bö97, Rei93]:

Strukturierte Abfolge von Teilschritten: Die Koordination der zu tätigenden Arbeitsschritte steht im Mittelpunkt.

Migrierende Objekte: Als Beispiel hierfür kann die Modellierung der Arbeitsvorgänge in Form von Umlaufmappen angesehen werden. Hierbei werden die zu bearbeitenden Objekte, d. h. die Daten, in den Mittelpunkt gestellt und die Aufgaben den Datenobjekten zugeordnet. In den meisten Fällen sind die jeweiligen Daten eng mit einer Anwendung verknüpft, so beispielsweise ein *Winword* Dokument mit der zugehörigen Textverarbeitung.

Aktualisierte Konversationstypen: Der Ansatz geht auf die Sprechakttheorie von Searle [Sea69] zurück. In diesem Modell wird ein geregelter und strukturierter Nachrichtenaustausch zwischen den beteiligten Personen in den Vordergrund gestellt.

Zustandsbehaftete Objekte: Analog zum Objektmigrationsmodell stehen hier die Daten im Vordergrund. Im Unterschied zum Objektmigrationsmodell äußert sich der Fortschritt der Abarbeitung im Wechsel von Zuständen (z. B. in Arbeit, erledigt), welche den Objekten explizit vom Bearbeiter zugewiesen werden.

Unterstützung durch das System

Hierbei werden die verschiedenen Möglichkeiten betrachtet, wie der Anwender vom System unterstützt werden kann. Das Spektrum reicht von einfacher textueller Unterstützung bis hin zu Systemen mit Methoden zur Entscheidungsfindung.

Unterstützung durch Information: In diesem einfachsten Fall erfolgt die Unterstützung in Form von Informationen, welche dem Anwender zur Verfügung gestellt werden. Eine weitere Unterstützung seitens des Workflowsystems muß in diesem Fall nicht gegeben sein.

Unterstützung durch Überwachung: Hierbei handelt es sich um ein "passives System", das die Abarbeitung überwacht und sich nur dann einschaltet, wenn eine explizit formulierte Regel verletzt wird. Damit kann eine Qualitätssicherung der Arbeit erreicht werden.

Unterstützung durch Steuerung und Kontrolle: In diesem Fall ist das Workflowsystem die aktive Komponente, die das Geschehen steuert. Es handelt sich hierbei um die verbreiteste Form der Unterstützung, wie sie in heutigen Systemen realisiert ist. Sie ermöglicht, durch die Festlegung der Reihenfolge der einzelnen Arbeitsschritte, u. a. eine genaue Zeitüberwachung und Terminplanung.

Unterstützung durch Assistenz und Planung: Bei der Unterstützung durch Assistenz und Planung spielt nicht nur der aktuelle Zeitpunkt der Durchführung eine Rolle, sondern es werden auch Informationen aus vergangenen Abläufen zu Rate gezogen, etwa bei Vorschlägen für das weitere Vorgehen im Rahmen einer Workflowinstanz. Im Unterschied zu den beiden zuvor besprochenen Unterstützungsarten erhält der Anwender Assistenz bei seiner kreativen Arbeit und nicht nur bei Routineaufgaben bzw. arbeitsorganisatorischen Vorgängen.

Im Fall der Unterstützung durch das System wachsen die Anforderungen an das Workflowsystem von Punkt zu Punkt, wie kurz am Beispiel der Festlegung der Ausführungsreihenfolge verdeutlicht werden soll. Je nach Art der Unterstützung muß diese anders realisiert werden. Reicht im ersten Fall die textuelle Beschreibung der Ablauffolge, so müssen im letzten Fall Ziele definiert werden können und Bewertungskriterien formulierbar sein, welche zur Bestimmung der weiteren Ablauffolge herangezogen werden können.

Konkrete Ausprägungen der genannten Punkte definieren ganz entscheidend das Aussehen und die Funktionalität des zu realisierenden Systems. Im Vorfeld ist aber keine generelle Aussage über die am besten geeignete Realisierung möglich, da diese speziell vom Aufgabengebiet und den damit verknüpften Randbedingungen abhängt. Auch spielt der Benutzer eine entscheidende Rolle, da seine Kenntnis über das zugrundeliegende System, den Arbeitsprozeß als solchen und der daraus abgeleitete Workflow bei der Realisierung des Systems wichtig sind. Bei bestimmten Punkten, wie etwa dem Grad der Unterstützung durch das System, ist es möglich, eine Realisierung zu wählen, welche die einzelnen Varianten umfaßt. Bei der Modellierung der Arbeitsvorgänge ist diese Vorgehensweise nicht mehr möglich, da die verschiedenen Ansätze zu unterschiedlich sind.

Zusammengefaßt kann gesagt werden, daß eine zufriedenstellende Lösung nur in Abhängigkeit vom jeweiligen Einsatzgebiet und Anwender gefunden werden kann.

1.2 Stand der Technik und Bewertung

Bei der Untersuchung von auf dem Markt verfügbaren kommerziellen Workflowprodukten kann beobachtet werden, daß der Schwerpunkt bei Systemen liegt, deren Einsatzgebiete im Geschäfts– und organisatorischen Bereich liegen. So sind Banken, Versicherungen, Ministerien, universitäre Verwaltungseinrichtungen und das Gesundheitswesen klassische Einsatzgebiete von Workflow-Anwendungen. Ohne im folgenden konkret auf einzelne Systeme eingehen zu wollen, sollen hier eine Reihe von Eigenschaften heute erhältlicher kommerzieller Systeme vorgestellt werden.

1.2.1 Kommerzielle Systeme

An dieser Stelle soll auf eine detaillierte Vorstellung oder einen Vergleich zwischen den verschiedenen Systeme verzichtet werden. Umfassende Informationen hierzu finden sich in [JB96, JBS97, Moh97, AAAM97, TG97].

Zu den positiven Eigenschaften heutiger Systeme zählen das Vorhandensein von graphischen Werkzeugen und Editoren zur Spezifikation der Arbeitsabläufe. Allerdings benutzt aber fast jeder Hersteller eine eigene Spezifikationsmethode [Alo96], so daß eine Interoperabilität zwischen verschiedenen Herstellern nicht gegeben ist. Das Spektrum reicht hier von der strukturierten Abfolge von Teilschritten, bei der die zu tätigenden Arbeitsschritte im Mittelpunkt stehen (*FlowMark* [IBM96], *WorkParty* [SIM95]) über migrierende Objekte und zustandsbehafteten Objekte (*INCAS* [BMR94], *ProMInanD* [IAB96]) bis hin zu aktualisierten Konversationstypen beim System *ActionWorkflow* [MMWFF92]. Abhilfe sollen hier Standards, wie sie etwa von der WFMC im Interface 1 definiert wurden, schaffen. So sind momentan[10] die Produkte *KI Shell* von Concentus, *COSA Workflow* von COSA Solutions, *Visual WorkFlo 3.0* und *Integrated WorkFlo* von FileNET, *Handy*Solution* von HandySoft Corp., *ARIS Toolset 3.2* und *Modelling and BPR Tool* von der IDS Prof. Scheer GmbH, *INCOME Workflow V.1.0* der Firma PROMATIS sowie *Staffware97* von Staffware von der WFMC als konform zu den einzelnen auf Seite 3 vorgestellten Schnittstellen ausgewiesen [WFM98a].
Allgemein kann gesagt werden, daß fast alle Systeme über ausgefeilte graphische Benutzeroberflächen verfügen. Viele der angebotenen Produkte weisen programmatische Schnittstellen (API[11] (Glossar Seite 179) zu Standard Anwendungen auf, die etwa die Anbindung an Produkte wie das *SAP/R3*–System erlauben. Zusatzanwendungen, wie Report– und Planungstools, sind ebenfalls bei den meisten Systemen vorhanden.

Fast allen Produkten ist gemeinsam, daß sie eine zentrale Datenbank zur Ablage der Prozeßinformationen und –zustände einsetzen [Moh97, AAAM97, GHS95]. Das Spektrum reicht hier von dateibasierten Ansätzen (z. B. *Staffware*) und eigenentwickelten Datenbanken (z. B. *OPEN/workflow* von WANG) bis zu kommerziellen relationalen Datenbanksystemen etwa von ORACLE oder Microsoft, bzw. ODBC[12]–Schnittstellen zu relationalen Datenbanken wie bei *ProcessIT* von AT&T oder auch objektorientierten Lösungen wie z. B. bei IBM's Produkt *FlowMark*. Diese Architektur ist aber Anlaß vielfältiger Kritik. So stellt die zentrale Datenbank einen *"Single Point of Failure"* dar, der die Verfügbarkeit des Workflowsystems bestimmt. Je nach zugrundeliegender Datenbank kann es auch zu Skalierungsproblemen kommen, d. h. der Einsatz ist beschränkt auf kleinere Gruppen.

Weitere Schwächen der heutigen kommerziellen Systeme liegen in den Bereichen Korrektheit im Fehlerfall, Architektur, Einbindung von externen Anwendungen, heterogene Umgebungen, Flexibilität, Behandlung von Ausnahmesituationen, Erweiterbarkeit [JET97, AAAM97, Alo96, AS96, AHST97b, HA98b, AAEA+96, GHS95, BW96, TG97, Moh97], um die wichtigsten Punkte zu nennen.

Die an kommerziellen Systemen geäußerte Kritik stammt zu einem Großteil aus Forschungslabors und dem universitären Bereich und dort sind auch eine Reihe von Systemen entstanden, die sich mit einzelnen der oben beschriebenen Kritikpunkte befassen.

[10] Stand Oktober '98
[11] **A**pplication **P**rogramming **I**nterface
[12] **O**pen **D**ata**B**ase **C**onnectivity (siehe Glossar Seite 182)

1.2.2 Forschungsprojekte

Inzwischen beschäftigen sich eine Reihe von universitären Einrichtungen und Forschungslaboren mit der Workflowtechnologie. Schwerpunkte im Bereich Workflow Forschung und Enwicklung, die auch speziell im Rahmen dieser Arbeit intensiv verfolgt wurden, können an folgenden Forschungseinrichtungen beobachtet werden.

Almaden Research Center, San Jose

An diesem Forschungslabor von IBM wurde das Project *Exotica* durchgeführt. Forschungsschwerpunkte von *Exotica* lagen in den Bereichen Skalierbarkeit, Verfügbarkeit und die räumliche Verteilung von Workflow Prozessen. Ausgangspunkt für die Entwicklung eines Prototyps waren die IBM Produkte *FlowMark*, das Groupeware-System *Lotus Notes/Domino* und *MQSeries* zum Austausch persistenter Nachrichten zwischen verteilten Workflowknoten. Basierend auf diesen Produkten wurden unterschiedliche Architekturen realisiert. Eine der Architekturen war eine "vollständig verteilte", die aus einer Reihe unabhängiger Rechner (Knoten) bestand. Durch persistente Nachrichten zwischen den einzelnen Knoten konnte eine vollständige Verteilung, ohne zentrale Datenbank, des Workflowsystems erreicht werden. Weitere Forschungsschwerpunkte waren erweiterte Transaktionsmodelle und Backup–Strategien [HA98b, AMG+95, MAA+95, MAGK95].

Database Research Group – ETH Zürich

Schwerpunkte der Workflowforschung sind hier die Punkte transaktionale Unterstützung, Ausnahmebehandlung und Skalierung. Das System *OPERA* stellt eine generische Plattform zum Aufbau von verteilten Anwendungen dar. Ideen, die in *OPERA* verarbeitet wurden, stammen aus den Bereichen verteilter und paralleler Datenbanken, transaktionaler Systeme und Workflow. Das Hauptaugenmerk in diesem Projekt liegt auf der Bereitstellung von Mechanismen für verteilte Ausführung von Prozessen [AHST97b, AHST97a, HA98a, Alo97]. Einige der Ideen aus dem *Exotica*–Projekt (s. o.) werden hier in Zürch weiterverfolgt.[13]

Universität Erlangen

Das System $\mathcal{MOBILE}$ wurde am Institut für Datenbanksysteme der Universität Erlangen entwickelt. Hauptziele hierbei waren Erweiterbarkeit und Offenheit [Bußß97]. Dies führte zu einem modularen Aufbau. Die Modularisierung erfolgte anhand sogenannter *Aspekte*. Jede Komponente ist für die Realisierung eines Aspektes verantwortlich. Weiterhin gibt es eine ausgezeichnete Komponente, über die die Kommunikation zwischen den Aspekt–Modulen zu erfolgen hat, so daß einzelne Module leicht ausgetauscht werden können. Weitere interessante Punkte sind die Integration von Middlewarediensten[14], eine flexible Organisationsverwaltung und der Einsatz eines erweiterbaren Workflowmetaschemas [Sch96b, Sch97, Bußß97, SJKB94, JB96, JBS97, SJHB96].

Weitere experimentelle Workflowsysteme aus Forschungseinrichtungen werden in Abschnitt 1.2.3.3 vorgestellt. Im Unterschied zu den hier besprochenen Projekten handelt es sich

[13] Dies ist durch den Wechsel von Gustavo Alonso vom Almaden Research Center an die Datenbankgruppe der ETH Zürch zu erklären.

[14] Middlewaredienste abstrahieren von plattform- und/oder sprachabhängigen Eigenheiten und stellen darüberliegenden Anwendungen eine einheitliche Schnittstelle zur Verfügung, die von den verschiedenen darunterliegenden Protokollen abstrahiert. Verbreitete Middlewaredienste sind CORBA von der OMG, *MQSeries* von IBM und *COM/DCOM* von Microsoft.

dabei um Systeme, die versuchen, den Anforderungen von Arbeitsabläufen im wissenschaftlich–technischen Bereich gerecht zu werden.

1.2.3 Workflowsysteme im wissenschaftlich–technischen Umfeld

Im folgenden soll der Einsatz von Workflowsystemen im wissenschaftlich–technischen Umfeld betrachtet werden. Ausgehend von einem Beispielszenario, das eine Reihe von typischen Merkmalen wissenschaftlich–technischer Abläufe aufweist und somit speziell einen mit diesem Gebiet nicht vertrauten Leser als Illustration dient, wird anschließend die Notwendigkeit bzw. die Vorteile eines Einsatz von Workflowsystemen in diesem Umfeld aufgezeigt. Danach werden die wichtigsten Charakteristiken von Arbeitsabläufen im wissenschaftlich–technischen Umfeld herausgearbeitet, die dann zur Formulierung der Anforderungen dienen.

Beispielhaft sollen an dieser Stelle Arbeiten der Abteilung Mikrosystem–Informatik des Instituts für Angewandte Informatik (IAI) vorgestellt werden. Die Abteilung beschäftigt sich unter anderem mit der Simulation von Bauteilen, die am Institut für Mikrostrukturtechnik gefertigt werden. Durch die Simulation kann die kostenintensive Herstellung von Variationen der Bauteile stark reduziert werden. Eng damit verbunden ist die anschließende Optimierung der Bauteilkonstruktion nach unterschiedlichen Kriterien (etwa Durchfluß bei einer Mikropumpe [Qui98, Mei98] oder der Optimierung des Designs bezüglich der Fertigungstoleranzen bei einem Heterodynempfänger [SEGJ98]).

In den Bereichen Modellierung, Simulation und Optimierung kommen kommerzielle[15] und eigenentwickelte[16] Werkzeuge zum Einsatz. Oft besteht die Notwendigkeit die Werkzeuge untereinander zu koppeln. Dies kann auf die verschiedensten Arten, teilweise auch über Rechnergrenzen hinweg, notwendig sein. Aufgrund der erforderlichen heterogenen Rechnerlandschaft kann es auch vorkommen, daß die Kopplung über Betriebssystemgrenzen und Hardwarearchitekturen hinweg erfolgen muß.

Weiterhin fallen, speziell im Bereich Simulation, eine große Menge von wissenschaftlich–technischen Daten, in zumeist unstrukturierter Form, an, die es zu verwalten gilt.

Durch die verteilte, heterogene Rechnerlandschaft stellt sich ein weiteres Problem, das bisher von den meisten Workflowsystemen nicht oder nur unzureichend gelöst wird. Es handelt sich hier um den Aufruf beliebiger Applikationen, die nicht für den Einsatz im Rahmen eines Workflowsystems konzipiert wurden [SJKB94, SJHB96, HS97, HA98b].

Nach dieser kurzen Situationsbeschreibung sollen nun anhand eines konkreten Beispiels aus dem Bereich Modellierung und Simulation die Anforderungen an Workflowsysteme im wissenschaftlich–technischen Bereich dargestellt werden.

1.2.3.1 Beispiel

Das Programmpaket *CFX* [CFX97] der Firma AEA Technology besteht aus mehreren eigenständigen Programmen zur Modellierung, Simulation und Visualisierung von Flüssen und Wärmeübertragungen.

Das Prinzip basiert darauf, daß die Geometrie eines Körpers in einzelne Zellen zerlegt wird. Anschließend werden die partiellen Differentialgleichungen (Navier-Stokes Gleichungen) jeder Zelle, die den Fluß beschreiben, in algebraische Gleichungen transformiert, welche die Werte für Druck, Geschwindigkeit, Temperatur, etc. mit den Werten der Nachbarzellen in Relation setzen.

[15] z. B.: *ANSYS*, *CFX*, *Solstis*, *Mathematica*

[16] *SIMOT* [JMQC96], *GADO* [JMQ+96], *COSMOS* [BGH91], *LIDES* [SBES96]

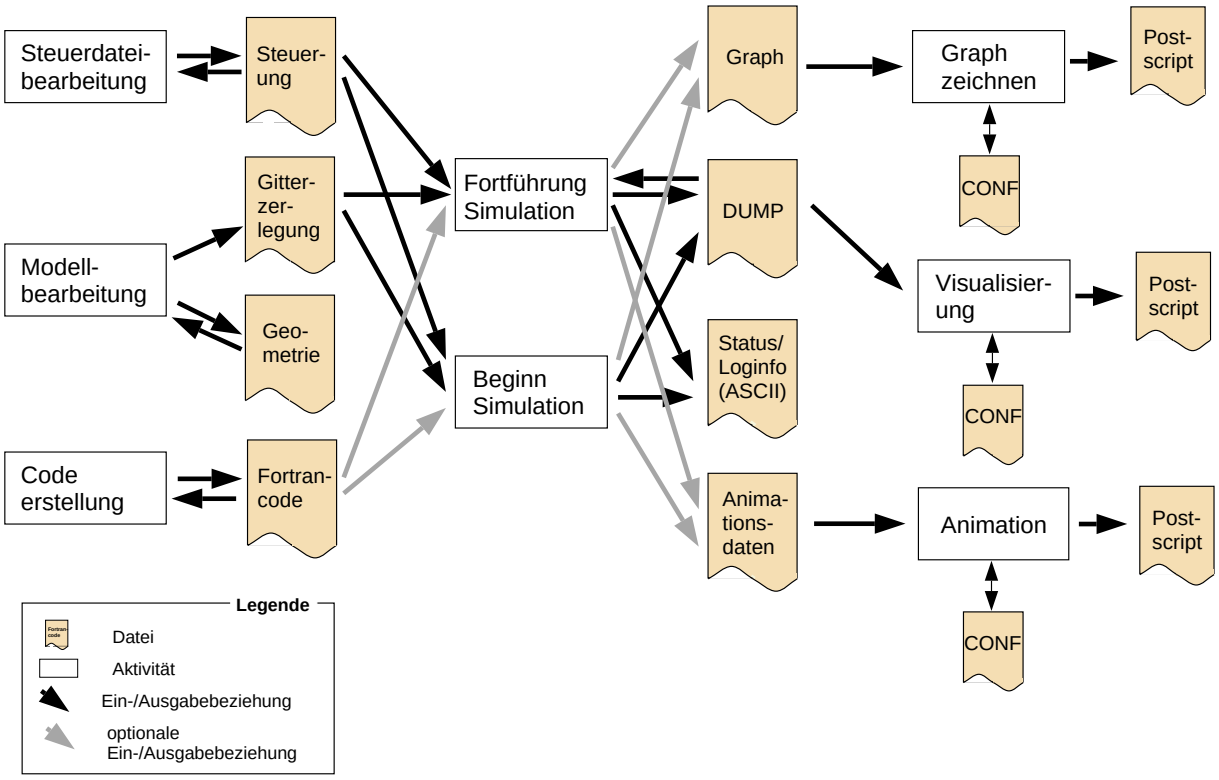


Abbildung 1.4: *Programmpaket CFX: Datenfluß*

Abbildung 1.4 zeigt in schematischer Form die wichtigsten Arbeitsschritte, sowie die Daten, die zum Ablauf einer Fluidik–Simulation notwendig sind.

Im ersten Schritt muß das zu simulierende Modell erstellt und anschließend eine Gitterzerlegung dieses Modells für das Simulationsmodul generiert werden. Bei der Gitterzerlegung wird das Modell in eine Anzahl von möglicherweise unterschiedlich großen Einzelzellen zerlegt.

Das Modell kann auch vom Entwickler erstellte Modulerweiterungen beinhalten, welche als Fortran Quellcode eingebunden werden. Weiterhin ist eine Steuerungsdatei zu erstellen, welche den Ablauf der Simulation festlegt. Der Ablauf einer Simulation, d. h. die Einträge in der Steuerungsdatei, sind vom erzeugten Modell abhängig, so daß hier von einer $1 : n$ Beziehung zwischen Modell und Steuerungsdatei gesprochen werden kann. Ob das Modell Fortran-Module enthält, muß ebenfalls in der Steuerungsdatei angegeben werden, so daß der Simulator in einem ersten Schritt die als Quellcode vorliegenden Module übersetzen und anschließend in die Simulation einbinden kann.

Im zweiten Schritt wird der Simulationslauf gestartet. Hierzu benötigt der Simulator als Eingabedaten die Steuerungsdatei, die Gitterzerlegung des Modells und gegebenenfalls die erstellten Fortran-Dateien.

Da eine Simulation einen großen Zeitraum in Anspruch nehmen kann, ist es möglich, sie zu jedem Zeitpunkt zu unterbrechen und anschließend wieder fortzusetzen. In Abhängigkeit von den Angaben in der Steuerungsdatei werden zwischen zwei und vier Ausgabedateien erstellt. Die wichtigste dieser Ausgabedateien ist die *DUMP*–Datei, welche den kompletten Simulationslauf mit den erzielten Ergebnissen enthält. Außerdem wird sie bei Wiederaufnahme der Simulationsrechnungen nach einer Unterbrechung als Eingabedatei benutzt, damit der Simulator den bishe-

rigen Stand der Simulation ermitteln und fortfahren kann. Weiterhin wird eine Datei mit Status– und Log-Meldungen erstellt, die im Falle eines Fehlers eine genaue Analyse ermöglicht. Optional werden noch eine Reihe von Ausgabedateien geschrieben, welche bei der visuellen Auswertung herangezogen werden können. Im einzelnen handelt es sich um eine Datei zum Erstellen von Graphen sowie um eine Datei mittels der Animationen erstellt werden können. Hierfür stehen jeweils eigenständige Programme zur Verfügung. Die Visualisierungswerkzeuge können mittels zusätzlicher Konfigurationsdateien (*CONF*) auf bestimmte Bereiche fokussiert werden. Dies ermöglicht bei der Visualisierung mit den unterschiedlichen Werkzeugen die Berücksichtigung bestimmter relevanter Aspekte.

Nachdem im vorherigen Abschnitt der Ablauf einer Simulation aufgezeigt wurde, soll nun kurz die Vorgehensweise bei der Modellerstellung beschrieben werden. Hierbei handelt es sich um einen iterativen Prozeß, bei dem versucht wird, ein Modell zu konstruieren, das dem Verhalten in der Realität entspricht und bestimmte Randbedingungen, wie etwa Simulationsdauer, erfüllt.

Fragen, die bei der Modellerstellung geklärt werden müssen, sind folgende:

Modell: Ist das zu simulierende Objekt adäquat modelliert worden?

Gitterzerlegung: Ist die gewählte Gitterzerlegung fein genug, um präzise Ergebnisse zu erhalten, bzw. erhält man auch mit einer gröberen Gitterzerlegung ausreichende Präzision?[17]

Steuerungsdatei: Kann der Ablauf einer Simulation optimiert werden bzw. sind zusätzliche Ausgabedaten zur Verifizierung der Ergebnisse notwendig?

Die im Rahmen einer Modellerstellung anfallenden Iterationsschritte erzeugen eine große Menge von in Betriebssystemdateien vorliegenden Ergebnisdaten. Weiterhin entstehen bei den einzelnen Simulationsläufen verschiedene Versionen der Eingabedaten, die verwaltet werden müssen.

Das Programmsystem *CFX* stellt zur Verwaltung der anfallenden Daten keinerlei Mechanismen zur Verfügung, so daß für den Anwender lediglich eine unübersichtliche Menge von Daten ohne erkennbaren Zusammenhang zurückbleibt. Der Anwender wird zwar automatisch versuchen, den Dateien sinnvolle Namen zu geben, bzw. verschiedene auseinander hervorgehende Versionen einer Modell- oder Steuerdatei mit aufsteigenden Nummern zu versehen, der Zusammenhang zwischen den verschiedenen Ein- und Ausgabedateien ist so aber meist nicht aufrecht zu erhalten.

Typische, im Verlauf einer Modellentwicklung auftretende Fragen sind etwa:

- Mit welcher Gitterzerlegung/Steuerungsdatei wurde die Ergebnisdatei (DUMP) *run_18a.dump* erzeugt?
- Aus welchem Modell (d. h. welche Modelldatei) wurde die Gitterzerlegungsdatei *ventil_12.geo* generiert?
- Mit welcher Steuerungsdatei und welchem Modell wurde die Visualisierungsdatei *ventilhals_2.ps* erzeugt?
- Seit welchem Modell wurde der modifizierte Fortran-Code eingesetzt?
- Wie muß die Animationskomponente konfiguriert werden, um für Modell *ventil_2v2.log* die interessanten Aspekte zu beleuchten?

[17] Die Gitterzerlegung bestimmt maßgeblich die Rechenzeit einer Simulation, so daß stets ein tragbarer Kompromiß aus Rechengenauigkeit und Simulationszeit gefunden werden muß.

Um obige Fragen auch nach längerer Zeit noch beantworten zu können, ist eine Datenhaltungskomponente notwendig, mit der es möglich ist, solche Zusammenhänge zwischen Daten modellieren zu können.

Obwohl solche Probleme im Datenbankbereich längst gelöst sind, sind heutige Workflowsysteme noch nicht in der Lage solche Beziehungen zwischen Daten auszudrücken [Alo96]. Dies betrifft nicht nur den Bereich Datenmanagement, sondern auch eine Reihe anderer Bereiche wie Verteilung [Moh97], Transaktionsmechanismen [AHST97b] und Ausnahmebehandlung [HA98b].

In Alonso et. al. [AHST97a] wird diese Problematik durch folgendes Zitat formuliert:

> *"the problem is not the lack of solutions but the lack of integrated solutions"*[18]

1.2.3.2 Nutzen und Anforderungen von Workflowsystemen im wissenschaftlich–technischen Umfeld

Nach diesem motivierenden Beispiel soll im folgenden der Bedarf für Workflowsysteme im wissenschaftlich–technischen Umfeld untersucht werden.

Im Umfeld von Wissenschaft und Technik ist der Computer nicht mehr wegzudenken. Der Computer gehört heute, nicht nur in den Naturwissenschaften, zum Handwerkszeug eines jeden Wissenschaftlers. Weiterhin läßt sich beobachten, daß die Rechnerumgebungen, welche den Forscher bei seiner Arbeit unterstützen, in bezug auf Verteilung, Heterogenität und unterstützender Software immer komplexer und leistungsfähiger werden. Dies erlaubt einerseits immer aufwendigere Experimente mit Unterstützung der Computer durchzuführen, andererseits erfordern die Umgebungen ein hohes Maß an Aufwand bei der Realisierung und Überwachung solcher Experimente.

Beobachtung 1:

Das wissenschaftlich–technische Arbeitsumfeld stellt aufgrund seiner Heterogenität und Komplexität sowohl im Hardware, als auch im Softwarebereich, hohe Anforderungen an ein Workflowsystem.

Ein zu realisierendes Workflowsystem, mit Schwerpunkt des Einsatzes im wissenschaftlich–technischen Umfeld, wird sinnvollerweise über die selbe Grundfunktionalität verfügen, wie ein System im administrativen Umfeld. Dies ist insofern plausibel, da die Funktionalität von administrativen Workflowsystemen, wie Definition und Ausführung von Workflows, Benutzerkontrolle und Bereitstellung der Daten, auch im wissenschaftlich–technischen Bereich vorhanden sein muß. Weiterhin kann man gerade in letzter Zeit beobachten, daß betriebswirtschaftliches Denken sich auch im Forschungsbereich durchzusetzen beginnt und ein Workflowsystem hierbei ein wichtiges Werkzeug zur Arbeitsoptimierung und Planung darstellen kann. Die Beschäftigung mit den zusätzlichen Anforderungen im wissenschaftlich–technischen Bereich kann wiederum das Verständnis von Arbeitsabläufen im allgemeinen voranbringen, so daß eine tiefergehende Beschäftigung mit wissenschaftlichen Workflows auch positive Auswirkungen für Workflowsysteme im betriebswirtschaftlichen Bereich haben wird.

Beobachtung 2:

Die Anforderungen an Workflowsysteme für das wissenschaftlich–technischen Umfeld umfassen die Anforderungen an Workflowsysteme im administrativen Bereich.

[18] Übersetzung: *"Das Problem ist nicht das Fehlen von Lösungen sondern das Fehlen von integrierten Lösungen"*.

Indem dem Wissenschaftler oder Entwickler ein System, mittels dem er seine Experimente automatisch steuern, überwachen und dokumentieren kann, zur Verfügung gestellt wird, ist es ihm möglich, sich auf seine eigentlichen Aufgaben zu konzentrieren, ohne daß er seine Zeit mit lästigen, ständig wiederkehrenden verwaltungstechnischen Routinearbeiten verbringen muß.

Da den Forschern momentan noch keine ausgereiften Werkzeuge zur Definition ihrer Arbeitsvorgänge und der anschließenden Analyse zur Verfügung stehen, bedeutet dies, daß die Experimente von Hand prozedural modelliert werden, was aber sehr aufwendig und fehleranfällig ist [SV96].
Eine Unterstützung von Seiten eines geeigneten Workflowsystems kann die Arbeit der Wissenschaftler somit entscheidend erleichtern, da die Modellierung der Arbeitsvorgänge mittels einem extra für diese Aufgabe entwickeltem Werkzeug erfolgen kann, das schon bei der Definition der Arbeitsvorgänge Inkonsistenzen erkennen bzw. eventuell die Möglichkeit der Simulation eines Ablaufs anbietet.

Die prinzipielle Struktur eines wissenschaftlichen Experiments weist starken Prozeßcharakter auf [BW96, ILGP96]. Ein Workflow, der ein wissenschaftliches Experiment beschreibt, läßt sich grob in die folgenden drei Schritte unterteilen:

Entwurf des Experiments: In einem ersten Schritt wird das durchzuführende Experiment skizziert, d. h. der Ablauf wird festgelegt und seine Ein– und Ausgabeparameter bestimmt [Zei76].

Datensammlung und Integration: Im nächsten Schritt werden die interessierenden Daten gesammelt. Dies kann manuell oder auch vollautomatisch erfolgen. Als Datenquellen kommt von Software wie etwa Simulationswerkzeuge bis hin zu externer Hardware alles in Frage. Die gesammelten Daten werden anschließend eventuell noch einer Reihe von Kalibrierungen unterworfen.

Untersuchung: In diesem Schritt werden die gesammelten Daten untersucht, d. h. es werden verschiedene Analysen durchgeführt und Schlüsse daraus gezogen.

Die Zunahme der Komplexität der Versuche führt zu einer extremen Zunahme der dabei anfallenden Datenmengen [MWBM96]. Die Probleme, die sich mit der Handhabung dieser Daten stellen, wurden in [FJP90] formuliert und später auch Lösungen für bestimmte Einsatzgebiete, wie z. B. Molekularbiologie [KSK93, SU94] vorgestellt. Neben der Handhabung und Verwaltung der Daten spielt auch noch die Semantik eine große Rolle, da die Daten oft nur noch mit Computerunterstützung analysiert werden können. Neben dem Forschungsgebiet *Scientific Databases*, das sich mit der Speicherung und Verwaltung beschäftigt, versucht der Forschungsbereich *Data–Mining*[19] (auch *Knowledge Discovery* [PS91] genannt), Informationen aus Datenmengen abzuleiten und daraus Schlüsse zu ziehen.

Die zentrale Rolle von Daten im Rahmen von Experimenten wird allgemein anerkannt [ILGP96, BW96, AIL98] und dies ist auch ein entscheidender Unterschied etwa zu Workflows im Geschäftsbereich (Zitat: *"Wheras office work is about goals, scientific work is about data"*[20] [WWVBM96]). Das zeigt sich auch in dem Beispiel aus Abschnitt 1.2.3.1.

[19] Data–Mining bedeutet übersetzt soviel wie das Schürfen in Datenbeständen. Hierbei wird versucht, mit Methoden der künstlichen Intelligenz und verwandter Gebiete interessante Informationen aus den Datenbeständen zu extrahieren.

[20] Übersetzung: *"Während Büro-Arbeit durch Ziele bestimmt wird, wird wissenschaftliche Arbeit durch Daten bestimmt."*

Beobachtung 3:

Durch die zentrale Rolle der Daten bei wissenschaftlichen Experimenten werden die Anforderungen an eine Datenhaltungskomponente, in bezug auf Modellierungsmöglichkeiten und Leistungsfähigkeit, bestimmt.

Eine leistungsfähige Datenhaltungskomponente erlaubt es somit dem Bearbeiter, Beziehungen zwischen bestimmten Datensätzen innerhalb seines Systems zu verwalten und konsistent zu halten, ohne daß er diese extern dokumentieren und ständig aktualisieren muß.

Eine weitere Anforderung an Workflows im wissenschaftlich–technischen Umfeld liegt in der großen Flexibilität, die bei der Durchführung der Arbeitsschritte nötig ist, da oft erst während der Ausführung eines Experiments die nachfolgenden Schritte festgelegt werden können [WWVBM96]. Dadurch wird der Wissenschaftler stärker in den Workflowablauf eingebunden, da er im Gegensatz zu Workflowanwendungen im administrativen Office–Bereich, entscheidend den weiteren Ablauf des Workflows bestimmen muß.

Diese Charakteristik wissenschaftlich/technischer Workflows ist am ehsten mit Situationen wie sie innerhalb der Abarbeitung von Produktionsworkflows auftreten vergleichbar. So treten im Rahmen der Abarbeitung von Produktionsworkflows aufgrund geänderter Prioritäten oder der Nichtverfügbarkeit bestimmter Ressourcen (Programme, Personen, Geräte, etc.) oft Änderungen im Herstellungsprozess auf. Als Folge muß ein laufender Workflow unterbrochen und zu einem späteren Zeitpunkt fortgesetzt, oder es müssen Modifikationen am laufenden Prozeß vorgenommen werden, was den Eingriff des verantwortlichen Bearbeiters erforderlich macht. In Produktionsworkflows wird idealerweise das Workflowsystem einen (sub)optimalen, korrekten Vorschlag zur Reorganisation der Abarbeitung anbieten, was im allgemeinen unter engen zeitlichen Bedingungen (*Nearline*–Betrieb) erfolgen muß (siehe auch Abschnitt 1.1.2.3 – Unterstützung durch Assistenz und Planung). Im Unterschied dazu spielen die Zeitbedingungen, die an die Abarbeitung eines Workflows im wissenschaftlichen Bereich gesetzt werden, keine so wichtige Rolle. Dafür ist es aufgrund des experimentelleren Charakters der Workflows schwieriger, das weitere Vorgehen zu bestimmen, und der weitere Ablauf kann im allgemeinen nur vom Bearbeiter ermittelt werden.

Neben klassischen Experimenten spielt auch der Bereich Entwicklung im wissenschaftlich–technischen Umfeld eine wichtige Rolle. In diesem Bereich haben sich, je nach Anwendungsgebiet, eine Reihe von Vorgehensmodellen etabliert, welche das ablauforganisatorische Konzept einer Entwicklung bestimmen. Hauptziel hierbei ist es, den komplexen Entwicklungsprozeß in kleinere, klar definierte Einheiten zu unterteilen. So können bei den bekannten Vorgehensmodellen im Bereich Software Entwicklung (*Wasserfallmodell*, *Spiralmodell* und *Prototypenmodell*) die einzelnen Objekte eines Workflows leicht identifiziert werden. Ein Vergleich zwischen administrativen Workflows und Vorgehensmodellen bei der Softwareentwicklung wird z. B. in [Chr95] geführt. Als Ergebnis des Vergleichs wird eine so hohe Ähnlichkeit zwischen beiden Prozeßmodellen festgestellt, daß sie mit denselben Prozeßmechanismen behandelt werden können. Neben den Gemeinsamkeiten finden sich Unterschiede im Prozeßcharakter, der in der Software-Entwicklung größere Freiheiten bei den Ausführungssequenzen zuläßt. Die Sequenzen werden im Gegensatz zu administrativen Workflows, bei denen die Reihenfolge der Ausführungsschritte durch einen expliziten Kontrollfluß fest vorgeschrieben ist, hauptsächlich durch Abhängigkeiten in den Daten vorgegeben. Dies erlaubt größere Freiheiten bei der Ausführung der Einzelschritte. Weiterhin sind Wiederholungen einzelner Schritte zulässig, was bei administrativen Workflows eher die Ausnahme darstellt. In [Chr95] wird weiterhin die Wichtigkeit der Integration von Software-

Entwicklung und administrativen Prozessen im Zusammenhang mit einer Zertifizierung nach ISO 9000 [Int87] betont, in deren Rahmen komplette Geschäftsprozesse beschrieben werden müssen. Die Eignung von Workflow Mechanismen in der Software-Entwicklung wird weiterhin in [Obe94] an einem konkreten System (*INCOME/STAR*) gezeigt.

Die Forderung nach hoher Flexibilität bei der Durchführung von technischen Entwicklungen deckt sich mit der Anforderung im Bereich wissenschaftlicher Experimente.

Beobachtung 4:

Durch die großen Freiheitsgrade bei den Ausführungssequenzen werden hohe Anforderungen an die Flexibilität eines Systems gestellt.

Ein weiterer elementarer Unterschied zu Workflows im administrativen Geschäftsbereich liegt in dem Ziel, das durch den Einsatz eines WFMS erreicht werden soll. Liegt der Fokus im administrativen Bereich hauptsächlich darin, eine Steigerung des Durchsatzes oder eine Kostenreduktion zu erreichen, so liegt der Grund für den Einsatz im wissenschaftlich–technischen, neben den zuvor genannten Gründen, hauptsächlich darin, ein Experiment oder eine Entwicklung begleitend zu dokumentieren, mit dem Ziel der Wiederholbarkeit bei Experimenten und damit der Sicherung der Reproduzierbarkeit der erzielten Ergebnisse. Dies spielt, auch speziell bei Produktionsworkflows, vor dem Hintergrund einer immer wichtiger werdenden Qualitätskontrolle eine tragende Rolle.

Eine weitere Charakteristik wissenschaftlicher Workflows ist ihr Übergang vom initialen, hoch dynamischen experimentellen Workflow mit hoher Benutzerinteraktion und Änderungshäufigkeit in den Abläufen während der F&E[21]–Arbeiten zu einem Produkt, hin zu einer Konsolidierung in Richtung Produktionsworkflow, mit höherem Automatisierungsgrad und geringerer Benutzerinteraktion [SV96], beim Übergang der entwickelten Verfahren in die Produktion. Die begleitende Dokumentation der Vorgehensweisen bereits während der F&E Phase innerhalb eines Workflow-Management-Systems kann zu einem weitestgehend automatischen Übergang in die Produktionsworkflows führen.

Zusammenfassung der Anforderungen

Im folgenden sollen die Anforderungen an ein wissenschaftlich–technisches Workflowsystem noch einmal, basierend auf den getätigten Beobachtungen, zusammengefaßt werden.

Einsatzumgebung: Die Rechnerumgebung in Labors und Instituten ist weitaus heterogener als im Bürobereich, wo oft nur Produkte eines Herstellers eingesetzt werden. Dies bezieht sich sowohl auf die eingesetzten unterschiedlichen Rechnerarchitekturen als auch auf die darauf ablaufenden Betriebssysteme und Anwendungsprogramme. Auch spielt die Anbindung externer Geräte eine weitaus größere Rolle. Ein System, das in diesem Umfeld eingesetzt werden soll, muß sowohl auf den unterschiedlichen Plattformen ablauffähig sein, als auch Probleme, die eine verteilte, heterogene Rechnerumgebung mit sich bringt, bewältigen.

Applikationsintegration: Die Anbindung externer Anwendungen[22] stellt eine der Hauptanforderungen an ein Workflowsystem dar, die in den meisten existierenden Systemen ver-

[21] **F**orschung und **E**ntwicklung

[22] sogenannte *Legacy* Anwendungen

nachlässigt wird und somit ein Haupthemmnis bei der Einführung von Workflowsystemen darstellt [TG97, JET97, Eur98]. Workflowsysteme im wissenschaftlich–technischen Umfeld müssen die Einbindung komplexer, bereits vorhandener Software erlauben.

Datenverwaltung: Wissenschaftliche Experimente sind um Daten herum aufgebaut. Diese bilden die Arbeitsgrundlage der durchzuführenden Workflows. Sie sind hoch komplex und in sehr großer Anzahl vorhanden. Das zu realisierende Workflowsystem muß demnach über eine leistungsfähige Datenhaltungskomponente verfügen.

Flexibiliät bei der Durchführung: Das zu realisierende System muß Freiheitsgrade bei der Durchführung der Workflows gestatten. Dies beinhaltet sowohl dynamische Änderungen an den Arbeitsabläufen, als auch große Freiheitsgrade bei der Abarbeitung. Weiter sollte das System verhaltensneutral sein, d. h. einzelne Workflows sollten mit unterschiedlichen Vorgehens–Modellen ausgestattet werden können.

Im folgenden Abschnitt sollen die dem Autor bekannten Systeme zur Unterstützung wissenschaftlich–technischen Arbeitsabläufe vorgestellt und ihre Eignung gemäß den oben formulierten Anforderungen untersucht werden. Hierbei handelt es sich sowohl um Workflowsysteme, welche um Elemente für den Einsatz im wissenschaftlich–technischen Umfeld erweitert wurden, als auch um Systeme aus dem wissenschaftlichen– oder Laborbereich, die um Merkmale von Workflowsystemen erweitert wurden.

1.2.3.3 Existierende Systeme für den Einsatz im wissenschaftlich–technischen Umfeld

Wie in der Einführung von Abschnitt 1.2 erwähnt, ist das Haupteinsatzgebiet von existierenden Workflowsystemen der Organisations– oder Geschäftsbereich. Daneben existieren aber auch noch einzelne experimentelle Systeme aus dem Forschungsbereich, deren Einsatzgebiet speziell im wissenschaftlich–technischen Bereich liegt. Die Auswahl der Systeme erfolgte unter dem Gesichtspunkt der größtmöglichen Generalität der Systeme, d. h. einem möglichst breiten Einsatzgebiet.

WASA

WASA (**W**orkflow–based **A**rchitecture to support **S**cientific **A**pplications) [BMVW95, MWBM96, VWW96] wurde am Institut für Wirtschaftsinformatik der Universität Münster entwickelt und wird im Bereich Molekularbiologie und pflanzliche Ökosysteme eingesetzt. Die Architektur von *WASA* erweitert ein kommerzielles Workflowsystem (IBM *FlowMark*) um zusätzliche Features, wie etwa *Decision Support*, zusätzliche Analyse Tools, Benutzerschnittstellen und Datenbankfunktionalität. Die Benutzerschnittstellen werden durch WWW–Browser und Java–Applets realisiert, was das System sehr plattformunabhängig macht. Die Datenbank ist in die Bereiche applikationsspezifische Daten und workflowspezifische Daten unterteilt und wird über ODBC bzw. CORBA Mechanismen angesprochen, so daß eine Trennung der Datenbank von der eigentlichen *WASA* Architektur gewährleistet ist. *WASA* erlaubt die dynamische Modifikation von Workflowspezifikationen. Änderungen wirken sich jedoch stets auf alle sich momentan in Arbeit befindlichen Workflows (Instanzen) des Systems aus. Dadurch ist ein Einsatz nur dann sinnvoll, wenn nur jeweils eine Instanz im System aktiv ist, um nicht andere Instanzen negativ zu beeinträchtigen. Eine Anbindung externer Applikationen ist nicht explizit realisiert, da davon ausgegangen wird, daß Anwendungen in Form von Java–Applets oder Java–Applikationen

realisiert werden. Bei der Realisierung der Workflowschemata unterliegen diese denselben Einschränkungen, wie es bei *FlowMark* der Fall ist. So können keine Schleifenkonstrukte realisiert werden, was bei wissenschaftlich–technischen Anwendungen sehr wichtig ist.

ZOO

ZOO [AIL98, ILGP96, Ioa98] wurde an der Universität von Wisconsin entwickelt. Es ist in C++ mit einer darunterliegenden *Informix* Datenbank realisiert. Im Gegensatz zu *WASA* liegt der Ursprung von *ZOO* im Bereich der Verwaltung wissenschaftlicher Datenbestände und wurde im nachhinein um Workflow Funktionalität erweitert. Aus diesem Grund ist es auch nicht verwunderlich, daß die zentrale Komponente des Systems ein Datenbanksystem ist. Die Philosophie des Ansatzes liegt darin, einen Hauptteil der Funktionalität des Workflowsystem vom Datenbanksystem zur Verfügung stellen zu lassen und die fehlende Funktionalität mit relativ geringem Aufwand oberhalb der Datenbank zu realisieren [AIL98]. Das System basiert auf einer Sammlung von Modulen z. B. für die Formulierung deklarativer Anfragen (FOX), einem Visualisierungstool für Datenbankobjekte (FROG), einem Experimentenmanager (EMU) und einem Schemamanager (OPPOSUM), die alle um die zentrale Datenbank (HORSE) gruppiert sind. Beim Einsatz von *ZOO* wird vorausgesetzt, daß die Daten sämtlicher, im Rahmen eines Workflows einzusetzenden, Softwarewerkzeuge, in der zentralen Datenbank HORSE abgelegt werden. Dies schränkt die Auswahl der einzusetzenden Applikationen auf solche Anwendungen ein, die speziell auf diesem Datenbanksystem aufsetzen, was den Einsatz von kommerziellen Werkzeugen praktisch ausschließt. *ZOO* wird als *generic Desktop Experiment Management Environment* (DEME) bezeichnet, das durch Anpassungen an die speziellen Laboranforderungen in eine *Customized Desktop Experiment Management Environment* (CDEME) adaptiert wird, d. h. das System ist, wie *WASA*, generisch in bezug auf die Domäne, innerhalb der es eingesetzt wird.

LabBase

LabBase [GRS94a, BSR96, RSG95, GRS95] wird als Datenbank und Arbeitsumgebung im Rahmen des *Genome*[23] Projekt am Whitehead Institute/MIT Center for Genome Research eingesetzt. *LabBase* ist für den Einsatz innerhalb von Laborumgebungen konzipiert und kann somit als ein **L**abor **I**nformation und **M**anagement **S**ystem (LIMS) angesehen werden, allerdings mit dem Unterschied, daß es verstärkt im Forschungsbereich eingesetzt wird und daher flexibler in der Handhabung sein muß, als es bei herkömmlichen LIMS der Fall ist. Realisiert ist das System in C++ mit Integration eines Datenbanksystems und enthält weiterhin eine Schnittstelle zu einer Skriptsprache. An diese Schnittstelle können z. B. Datenbankanfragen formuliert werden, so daß das System offen für Erweiterungen ist. Das Datenmodell von *LabBase* ist speziell auf den Einsatz im Rahmen des Genome-Projekts ausgelegt. So gibt es spezielle Datentypen und Operatoren zur Verwaltung der DNA–Sequenzen, was aber das Einsatzgebiet auf diesen Bereich einschränkt.

Wie man leicht erkennen kann, weisen all die oben genannten Systeme eine Reihe von Einschränkungen in bezug auf die in Abschnitt 1.2.3.2 formulierten Anforderungen auf. Insbeson-

[23] Das *Human Genome* Projekt ist eine weltweite Forschungsinitiative, um das menschliche Erbgut auf molekularer Ebene zu entschlüsseln. Ein Ziel ist es, die Funktionen des menschlichen Erbgutes zu verstehen und neue Erkenntnisse zur Behandlung genetischer Krankheiten zu erlangen.

dere bei der Einbindung externer Applikationen haben die obigen Produkte große Schwächen. Während die ersten beiden Produkte nur die Integration spezieller, für das Workflowsystem entwickelter, Programme erlauben, stellt *LabBase* zumindest eine Schnittstelle zu einer Skriptsprache bereit, über die Anwendungen angebunden werden können. Die Möglichkeit, Programme ohne zusätzlichen Programmieraufwand auf entfernten Rechnern auszuführen, besteht bei keinem der Produkte. Weitere Einschränkungen ergeben sich im Einsatzbereich der Systeme, die teilweise auf bestimmte Anwendungsdomänen spezialisiert sind.

Zusammenfassend kann gesagt werden, daß der Einsatz von Workflowsystemen im wissenschaftlich–technischen Umfeld sehr erfolgversprechend ist. Allerdings werden höhere bzw. zusätzliche Anforderungen an solche Systeme gestellt, die auch nicht von heute vorhandenen wissenschaftlichen Workflowsystemen abgedeckt werden können.

1.3 Aufgabenstellung und Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist die Realisierung von Softwaremodulen, die in der Lage sind, innerhalb von komplexen, heterogenen Arbeitsumgebungen, die Arbeiten von Wissenschaftlern und Entwicklern zu unterstützen.

Durch den Einsatz in einem Umfeld, das sich durch ein hohes Maß an Unsicherheiten und Freiheitsgraden in bezug auf Planung und Ablauf auszeichnet, können Mechanismen, wie sie in heutigen Workflowsystemen vorkommen, nicht eingesetzt werden, sondern es müssen flexiblere Mechanismen zur Verfügung gestellt werden, die es dem Anwender erlauben, Lösungen speziell für seine Aufgabenstellung zu formulieren. Dies impliziert Forderungen nach ausgereiften Schnittstellen sowohl zur Anbindung von externen Applikationen als auch für den Zugriff auf das System zur Laufzeit.

Die Workflowtechnologie stellt noch ein relativ junges Forschungsgebiet dar, innerhalb dem noch eine Reihe von ungelösten Problemen, die momentan Gegenstand aktueller Forschung sind, existieren. Beispiele sind Transaktionsverwaltung [AAEA+96, BHP96, BH96, Moh94], Ausnahmebehandlung [HA98a, HA98b, Lie98] oder dynamische Prozeßbeschreibung [CCPP96]. Aus diesem Grund konzentriert sich die vorliegende Arbeit auf folgendes Ziel:

Die Entwicklung und Realisierung eines Konzeptes für ein komponentenorientiertes Baukastensystem zum Aufbau von Workflowsystemen, mit Berücksichtigung der speziellen Anforderungen eines späteren Einsatzes im wissenschaftlich–technischen Umfeld.
Diese Aufgabe untergliedert sich in die folgenden Teilziele:

1. Es soll eine voll verteilte Architektur entwickelt werden, um von vornherein die Einschränkungen die durch eine zentrale Datenbank gegeben sind zu vermeiden.
2. Es soll für die zu realisierenden Komponenten eine Basisfunktionalität entwickelt werden.
3. Aufbauend auf der im vorherigen Punkt entwickelten Basisfunktionalität sollen eine Reihe von verfahrensunabhängigen Komponenten konzipiert und entwickelt werden.

Dieser Baukasten soll es zum einen den Experten[24] erlauben, maßgeschneiderte Workflowsysteme zu realisieren und es andererseits auch Entwicklern oder Wissenschaftlern ermöglichen, effektive Arbeitsumgebungen für konkrete Projekte oder Experimente aufzubauen. Bei der Realisierung

[24] Workflow-Entwickler oder beispielsweise auch spezialisierte Mitarbeiter einer Unternehmensberatung.

des Baukastens werden insbesondere die Anforderungen, welche der Einsatz im wissenschaftlich–technischen Bereich mit sich bringt, beachtet (Abschnitt 1.2.3) und die dafür notwendigen Basistechnologien als Bausteine in den Baukasten integriert. Weiterhin soll der Baukasten so realisiert werden, daß eine Großzahl der Kritikpunkte, wie sie an heutigen Systemen (Abschnitte 1.2.1 und 1.2.3.3) laut werden, schon durch die Architektur vermieden werden. Die Schnittstelle des Baukastens liegt unterhalb von existierenden Vorgehens– oder Verfahrensmodellen, um bei der Realisierung von Workflowsystemen von vornherein möglichst keinen Einschränkungen, bezüglich eines gewählten Verfahrens oder Vorgehensmodells, zu unterliegen.

Die Realisierung des Baukasten konzentriert sich auf die Kernkomponenten, d. h. Laufzeitkomponenten ohne *Buildtime* und ohne Bereitstellung von Benutzeroberflächen. Dafür stellt das System eine Reihe von Schnittstellen zur Verfügung, die den Aufbau der fehlenden Komponenten erlauben. Hierbei wird besonderer Wert darauf gelegt, daß der Aufbau auf einfache und schnelle Art und Weise erfolgen kann. Zur Realisierung soll eine Technik angewandt werden, die in den letzten Jahren immer mehr an Bedeutung gewonnen hat und mit *komponentenorientierte Softwareentwicklung* [Mei97, Szy98, Sch96a] bezeichnet wird. Kernpunkt dieser Technologie ist es, vorgefertigte Komponenten mit fest definierter Schnittstelle miteinander zu einem Gesamtsystem zu verknüpfen. Bei der Realisierung soll weiterhin besonderes Augenmerk auf die Integration aktueller Technologien wie *Java*, Internet, verteilte Rechnerumgebungen, etc. gerichtet werden.

Durch die Realisierung als Baukasten mit vorgefertigten Basiskomponenten ergeben sich folgende Vorteile gegenüber einem kompletten Workflowsystem:

- Optimale Adaption an konkrete Anforderungen.
- Keine Festlegung auf bestimmte Vorgehensmodelle oder Methoden.
- Offenes, erweiterbares System.
- Eignung als Basisplattform für Forschungen im Bereich Workflow-Management.
- Weitaus größerer Einsatzbereich, als ein komplett realisiertes Workflowsystem.
- Aufbau von maßgeschneiderten, komplexen Entwicklungsumgebungen.

Zusammengefaßt kann gesagt werden, daß es das Ziel der vorliegenden Arbeit ist, einen komponentenorientierten Baukasten zum Aufbau von Workflow-Management-Systemen zu entwickeln, mit Hinblick auf ein späteres Einsatzgebiet im wissenschaftlich–technischen Umfeld. Besondere Beachtung finden hierbei Aspekte wie Offenheit und Verteilung des Systems sowie Einsatz von Techniken des *Rapid Application Development* (RAD). Die Architektur, die angestrebt wird, ist in Abbildung 1.5, in Anlehnung – und Abgrenzung – zu der Architektur in Abbildung 1.2 (d) zu sehen. Das Bild macht deutlich, daß es sich keinesfalls um ein monolithisches, in sich geschlossenes Gesamtsystem handelt, sondern um ein offenes, mit verschiedenen Schnittstellen zur Außenwelt versehenes System, das als Baukasten realisiert ist. Die in Abschnitt 1.2.3 ermittelten speziellen Anforderungen an ein solches System in Bezug auf *Anbindung externer Applikationen*, *Ablaufsteuerung* und *Datenorganisation* spielen in diesem Fall eine zentrale Rolle und stellen Kernpunkte der zu realisierenden Aufgabe dar.

Die Hauptkomponenten des Baukastens sind: *Container* zur Datenorganisation, *Aktivitäten* zur Realisierung der Ablaufsteuerung und die *Toolservices* zur Anbindung externer Anwendungen.

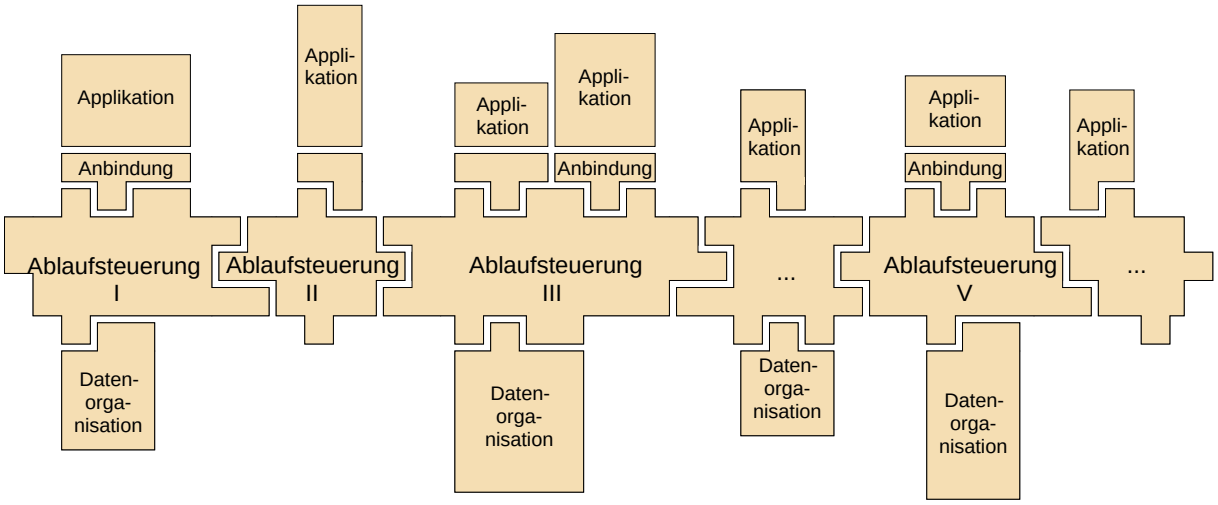


Abbildung 1.5: *Architektur von W∗FLOW*

Neuartig an dem hier vorgestellten Konzept ist vor allem die Integration der verschiedenen Technologien innerhalb eines Baukastens in Form von eigenständigen, mittels einer netzwerktransparenten Skriptsprachenschnittstelle miteinander kombinierbaren, Komponenten. Dies ermöglicht sowohl die Erstellung von flexiblen, leicht änderbaren Arbeitsumgebungen, als auch die Entwicklung kompletter Workflowsysteme auf Basis der vorgestellten Komponenten.

In den folgenden Kapiteln erfolgt nun die Vorstellung des zu realisierenden Systems *W∗FLOW*[25]. Zu Beginn des folgenden Kapitels wird zunächst das zu realisierende Konzept erarbeitet. Hier werden in Abschnitt 2.1 die verwendeten Technologien vorgestellt und anschließend in Abschnitt 2.2 die Gesamtarchitektur entwickelt. Hierbei werden Aspekte der Aufteilung in einzelne Komponenten, der Verteilung und der Kommunikation besprochen. Abgeschlossen wird dieses Teilkapitel mit Erläuterungen zu getroffenen Realisierungsentscheidungen, wie Auswahl der Schnittstellensprachen, der Datenbank und des zugrundeliegenden Kommunikationsmechanismus. Abschnitt 2.3 stellt dann die Basisfunktionalität der Komponenten, die in den sich darauffolgenden Kapiteln konzipiert werden, vor.
Die Vorstellung der einzelnen Komponenten beginnt in Kapitel 3 mit der *Containerkomponente*, die für die Verwaltung der Datenstrukturen eingesetzt wird und der *Aktivitätenkomponente*, welche die Durchführung von Workflows realisiert. Als letzte Einzelkomponenten werden in Kapitel 4 die *Toolservices* vorgestellt, die aus *Toolserver* und *Toolstarter* bestehen, und den Applikationsaufruf in heterogenen Umgebungen realisieren.

Kapitel 5 zeigt an einem konkreten Beispiel, das auf die in den vorherigen Kapiteln vorgestellten Beispielfragmenten aufsetzt, den Aufbau einer konkreten Entwicklungsumgebung. Diese wurde im Rahmen des *PRAXIS*–Projektes [GD96, DGS98a, DGS98b, DGS98c] am Institut für Angewandte Informatik zum Teil prototypisch realisiert und zeigt die Tragfähigkeit des gewählten Ansatzes.

Den Abschluß der Arbeit bilden eine Zusammenfassung und ein Forschungsausblick.

[25] gesprochen "Wildflow" – das *–Zeichen, auch als *Wildcard* bezeichnet, soll die Freiheitsgrade bei der Realisierung von Workflowsystemen verdeutlichen.

Kapitel 2

Konzept eines komponenten-orientierten, verteilten neuen Workflow-Management-Systems

Um die Zielsetzungen aus dem vorherigen Kapitel erfüllen zu können, kombiniert $\mathcal{W}*FLOW$ innerhalb eines komponentenorientierten Baukastens eine Reihe von Konzepten aus der Informatik. Diese sind im einzelnen wohlbekannt, können in einer geeigneten Kombination miteinander aber zu einem neuartigen Gesamtsystem zusammengefaßt werden, das die vorteilhaften Eigenschaften der Konzepte in optimaler Weise verbindet. In den folgenden Abschnitten erfolgt zuerst die Vorstellung der zu integrierenden Einzelkomponenten und der Technologie, mit der die Komponenten verbunden werden können, und anschließend wird bei der Beschreibung der $\mathcal{W}*FLOW$ Architektur auf die Integrationsaspekte eingegangen.

2.1 Basistechnologien

2.1.1 Workflow Technologie

Da es sich bei $\mathcal{W}*FLOW$ um einen Baukasten zum Aufbau von Workflowsystemen handelt, überrascht es nicht, daß ein Workflowkonzept integraler Bestandteil des Systems ist. Die durch dieses Konzept definierten Funktionalitäten können, in Anlehnung an [JB96], in sogenannte *Aspekte* unterteilt werden. Ein Aspekt bezeichnet hierbei eine bestimmte Sichtweise auf die Workflow-Thematik. Die Idee ist, die Inhalte bei der Modellierung von Workflows zu gliedern. Unterschieden wird üblicherweise in funktionsbezogene, informationsbezogene, verhaltensbezogene, operationsbezogene und organisationsbezogene Aspekte. Je nach Anforderung an das Workflowsystem werden zum Teil auch noch historienbezogene und transaktionsbezogene Aspekte genannt [Bö97].

So versteht man unter dem funktionsbezogenen Aspekt die Arbeitsstrukturierung, d. h. die Beschreibung der Arbeitsvorgänge. Der verhaltensbezogene Aspekt bestimmt die kausalen Gesichtspunkte, wie die Reihenfolge der Abarbeitung von Arbeitsschritten, Nebenläufigkeit, alternative Vorgehensweisen oder asynchrone Reaktionen auf Ereignisse. Der informationsbezogene Aspekt beschäftigt sich mit den Beziehungen zwischen Daten und Arbeitsschritten, Daten und Werkzeugen und die Beschreibung des Datentransfers zwischen Arbeitsschritten.
In $\mathcal{W}*FLOW$ werden diese Aspekte durch eine eigene Komponente, der Aktivität, erbracht, die weiterhin noch die transaktionsbezogenen Aspekte erfüllt.

Bei vielen Workflowsystemen enthält der informationsbezogene Aspekt auch Mechanismen zur Speicherung von Anwendungsdaten [JBS97, JB96]. Dieser Aspekt ist aber in den meisten Workflowsystemen nur mit einer minimalen Funktionalität realisiert und kann von einem auf Datenmanagement spezialisierten System (Datenbank) viel besser wahrgenommen werden. Das Management der Anwendungsdaten ist daher in $\mathcal{W}*FLOW$ bewußt nicht Bestandteil des Workflowkonzeptes, sondern wird von einer eigenständigen Datenhaltungskomponente, den $\mathcal{W}*FLOW$–Containern, wahrgenommen. Die Beziehung der Daten zu den Arbeitsvorgängen sind aber, wie schon beschrieben, im Workflowkonzept enthalten.

Üblicherweise erfüllt ein Workflowkonzept noch weitere Funktionalitäten. Zwei wesentliche Bereiche sind hier Mechanismen zur Anbindung externer Programme zur Ausführung von Arbeitsschritten (operativer Aspekt) und die Beschreibung der Organisationsstrukturen der von einem Workflow umfaßten Organisationseinheiten (organisatorischer Aspekt).

Der operative Aspekt wird in $\mathcal{W}*FLOW$ durch zwei eigenständige Komponenten, den *Toolservices*, realisiert. Die Konzepte hierfür stammen dabei im wesentlichen aus der Entwicklungszeit von Applikations-Frameworks und werden weiter unten im Abschnitt "Applikations-Framework" vorgestellt.

Der organisatorische Aspekt soll gemäß der Architektur von $\mathcal{W}*FLOW$ ebenfalls durch eine gesonderte Komponente bereitgestellt werden. Dieser Aspekt unterscheidet sich dahingehend von den anderen Aspekten, daß er sehr abhängig von den im jeweiligen Einsatzgebiet vorherrschenden organisatorischen Strukturen ist. Es gibt inzwischen zwar theoretische Ansätze [Bußß97] eine reale Organisationsstruktur frei modellieren zu können und Zuweisungsregeln zu definieren, eine Realisierung als eigene Komponente in $\mathcal{W}*FLOW$ wird aber nicht weiter betrachtet.

2.1.2 Datenmanagement Technologie

Da $\mathcal{W}*FLOW$ für den Umgang mit komplexen Anwendungsdatenstrukturen und –beziehungen realisiert werden soll, wird hier eine eigenständige Datenhaltungskomponente, auf Basis einer objektorientierten Datenbank, eingesetzt und die Aufgabe der Datenverwaltung an diese Komponente delegiert. Im Rahmen dieser Arbeit wird die Objektdatenbank *ObjectStore* der Firma Object Design eingesetzt. Die Gründe, die zur Auswahl dieser Datenbank geführt haben, werden in Abschnitt 2.2.2.2 dargelegt. Zusätzlich zu Standard-Datenbankfunktionalitäten wie Transaktionsmechanismen, Suchanfragen, etc. müssen allerdings Konzepte zur Organisation und Modellierung von Datenbeziehungen, zur Addition semantischer Zusatzinformation zu existierenden Datensätzen sowie ein allgemeines *Link*–Konzept, basierend auf der Referenzierung von externen Datensätzen, das es erlaubt, Daten auch außerhalb der Datenbank zu verwalten, hinzugefügt werden. $\mathcal{W}*FLOW$ verwendet eine Referenzierung basierend auf URLs[1] [BLMM94, Fie95], die sich als Adressierungsform innerhalb des World Wide Web durchgesetzt haben, und auch von Seiten der Internetbrowser eine Unterstützung gewährleistet wird. Dieses Datenmanagement Konzept erlaubt es, zum einen die Applikationsdaten datenbanktechnisch zu erfassen, andererseits stehen die Daten den Anwendungen in gewohnter Form zu Verfügung.

2.1.3 Applikations-Framework

Neben den eigentlichen Workflowsystemen wurden in den achtziger und neunziger Jahren eine Reihe von Frameworksystemen entwickelt, deren Aufgabe die Integration von Programmen im gleichen Anwendungsumfeld ist. Als Beispiele können hier etwa Produkte genannt werden, die

[1] **U**niform **R**essource **L**ocator (siehe Glossar Seite 184)

im Rahmen der CAD Framework Initiative (CFI) entstanden sind.[2] Ziel der Initiative war es, eine Infrastruktur zu entwickeln, welche die Entwicklung anwendungsspezifischer Entwurfsumgebungen erlaubt. Eine Aufgabe der entstandenen Frameworks ist es, eine Schnittstelle bereitzustellen, welche die Einkapselung von CAD Werkzeugen erlaubt. Dies ermöglicht dem Anwender eine einheitliche Sicht auf die verwendeten Programme über verschiedene Betriebssystemgrenzen hinweg, ohne daß er sich Gedanken über die konkrete Kommandosyntax und Aufrufkonventionen machen muß [CFI92].

Der Ansatz einer einheitlichen Behandlung von verschiedenen Softwarewerkzeugen, über Hard– und Softwaregrenzen hinweg, wird in $\mathcal{W}*FLOW$ ebenfalls aufgegriffen. Dies wird durch die *Toolservice* Komponenten (Kapitel 4) realisiert. Neben der Bereitstellung einer einheitlichen Plattform und der Versorgung mit Eingabedaten und Parametern, wie es auch von den Applikationsframeworks ermöglicht wird, sind im Rahmen der *Toolservices* noch folgende Leistungsmerkmale vorhanden:

- Unterstützung von lokalem und entferntem Aufruf von Programmen.
- Integration von Dokumentation zu den Werkzeugen.
- Flexible Benutzerkonfiguration, die den einzelnen Benutzern ermöglicht, sich ihr Arbeitsumfeld, in Bezug auf die einzusetzenden Werkzeuge, selbst zu konfigurieren.

2.1.4 Komponentenbasierte Programmierung und Skripting

Nachdem in den vorherigen Abschnitten die Technologien vorgestellt worden sind, die innerhalb des Systems als Komponenten realisiert werden sollen, werden im folgenden die Mechanismen beschrieben, mit denen auf die Komponenten zugegriffen werden kann und es wird gezeigt, wie die Komponenten integriert werden können. Zuvor soll aber zuerst der Begriff der Komponente und Komponentenschnittstelle präzisiert werden, da dieser Begriff Voraussetzung für das Verständnis des nachfolgenden Textes ist.

2.1.4.1 Softwarekomponenten

Die Begriffe "Softwarekomponenten" oder "Componentware" haben in den letzten Jahren stark an Bedeutung gewonnen. Nicht zuletzt tauchen sie immer wieder im Zusammenhang mit Produkten wie *JavaBeans* [Eng97], *ActiveX* [Den97], *Delphi* [Ley95] und *OpenDoc* [FM96] auf. Ziel all dieser Produkte und Konzepte ist es, den Enwicklungsprozeß von Anwendungssystemen durch den Einsatz vorgefertigter Bauteile, den sogenannten Komponenten, zu vereinfachen. Vorteile verspricht man sich unter anderem bei der Entwicklungsdauer, der Qualität und Wartbarkeit des entstandenen Produktes und somit insgesamt eine Reduzierung der Entwicklungskosten [Mei97].

In [NL97] wird eine Softwarekomponente wie folgt definiert:

> **Definition 2.1** *"Eine* Softwarekomponente *ist ein* Element eines Komponentenframeworks*".*

Diese Definition legt auch gleich den Einsatzbereich einer Komponente fest, nämlich das Zusammenspiel mit anderen Komponenten im Rahmen eines Baukastens, der die Struktur der späteren Anwendung festlegt.

[2] z. B.: *JESSI COMMON FRAMEWORK* – Esprit Project 7364 und *Merlin's Framework* von Knights Technology Inc.

Die gesamte Architektur der späteren Anwendung wird durch Instanziierung von Softwarekomponenten des zur Verfügung gestellten Baukastens festgelegt.

Weiterhin wird definiert, daß eine Komponente eine Softwareschnittstelle besitzen muß, über welche die Dienste, die die Komponente anbietet, angesprochen werden können. Diese Schnittstelle (API) stellt einen Satz von Funktionen oder Methoden bereit, welche den Zugriff auf die Komponente erlauben.

Die Gesamtfunktionalität, die von einer komponentenbasierten API zur Verfügung gestellt wird, kann in mehrere funktionale Teilbereiche untergliedert werden, die im folgenden kurz angesprochen werden sollen:

Eigenschaften (Properties): Die *Properties* einer Instanz definieren deren sichtbare Eigenschaften, das Verhalten sowie eventuell die graphische Repräsentation [Eng97]. Bei den *Properties* handelt es sich um den Datenteil eines Objekts, der aus diskreten, über den Namen anzusprechende Attribute und deren Werte besteht. Der Zugriff auf die *Properties* erfolgt nicht direkt, sondern über Zugriffsmethoden. Hierbei handelt es sich um zumeist einfache Methoden, die den lesenden und schreibenden Zugriff auf ein Attribut erlauben, ohne daß die Kapselung des Objekts verletzt wird. In den meisten Fällen besitzen diese Methoden den Namen des jeweiligen Attributs, eventuell mit den Präfixen `get_` und `set_` versehen, um zwischen Lese- und Schreibzugriff zu unterscheiden.

Ereignisbehandlung: Mechanismus, der eine Benachrichtigung über Ereignisse zwischen Komponenten erlaubt. Hierbei wird zwischen Ereignisquelle, d. h. der Komponente, die ein Ereignis auslöst, und den "Interessenten" an Ereignissen, den Ereigniskonsumenten, unterschieden. Einzelne Komponenten können sich als "Interessent" an bestimmten Ereignissen bei einer anderen Komponente registrieren lassen. Anschließend werden sie über das Auftreten dieser Ereignisse informiert. Bei Ereignissen kann es sich beispielsweise um Zustandsänderungen der Komponente oder um externe Ereignisse, wie das Drücken einer Maustaste, handeln. Zusätzlich mit einem Ereignis wird noch ein Datenobjekt übertragen, das das Ereignis näher beschreibt.

Introspektion: Hierunter wird ein Mechanismus zur Ermittlung der zu einer bestimmmten Komponente gehörenden *Properties*, *Methoden* und *Ereignisse* verstanden. Der Mechanismus spielt an zwei unterschiedlichen Stellen eine Rolle:

1. Bei der Laufzeit einer Anwendung:Dies ermöglicht beispielsweise die automatische Ermittlung aller Zugriffsmethoden für die definierten *Properties*.
2. Bei der Entwicklungszeit: Im Rahmen einer Applikationsentwicklung mittels graphischer Enwicklungstools (siehe auch den nächsten Punkt *Customization*) erlaubt die Introspektion die automatische Ermittlung der Schnittstelle der Komponente und ermöglicht es so der Entwicklungsumgebung alle relevanten Informationen, welche für das Zusammenspiel mit anderen Komponenten von Belang sind, zu extrahieren und dem Entwickler zur Verfügung zu stellen.

Customization: Unter *Customization* wird die Bereitstellung von *Property*–Editoren im Rahmen der Entwicklung von Anwendungen mit graphischen *Application Builder Tools* verstanden. Hierbei werden von einem *Application Builder* mittels *Introspection* (s. o.) die *Properties* einer Komponente ausgelesen und entsprechende Editoren zur Manipulation der Werte bereitgestellt. Diese Werte definieren das spätere Verhalten bzw. Aussehen einer Komponente. Hierbei kann es sich, je nach Typ einer *Property*, um einfache Standard-

Editoren (z.B für das Setzen numerischer Werte) bzw. komponentenspezifische und vom Komponentenentwickler bereitszustellende Spezial-Editoren handeln.

Persistenz: Durch das *Persistenz*–Konzept ist es möglich, den internen Zustand einer Komponente zu speichern und aus dieser Information die Komponente wieder zu rekonstruieren. Im allgemeinen müssen hierbei die Werte der *Properties* und Teile des internen Zustands einer Komponente gespeichert werden.

Die Granularität einer Softwarekomponente kann von einfachen Bauelementen (z. B. einfache Dialogkomponenten wie Button oder Textfeld) bis hin zu kompletten Applikationen (z. B. eine *Excel* Tabelle innerhalb eines Word Dokumentes oder ein Filterprogramm im Rahmen einer *UNIX*–Pipe) reichen.

Im Falle von $\mathcal{W}$*FLOW* handelt es sich um komplexere Komponenten, die pro Komponente aus etwa 20–50 Klassen im objektorientierten Sinn aufgebaut sind.

In den folgenden Abschnitten sollen nun eine Reihe von Designentscheidungen für einen solchen komponentenbasierten Baukasten getroffen werden, die jeweils auf Grundlage der Zielanforderungen aus Abschnitt 1.3 erfolgen.

2.1.4.2 Baukasten-Programmiersprache

Es stellt sich nun die Frage, wie die Komponenten miteinander verknüpft werden. Dieser Vorgang wird im Englischen als *Gluing*[3] oder *Component Scripting* bezeichnet und wird durch eine Programmiersprache realisiert. Das Spektrum reicht hier von C++ auf der einen, bis hin zu Skript– und 4GL[4] Sprachen auf der anderen Seite [NL97]. Die Auswahl einer bestimmten Sprache hat große Auswirkungen auf die spätere Leistungsfähigkeit bzw. die Entwicklungsdauer einer Anwendung.

Prinzipiell kann jede Programmiersprache zum Aufbau der Anwendungen verwendet werden, sofern diese eine Schnittstelle zu der Sprache besitzt, in der die einzelnen Komponenten realisiert sind. Im einfachsten Fall handelt es sich dabei auch um die Sprache, in der die Komponenten realisiert sind. Im folgenden soll aber eine etwas differenzierte Betrachtung geführt werden, um eine möglichst geeignete Sprachform auszuwählen.

In [Ous98] wird zwischen zwei Arten von Sprachen unterschieden. Erstens die *Systemprogrammiersprachen* wie *C*, C++, *PL/1*, *Pascal*, etc. und zweitens *Skriptsprachen* wie *Rexx* [Cow85], *Python* [Lut96], *TCL/Tk* [Ous94], *Visual Basic* [Mic98] und *Perl* [WCS96]. Während erstere dazu geeignet sind, Datenstrukturen und Algorithmen auf Basis von Computerprimitiven wie beispielsweise Speicherplatz von Grund auf aufzubauen, eignen sich Skriptsprachen zum Verbinden bereits bestehender Komponenten. In [Ous98] werden die Unterschiede zwischen beiden Ansätzen, die daraus resultierenden Vor– und Nachteile und, abgeleitet daraus, die geeigneten Einsatzgebiete für die jeweilige Sprachform herausgearbeitet. Als Leitfaden ist ein Fragenkatalog enthalten, der neun Fragen enthält, die für die zu realisierende Anwendung zu beantworten sind. Die Antwort "Ja" auf die ersten sechs Fragen ist ein Hinweis auf den Einsatz einer Skriptsprache, bei den restlichen drei Fragen ist die Antwort "Ja" ein Anhaltspunkt eine Systemsprache einzusetzen. In Tabelle 2.1 sind die neun Fragen aufgelistet, die im folgenden Abschnitt diskutiert werden und als eine Entscheidung für eine bestimmte Sprachform dienen sollen.

[3] Engl: to glue - kleben. Gemeint ist hier das "Zusammenkleben" der einzelnen Komponenten zu einem Gesamtsystem.

[4] Fourth Generation language (siehe Glossar Seite 179)

Tabelle 2.1: *Fragenkatalog aus [Ous98]*

pro Skriptsprache	
1	Ist es Hauptaufgabe der Anwendung bereits existierende Komponenten zu verbinden?
2	Muß die Anwendung eine Vielzahl unterschiedlicher Aufgaben erledigen?
3	Enthält die Anwendung eine graphische Benutzeroberfläche?
4	Muß eine Reihe von String–Manipulationen durchgeführt werden?
5	Wird die Funktionalität der Anwendung schnell anwachsen?
6	Muß die Anwendung erweiterbar sein?

pro Systemsprache	
7	Implementiert die Anwendung komplexe Algorithmen oder Datenstrukturen?
8	Bearbeitet die Anwendung große Datenmengen (wie z. B. alle Pixel eines Bildes), so daß die Ausführungsdauer eine Rolle spielt?
9	Ist die Funktionalität der Anwendung wohldefiniert und nur einem langsamen Wandel unterzogen?

Bevor auf die einzelnen Fragen näher eingegangen wird, soll kurz der Einsatzbereich des Baukastens wiederholt werden, wie er in der Zielsetzung der Arbeit charakterisiert wurde.

Der Baukasten soll den Entwickler beim Aufbau komplexer Workflow-Management-Systeme unterstützen. Dazu enthält er eine Reihe von vorgefertigten Komponenten, aus denen die Anwendung realisiert wird. Er konzentriert sich dabei auf die Bereitstellung der Kernfunktionalität, ohne Benutzeroberflächen. Besondere Beachtung finden hierbei Aspekte wie Offenheit und Verteilung des Systems sowie Einsatz von Techniken des *Rapid Application Development* (RAD).

Eine Untersuchung des Anforderungskatalog im Hinblick auf die einzusetzende Sprache anhand der Fragen aus Tabelle 2.1 ergibt, daß die Fragen (1), (2), (3) und (6) sofort mit "Ja" beantwortet werden können. Frage (4) kann ebenfalls mit "Ja" beantwortet werden, da die Verwaltungsdaten zu allen möglichen Objekten in einem Workflowsystem zum großen Teil als Textstrings vorliegen und ein "Ja" auf Frage (5) folgt aus der Anforderung, daß der Baukasten als Basisplattform für weitere Forschungen und Entwicklungen im Bereich Workflow Management eingesetzt werden soll.

Aus der Überlegung heraus, daß ja gerade die im Rahmen dieser Arbeit entwickelten Komponenten und eventuell bereits bestehende Programme, in einem zu realisierenden Workflowsystem miteinander kombiniert werden sollen, kann keine der drei Fragen (7)–(9) mit "Ja" beantwortet werden. Frage (7) und (8) müssen verneint werden, da diese Anwendungen bereits als Programme existieren und nur im Rahmen einer Workflow-Anwendung in ein Gesamtkonzept integriert werden und Frage (9) muß nicht zuletzt aufgrund des wissenschaftlich–technischen Einsatzgebietes verneint werden.

Ousterhout geht in seinem Artikel auch speziell auf komponentenbasierte Baukästen ein und zeigt an ihnen, wie sich Systemprogrammiersprachen und Skriptsprachen ergänzen können. So sind Systemprogrammiersprachen hervorragend dazu geeignet, komplexe Komponenten aufzubauen, an die hohe Anforderungen im Bereich Performance gestellt werden, während das "Zu-

sammensetzen" der Komponenten idealerweise von einer Skriptsprache erledigt werden kann. Als Beispiel für erfolgreiche Komponentenframeworks wird das PC Umfeld genannt, wo im Gegensatz etwa zu *UNIX/CORBA* mit *Visual Basic* eine weit verbreitete Skriptsprache mit der Möglichkeit der Integration von Komponenten (*ActiveX*) zur Verfügung steht.

Insbesondere unter dem Aspekt des Rapid Applikation Development (RAD), d. h. des schnellen Aufbaus von Anwendungen, sind Skriptsprachen eindeutig die erste Wahl. In [Ous98] sind Vergleiche der Entwicklungszeiten zwischen Systemprogrammiersprachen und Skriptsprachen geführt, deren Ergebnis je nach Anwendungsgebiet zwischen dem Faktor 3,5 und 60 zugunsten der Skriptsprachen ausfallen. John Ousterhout äußert in seiner Schlußfolgerung die Hoffnung, daß Entwickler von Komponentenframeworks die Bedeutung einer Skriptsprache erkennen und ihren Nutzen daraus ziehen.

Zitat:

> *"I hope that designers of component frameworks will recognize the importance of scripting and ensure that their frameworks include not just facilities for creating components but also facilities for gluing them together".*[5]

2.1.4.3 Laufzeitschnittstelle

Eine weitere Forderung an wissenschaftlich-technische Workflowsysteme, die eng mit dem obigem Punkt zusammenhängt, ist die Offenheit gegenüber Erweiterungen des zu realisierenden Systems. Dieser Forderung wird durch die Definition einer Laufzeitschnittstelle Rechnung getragen, mittels der es möglich ist, zur Laufzeit auf die Informationen innerhalb der $\mathcal{W}*FLOW$–Engine[6] zuzugreifen.

Analog zur Baukasten-Programmiersprache muß auch für die Laufzeitschnittstelle eine geeignete Sprachform gefunden werden. Prinzipiell sind auch hier die beiden Möglichkeiten Systemprogrammiersprache oder Skriptsprache gegeben. So enthalten viele kommerzielle Workflow-Anwendungen APIs, die in Form von dynamischen Bibliotheken zu Anwendungen hinzugefügt werden können, um so auf die Funktionalität der Anwendung zugreifen zu können. Beispiele sind hier *COSA* von Software-Ley, *FlowMark* von IBM und *InConcert* von XSoft, die alle über eine Systemprogrammiersprachen API den Zugriff auf bestimmte Funktionen des Workflowservers erlauben.

Hauptaufgabe einer Laufzeitschnittstelle ist die Entwicklung von externen Anwendungen, die eng mit dem Workflowsystem interagieren können, da sie Zugriff auf Informationen des Systems haben. So können solche Anwendungen beispielsweise ihre Eingabedatensätze auslesen oder den momentanen Zustand einer Aktivität erkennen.

Bei den zuvor genannten Produkten wird diese Schnittstelle weiterhin zur Erweiterung des Programmsystems um spezifische Funktionseinheiten oder Benutzerschnittstellen eingesetzt. Dies wird bei einem Baukastensystem, wie $\mathcal{W}*FLOW$, aber bereits von der Baukasten-Programmiersprache erfüllt. Die Laufzeitschnittstelle in $\mathcal{W}*FLOW$ kann sich also darauf konzentrieren, an der Schnittstelle zu den Anwendungsprogrammen Zugriff auf das System zu erlauben.

Liegen beide Schnittstellen in der gleichen Sprache vor, so wird eine weitaus engere Kopplung

[5] Übersetzung: *"Ich hoffe, daß die Entwickler von komponentenbasierten Baukästen die wichtige Rolle von Skriptsprachen erkennen und ihre Baukästen nicht nur die Möglichkeit eröffnen, Komponenten zu erzeugen, sondern es auch erlauben, diese Komponenten zu verbinden.*

[6] wenn im folgenden von $\mathcal{W}*FLOW$–Engine gesprochen wird, so ist darunter ein aus den Komponenten aufgebautes System gemeint.

zwischen Workflowengine und externer Anwendung ermöglicht. So kann beispielsweise eine extern anzubindende Applikation, welche die zur Verfügung gestellte API nutzt, als Erweiterung der Funktionalität der $\mathcal{W}*FLOW$–Engine betrachtet werden. Das Interessante daran ist, daß die Funktionalitätserweiterung zur Laufzeit durchgeführt werden kann, da lediglich ein Programm, welches die $\mathcal{W}*FLOW$–API nutzt, als Applikation in einen Workflow integriert werden muß. Daneben kann mit der Schnittstelle ein *Wrapper*[7] um eine existierende Anwendung gelegt werden, um sie *workflow–konform* zu machen. Eine weitere Möglichkeit besteht darin, komplette, auf Basis von $\mathcal{W}*FLOW$ realisierte Anwendungssysteme, mittels der API zu koppeln. Der Hauptunterschied zu einer Erweiterung durch eine Systemsprachen API liegt darin, daß bei dieser eine eigenständige Entwicklungsumgebung für die entsprechende Systemsprache verfügbar sein muß, was weitaus aufwendiger und fehleranfälliger ist, als dies bei einer Skriptsprachen API der Fall ist.

Die Anwendung des Fragenkatalogs aus Tabelle 2.1, kommt ebenfalls zu dem Ergebnis, daß eine Skriptsprachenschnittstelle auch hier viel besser geeignet ist, als eine Systemsprachen API. Die Eignung einer Skriptsprache zum Aufbau von kompletten Anwendungen aus bereits vorhandenen Bausteinen wird weiterhin in [LS98c] am Beispiel der Skriptsprache Perl beschrieben. Im Gegensatz zu [Ous98], handelt es sich bei den Bausteinen nicht um Komponenten eines Baukastens, sondern um Komponenten im Sinne von vorhandenen Anwendungs– oder Betriebssystemprogrammen, die auf einfachste Weise mittels einer solchen Sprache kombiniert werden können. Ein weiterer Vorteil stellt die High–Level Schnittstelle der meisten Skriptsprachen zu systemnahen Funktionen des Betriebssystems, wie der Zugriff auf das Mailsystem oder Benutzer– und Passwortinformationen, dar.

Nach den Ausführungen aus Abschnitt 2.1.4.2 und Abschnitt 2.1.4.3, wird festgelegt, daß die Baukasten-Programmiersprache von $\mathcal{W}*FLOW$ auf Basis einer Skriptsprache realisiert wird und weiterhin eine Laufzeitschnittstelle, ebenfalls auf Basis derselben Skriptsprache, integriert wird. Aufgrund der obigen Ausführungen ergibt sich der Vorteil einer weitaus einfacheren und flexibleren Kopplung von Anwendungen an die zur Verfügung gestellte Skriptsprachen API, ohne die Notwendigkeit des Einsatzes einer Entwicklungsumgebung, und damit verbunden eines zusätzlichen Übersetzungs– und Einbindeschritts. Die zum Teil extremen Zeitunterschiede beim Vergleich der Entwicklungsdauer von Anwendungssystemen mittels Skript– bzw. Systemsprachen liefern ein weiteres Argument für die Bereitstellung einer Skriptsprachen API. Der Nachteil dieser Kopplung liegt in der bedeutend höheren Ausführungszeit von Skriptsprachen durch den Interpreter, was aber zu tolerieren ist, da die zeitkritischen Operatoren ja innerhalb der Komponenten und externen Programme ablaufen, die, sofern notwendig, mittels Systemsprachen realisiert sind.

2.2 Architektur des Baukastens

Nach den eher konzeptionellen Ausführungen am Anfang des Kapitels, soll im folgenden konkreter auf das zu realisierende System eingegangen werden. Die Vorgehensweise ist hierbei so, daß in Abschnitt 2.2.1 eine horizontale Betrachtung der Architektur erfolgen soll, d. h. die Aufteilung in einzelne Komponenten, das Zusammenspiel zwischen diesen und Verteilungsaspekte beschrieben werden sollen. Die Verteilung der Komponenten über Rechnergrenzen hinweg führt dann anschließend zu einigen Betrachtungen über Kommunikationsmechanismen in verteilten Systemen.

[7] siehe Glossar Seite 185

Eine sich anschließende vertikale Betrachtung der Komponenten in Abschnitt 2.2.2 liefert das Schichtenmodell von $\mathcal{W}*FLOW$. Hierbei werden verschiedene Realisierungsentscheidungen, z. B. über das eingesetzte Datenbanksystem, konkrete Sprachentscheidungen, Schnittstellen und Kommunikationsprotokolle dargelegt.

2.2.1 Aufteilung und Zusammenspiel

Da die im vorherigen Abschnitt vorgestellten Konzepte zur Ablaufsteuerung, zum Datenmanagement und zur Programmintegration in der Informatik unabhängig voneinander existieren, soll versucht werden, diese Konzepte auf möglichst autarke Teilkomponenten zu verteilen und das Zusammenspiel der Komponenten über klar definierte Schnittstellen zu realisieren. Dies hat den Vorteil, daß bei Erweiterungen am System Komponenten mit fest umrissenen Aufgabenbereichen und Kommunikationsschnittstellen vorhanden sind. Die Vergangenheit hat gezeigt, daß große monolithische Gesamtsysteme, bei denen die Interaktion zwischen den funktionalen Teilbereichen nicht klar definiert und dokumentiert sind, nur schwer zu erweitern sind [GHJV96].

Im folgenden werden nun eine Reihe von Komponenten definiert, aus denen das System aufgebaut werden soll. Die Komponenten sind im einzelnen:

$\mathcal{W}*FLOW$–Container: Diese werden zur Verwaltung der Applikationsdaten eingesetzt. Sie erlauben die Modellierung von komplexen Datenbeziehungen, wie Versionsverwaltung, Mengen, Referenzen, Konfigurationen und stellen datenbankspezifische Dienste wie Transaktionsmechanismen, Suchanfragen etc. bereit. Eine genaue Beschreibung der Leistungsmerkmale findet sich in Abschnitt 3.1.

$\mathcal{W}*FLOW$–Engine: Sie nutzt die von den anderen Komponenten realisierte Funktionalität zur Datenhaltung und zum Starten von Applikationen und verwaltet die komplexen Beziehungen zwischen ablaufenden Aktivitäten, ihren Ein- und Ausgabedaten und den die Tätigkeiten durchführenden Benutzern. Eine genaue Beschreibung der Leistungsmerkmale findet sich im Abschnitt 3.2.

$\mathcal{W}*FLOW$–Toolservices: Diese Komponenten dienen zum Starten von Applikationen in verteilten, heterogenen Umgebungen. Kern der Dienste sind Abbildungsvorschriften, die abstrakte Tätigkeitsbeschreibungen konkrete Programme oder Skripte, in Abhängigkeit von bestimmten Laufzeitbedingungen, zuordnen. Die Abbildungsvorschriften werden in einer Tooldatenbank, dem *Toolserver*, verwaltet. Der Start der eigentlichen Applikationen wird von der *Toolstarter* Komponente durchgeführt. Eine genaue Beschreibung der Leistungsmerkmale findet sich in Kapitel 4.

Die einzelnen Komponenten besitzen jeweils eine API in der gewählten Skriptsprache, über die sie angesprochen und ihre Dienste genutzt werden können.

Abbildung 2.1 zeigt die Kommunikation und Aufrufhierarchie. Wie oben beschrieben wird hier deutlich, daß die $\mathcal{W}*FLOW$–Engine, die von den anderen Komponenten zur Verfügung gestellte Funktionalität nutzt (dicke Pfeile), während die beiden anderen Komponenten unabhängig von der $\mathcal{W}*FLOW$–Engine realisiert sind. Dies bedeutet insbesondere, daß *Toolservices*[8] und *Container* völlig eigenständig ohne $\mathcal{W}*FLOW$–Engine für Applikationen zu Datenmanagement und Toolintegrationszwecken nutzbar sind.

[8] Der Einsatz der *Toolservice* Komponenten in einer verteilten Simulationsumgebung ist in [UNJ+99] beschrieben.

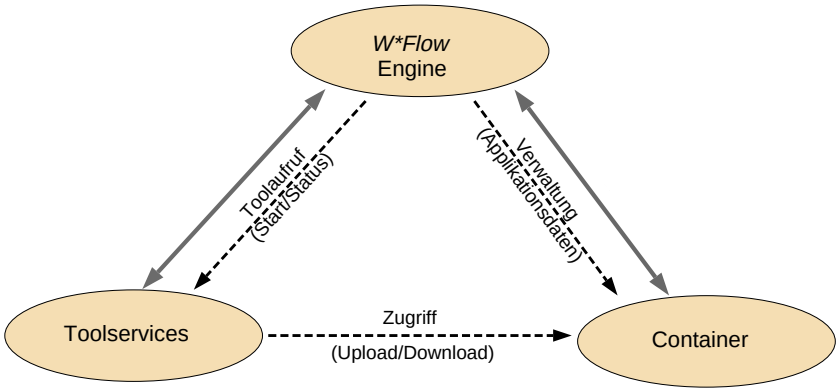


Abbildung 2.1: *W∗FLOW: Zusammenspiel der Komponenten*

2.2.1.1 Verteilung

Ein Hauptkritikpunkt an vielen kommerziellen Workflowsystemen ist die mangelnde Skalierbarkeit. Dies ist ein Hauptgrund für Performance–Probleme bei steigender Last sowie für mangelnde Verfügbarkeit und verhindert somit den Einsatz von Workflowsystemen in größerem Rahmen [AAAM97]. Grund für die Beschränkung ist zumeist ein zentrales Datenbanksystem, bei dessen Versagen das ganze System ausfällt. Um die Beschränkung zu umgehen, ist *W∗FLOW* so ausgelegt, daß die einzelnen Komponenten im Netz verteilt werden können und weiterhin einzelne Komponenten parallel auf mehreren Rechnern eingesetzt werden können, was eine hohes Maß an Skalierbarkeit und Verfügbarkeit, infolge von Partitionierung und/oder Replikation, erlaubt.

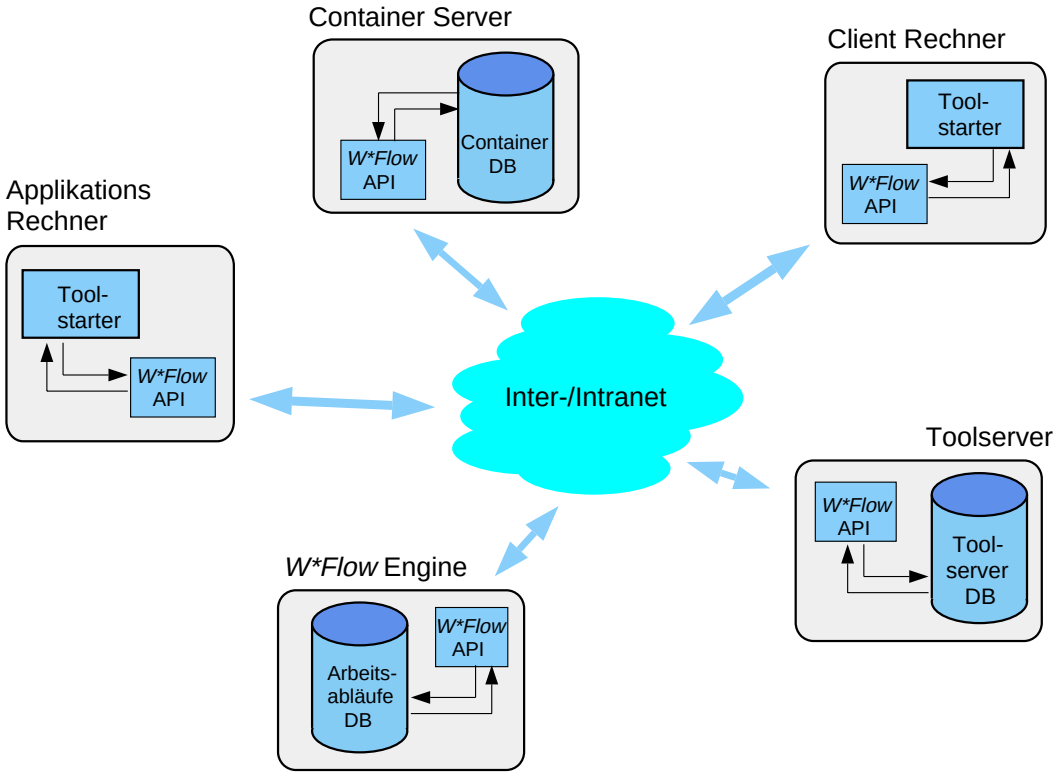


Abbildung 2.2: *Komponenten des Systems*

Abbildung 2.2 zeigt eine Verteilung, bei der jede Komponente auf einem separaten Rechner läuft. Die Abbildung zeigt die drei Hauptkomponenten *Container*, *W*FLOW* Engine und *Toolserver*. Zusätzlich laufen auf den Applikations– bzw. Client Rechnern noch sogenannte *Toolstarter* Prozesse, welche im Zusammenspiel mit dem *Toolserver* für den eigentlichen Start einer externen Anwendung verantwortlich sind. Neben der vollständigen Verteilung der Komponenten ist es jedoch auch möglich mehrere Komponenten auf einem Rechner ablaufen zu lassen, bzw. einzelne Komponenten, wie z. B. die *Containerkomponente* mehrfach einzusetzen.

2.2.1.2 Kommunikation

Durch die Möglichkeit, Komponenten auf unterschiedlichen Rechnern zu verteilen, werden zusätzliche Anforderungen an die bereitzustellende Baukasten-Sprache gestellt. Die von den Komponenten bereitgestelle API muß nun derart konzipiert sein, daß eine Kommunikation über Rechnergrenzen hinweg erfolgen kann.

Ein in Frage kommender Kommunikationsmechanismus muß hierbei, entsprechend der Anforderung an das Gesamtsystem, insbesondere die Forderung nach Plattformunabängigkeit erfüllen. Diese Forderung wird durch das TCP/IP [Ste98b] Protokoll erfüllt, das sich als Standardprotokoll im Internet etabliert hat und so gut wie in jedem Betriebssystem vorhanden ist. TCP/IP benutzt sogenannte *Sockets* als Kommunikationsendpunkte, über die eine gesicherte verbindungsorientierte Kommunikation möglich ist.

Bei den übertragenen Daten handelt es sich um einen uninterpretierten Datenstrom, der den Nachteil hat, daß die Bytes bei der Kommunikation zwischen Rechnern unterschiedlicher Architektur eine andere Bedeutung haben können, d. h. die Darstellung der Zeichen– und Zahlenformate auf den jeweiligen Rechnern unterschiedlich ist.

Aus diesem Grund hat sich oberhalb von TCP/IP eine weitere Protokollschicht etabliert, die sich mit der semantischen Repräsentation der übertragenen Daten beschäftigt. Diese Schicht wird als Darstellungsschicht bezeichnet. Beispiele sind etwa ASN.1 [CCI88] (Abstract Syntax Notation), das im Rahmen des ISO/OSI [Tan88] Protokolls entwickelt wurde oder XDR [Sun87] (External Data Representation), das im Rahmen von Suns RPC[9] Implementierung eingesetzt wird.

Im Rahmen von *W*FLOW* erfolgt die Kommunikation durch den Aufruf von Objektmethoden. Hierbei handelt es sich um eine Form des Client–Server Prinzip [FSH+97, Sch97], das den Aufruf von Funktionen und Methoden auf entfernten Rechnern erlaubt. Diese Art der Kommunikation wird z. B. durch das RPC Protokoll[10] [Blo92], dem RMI[11] [Dow98] Mechanismus von SUN oder durch CORBA [OMG95a, Sie96, Nat95] Implementierungen zur Verfügung gestellt. Die dazu notwendigen Mechanismen werden oberhalb der Darstellungsschicht realisiert.

Bei der Kommunikation mittels RMI bzw. CORBA werden im Gegensatz zu RPC Methodenaufrufe auf entfernten Objekten ausgeführt, während RPC nur den Aufruf von entfernten Prozeduren bzw. Funktionen erlaubt[12]. RMI ist ein Protokoll, das von SUN zur Kommunikation innerhalb von Java eingesetzt wird, während CORBA von der Object Management Group als Referenzarchitektur für verteilte Anwendungen entwickelt wurde, die im Gegensatz zu den anderen Ansätzen sprach– und herstellerunabhängig ist[13].

[9] **R**emote **P**rocedure **C**all (siehe Glossar Seite 183)

[10] Hiervon gibt es eine Reihe verschiedener Implementierungen, die aber jeweils zueinander inkompatibel sind.

[11] **R**emote **M**ethod **I**nvocation (siehe Glossar Seite 183)

[12] Die Kommunikation mittels RMI bzw. CORBA stellt somit eine objektorientierte Variante der Kommunikation mittels RPC dar.

[13] Während es sich bei RMI und RPC primär um Kommunikationsprotokolle handelt, bietet CORBA eine komplette Infrastruktur mit einer Vielzahl verschiedener Dienste (*Facilities*), wie auch in Abschnitt 1.1.1 über die

Ohne eine spezielle Realisierung der Kommunikationsdienste an dieser Stelle auszuwählen, wird die Funktionalität der Komponenten API dahingehend erweitert, daß ein netzwerktransparenter Kommunikationsmechanismus, der den Aufruf von Methoden und Funktionen über Rechnergrenzen hinweg unterstützt, vorhanden sein muß. Auf die Auswahl einer geeigneten Realisierung wird in Abschnitt 2.2.2.3 genauer eingegangen.

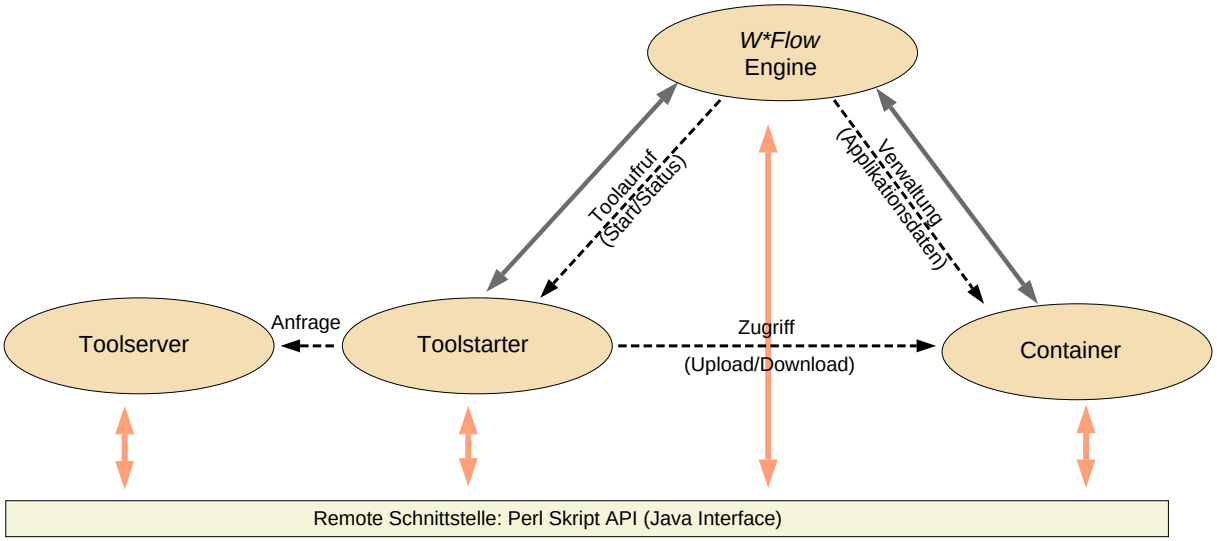


Abbildung 2.3: *W*FLOW: Zusammenspiel der Komponenten*

In Abbildung 2.3 sind das Zusammenspiel der Komponenten sowie ihre Schnittstellen noch einmal graphisch zusammengefaßt.

2.2.2 Realisierungsaspekte

Nachdem im vorherigen Abschnitt über den Aufbau und das Zusammenspiel der *W*FLOW* Komponenten gesprochen wurde, enthält dieser Abschnitt konkrete Realisierungsaspekte, die gewählt wurden, um obige Konzepte umzusetzen.

2.2.2.1 Schnittstellen API

Die Entscheidung, welche konkrete Skriptsprache zur Schnittstellenrealisierung eingesetzt werden soll, fiel auf *Perl* [Sch93, WCS96], da hier eine Reihe von Vorzügen[14] vorhanden sind, die im folgenden kurz angeführt werden sollen:

- Die Sprache ist ein Hybrid aus interpretierter und compilierter Sprache. Bei jedem Start wird das Programm in eine interne, optimierte p–Code Darstellung übersetzt, die dann anschließend ausgeführt wird. Dies macht *Perl*, im Vergleich zu den meisten anderen Skriptsprachen sehr schnell, allerdings mit dem Nachteil einer kurzen Übersetzungsphase zum Startzeitpunkt.
- Ab Version 5 kann in *Perl* objektorientiert programmiert werden. Die objektorientierten Mechanismen in *Perl* unterstützen dabei die komponentenbasierte Programmierung.

Object Management Group nachgelesen werden kann.

[14] Dies heißt nicht, daß die aufgeführten Features nicht auch bei anderen Sprachen vorhanden sein können. Die Entscheidung für *Perl* fiel aufgrund der Fülle und Kombination einzelner Merkmale.

- Es stehen eine Reihe von Oberflächentoolkits zur Verfügung. Unter anderem ist eine Portierung der Tk-Bibliothek von *tcl* [Ous94] verfügbar, die den einfachen Aufbau von graphischen Benutzerschnittstellen (GUI[15]) erlaubt.

- *Perl* besitzt eine Schnittstelle zu C [Roe96, Sri97, Oka98], mittels der es möglich ist, sowohl *Perl*–Code in C Programme zu integrieren, als auch einzelne Methoden und Funktionen in C zu entwickeln[16] und von *Perl*skripten aus aufzurufen. Dies erlaubt es z. B. in C oder C++ entwickelte Libraries von *Perl* aus in einfachster Weise zu nutzen.

- Für *Perl* steht eine große Library an freier Software zur Verfügung, die auf den CPAN[17] Servern zu finden ist. Momentan finden sich dort Bibliotheken, die in 22 Kategorien, wie z. B. Development_Support, Networking_Device_Inter_Process, Database_Utilities, World_Wide_Web, User_Interfaces, String_Processing, etc., untergliedert sind [CPA99].

- Insbesondere sind eine Vielzahl von Modulen zur Systemintegration und –verwaltung vorhanden. Dies ermöglicht es mit *Perl*, im Gegensatz zu vielen anderen Skriptsprachen, auch eine sehr systemnahe Programmierung durchzuführen, was speziell für den Aspekt der Integration von vorhandenen Bausteinen wichtig ist.

- *Perl* eignet sich hervorragend zur Entwicklung von verteilter Anwendungen. Neben den bereits innnerhalb der Sprache realisierten Möglichkeiten finden sich auf den CPAN Servern eine Vielzahl von Modulen zur Interprozeßkommunikation [CPA00]. So gibt es beispielsweise Module zur Kommunikation basierend auf Sockets, UDP, RPC, CORBA, HTTP, SMTP, PVM, etc.[18]

- *Perl* unterstützt verschiedene Sicherheitskonzepte, die es beispielsweise erlauben, nur bestimmte Anweisungen durchzuführen. Die Einschränkung erfolgt auf Ebene der OP-Codes, so daß es beispielsweise möglich ist, fremden *Perl*–Code innerhalb der $\mathcal{W}$*FLOW*–Engine ausführen zu lassen, der dann, vergleichbar den *Java–Applets* nur bestimmte Funktionen ausführen darf [SSP99].

- *Perl* läuft unter einer Vielzahl von Betriebssystemen, wie z. B. fast allen *UNIX* und *Windows* Plattformen, *VMS*, *OS2*, *BeOS*, *AS400*, *MVS*, *Macintosh*, um nur die wichtigsten zu nennen.

- Es existiert ein Interface zu Java (*JPL* [Jep97]), das es erlaubt, sowohl von *Perl* aus Java Methoden aufzurufen, als auch umgekehrt *Perl* Methoden in Java Code einzubinden. In Verbindung mit der Remote Method Invocation (RMI) von Java ist es möglich, die komplette $\mathcal{W}$*FLOW*–API auch als Remote Schnittstelle in Java anzubieten, was bei der Entwicklung von Benutzerinterfaces in heterogenen Umgebungen eine wichtige Rolle spielen kann.

- Es lag bereits Erfahrung mit der Programmierung von *Perl* vor.

Speziell durch die Möglichkeit relativ einfach C/C++ Code einzubinden, als auch umgekehrt *Perl*code in C/C++ Programme zu integrieren, kann *Perl* sowohl als Laufzeitschnittstelle des

[15] **G**raphical **U**ser **I**nterface
[16] z. B. wenn es auf die Performance ankommt.
[17] **C**omprehensive **P**erl **A**rchive **N**etwork
[18] Eine Erläuterung der Abkürzungen und Begriffe findet sich im Glossar der Arbeit ab Seite 179.

Systems zu einer Skriptsprache genutzt werden, als auch als Entwicklungssprache für die zu realisierenden Workflowsysteme eingesetzt werden. Durch bereits entwickelte Datenbankschnittstellen für unterschiedliche Datenbanksysteme, welche auf den CPAN–Servern zur Verfügung stehen, ist ein existierender Mechanismus zur Persistenzsicherung vorhanden.

2.2.2.2 Datenbanksystem

Der $\mathcal{W}*FLOW$–Baukasten benötigt ein Persistenzkonzept aus zwei verschiedenen Gründen. Zum einen muß das System seine eigenen Daten über die zu bearbeitenden Workflows (z. B. Status der Abarbeitung, zugeordnete Bearbeiter, ...) halten, und zum anderen ist $\mathcal{W}*FLOW$ dahingehend konzipiert, daß es auch für die Applikationsdaten ein Speichersystem zur Verfügung stellt, das es erlaubt, die Daten mitsamt weiteren Informationen und Beziehungen zu verwalten.

In einem ersten Prototyp der Engine wurde die relationale Datenbank *msql* von Huges Technology [Hug96] eingesetzt, die frei erhältlich ist und für die eine Schnittstelle zu *Perl* auf den CPAN–Servern verfügbar ist.

Dadurch, daß $\mathcal{W}*FLOW$ streng objektorientiert entwickelt wurde, mußte bei jedem Abspeichervorgang das komplette Objekt auf die einzelnen Tabellen der Datenbank abgebildet werden und umgekehrt bei jedem Lesezugriff das Objekt durch Join–Operatoren aus mehreren Tabellen aufgebaut werden, was zum einen aus Performancegründen nicht optimal ist und zum anderen die Entwicklung des Baukastens durch die teilweise notwendigen Schemaänderungen innerhalb der Datenbank nicht unerheblich verlangsamt. Weiterhin existiert für *msql* kein Transaktionsmechanismus, so daß das ACID[19] Prinzip durch strenge Serialisierung und zusätzliche Log-Daten für UNDO Operationen nachgebildet werden mußte.

Es zeigte sich somit schnell, daß die gewählte Datenbank den Anforderungen, die an ein solches System, auch in bezug auf Verteilung und Multiuser–Betrieb gestellt werden, nicht gewachsen war. Auf der Suche nach einer Alternative für *msql* wurde neben relationalen Datenbanksystemen wie z. B. *ORACLE* [KL95] auch objektorientierte Systeme gesichtet. Die Wahl fiel schließlich auf das objektorientierte Datenbanksystem *ObjectStore* der Firma Object Design. Die Gründe für die Entscheidung *ObjectStore* einzusetzen waren folgende:

- Durch das rein objektorientierte Design von $\mathcal{W}*FLOW$ bietet es sich an, die Daten auch als Objekte abzuspeichern, da hierbei die aufwendige (De–)Komposition der Objektstrukturen unterbleiben kann.
- *ObjectStore* besitzt ein patentiertes Verfahren, das es erlaubt, persistente Datenstrukturen in den Speicherbereich der Client–Programme abzubilden (Cache Forward Architecture). Sind die benötigten Daten einmal in den Cache Bereich der Client Anwendung geladen, so erfolgt der Zugriff auf persistente Objekte genauso schnell wie auf transiente Objekte.
- Die Funktionalität von *ObjectStore* umfaßt neben der persistenten Speicherung von Objekten auch Datenbankdienste wie Transaktionen, Suchanfragen, Indizierung von Datensätzen etc.
- Object Design ist mit ca. 33% Marktanteil der Marktführer im Bereich objektorientierter Datenbanken (Quelle: [Rog97]) und ist als Datenbankkomponente in einer Reihe von großen kommerziellen Systemen eingesetzt.

[19] Unter dem ACID Prinzip wird ein Konzept zur Konsistenzsicherung in Datenbanken verstanden. Es basiert auf den Prinzipien Atomiziät (**A**tomicity), Synchronisation konkurrierender Zugriffe (**C**oncurrency), Unsichtbarkeit temporärer Änderungen vor anderen Benutzern (**I**solation) und Dauerhaftigkeit der Ergebnisse (**D**urability). Eine ausführlichere Beschreibung findet sich in Abschnitt 3.2.2.

- Neben einer Sprachanbindung für C++ besitzt *ObjectStore* noch Sprachanbindungen für *Java* und *ActiveX*, die eventuell bei der Realisierung von GUI Komponenten auf Datenstrukturebene von Hilfe sein könnten. Gatewayfunktionalität zu relationalen Datenbanken ist ebenfalls vorhanden.

Nach Erfahrungen mit der *msql* Schnittstelle für *Perl*, wurde eine Schnittstelle zwischen *Perl* und der *ObjectStore* Sprachanbindung für C++ entwickelt. Die Anbindung umfaßte die elementaren Operationen, Transaktions– und Anfragedienste.

Nachdem im Juni '97 von Joshua Pritikin in der comp.lang.perl.announce Newsgroup ein Modul vorgestellt wurde, das ebenfalls eine Schnittstelle zwischen der *ObjectStore* Datenbank und *Perl* realisierte, wurde $\mathcal{W}$*FLOW* Ende '97 ebenfalls auf dieses Modul portiert und die Entwicklung des eigenen Moduls eingestellt.

Das Modul liegt inzwischen in der Version 1.53[20] vor und ist als sehr stabil zu bezeichnen. Es wird vom Autor ständig weiterentwickelt und es existiert eine Mailingliste[21], in der Probleme und Erfahrungen mit dem Modul diskutiert werden.

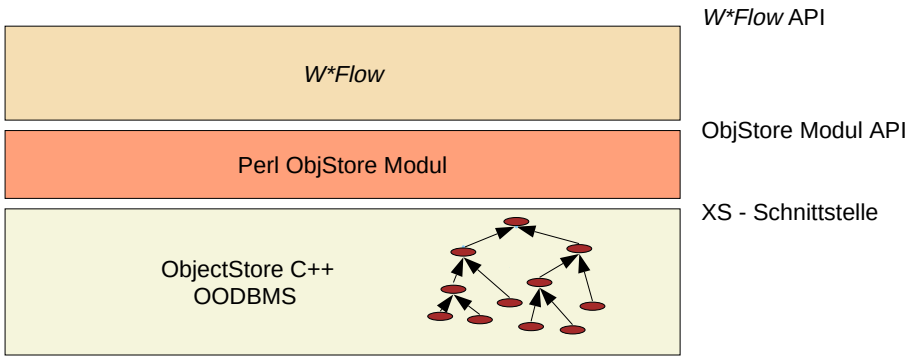


Abbildung 2.4: *Schichtenmodell von $\mathcal{W}$*FLOW*

Die Architektur, welche sich somit für die $\mathcal{W}$*FLOW*–Komponenten präsentiert, ist in Abbildung 2.4 dargestellt. Als unterste Schicht ist die *ObjectStore* Datenbank (OODBMS) mit der C++ Sprachanbindung dargestellt. Die Schnittstelle zu dem *Perl ObjStore* Modul wird durch die XS–Schnittstelle realisiert, welche hauptsächlich eine Abbildung der C/C++ Datentypen auf die *Perl*datentypen vornimmt. Das *Perl ObjStore Modul* stellt die Basisfunktionalität für den Zugriff von *Perl* auf die *ObjectStore* Datenbank zur Verfügung. Die Basisfunktionalität wird durch die *ObjStore* API, welche in [Pri99] beschrieben wird, bereitgestellt. Die API wird vom eigentlichen $\mathcal{W}$*FLOW*–Kernel zur Realisierung der Basisfunktionalität des Baukastens genutzt, die dann wiederum im Rahmen einer API dem Workflowsystem-Entwickler zur Verfügung gestellt wird.

2.2.2.3 Kommunikationsprotokoll

Bei der Betrachtung der Kommunikationsprotokolle aus Abschnitt 2.2.1.2 unter dem Gesichtspunkt des Einsatzes im Rahmen von $\mathcal{W}$*FLOW* bleiben aufgrund der objektorientierten Realisierung der $\mathcal{W}$*FLOW* Komponenten nur RMI und CORBA übrig. Für CORBA spricht, daß für C++ eine Sprachanbindung vorliegt, was eine Anbindung an *Perl* ($\mathcal{W}$*FLOW*–Frameworksprache) möglich macht. Aber auch der Java–RMI Mechanismus kann durch das *JPL* Interface (siehe Abschnitt 2.2.2.1) angebunden werden.

[20] Stand Februar '99
[21] osperl@listbox.com

Wegen der Sprachunabhängigkeit und der Bedeutung von CORBA im Bereich verteilter, heterogener Anwendungen ist CORBA im Rahmen von $\mathcal{W}*FLOW$ als das interessantere und auch flexiblere Kommunikationsprotokoll anzusehen. Zum momentanen Zeitpunkt gibt es eine Reihe von Anstrengungen, eine Sprachanbindung von *Perl* an den CORBA Standard zu entwickeln (siehe [Sch98b]). Diese sind aber noch in einem sehr frühen Stadium und deshalb für die Einbindung in $\mathcal{W}*FLOW$ zu diesem Zeitpunkt nicht geeignet.

Es gab jedoch eine andere Alternative, mit deren Hilfe der Aufwand einer eigenen Realisierung für eine rudimentäre Sprachanbindung von Perl an CORBA umgangen werden konnte. Auf den *Perl* CPAN–Servern findet sich in der Kategorie Networking_Devices_Inter_Process ein RPC Modul von Jochen Wiedmann [Wie98], das neben "Standard" RPC Mechanismen auch in der Lage ist, entfernte Methoden aufzurufen, also eine Art *Remote Method Invocation* im Sinne von CORBA zu realisieren. Die Vorteile dieses Paketes liegen in seiner geringen Größe und dem direkten Aufsetzen auf TCP/IP, ohne etwaige zueinander inkompatible RPC Protokolle wie *ONC RPC*, *DCE RPC* oder *DFN RPC*[22] zu benötigen. Durch das direkte Aufsetzen auf TCP/IP wird eine hohe Plattformunabhängigkeit erreicht, und das Modul ist auf allen Plattformen einsetzbar, auf denen auch *Perl* verfügbar ist. Ein weiterer Vorteil liegt darin, daß das Modul bereits in *Perl* vorliegt, so daß die Realisierung einer Anbindung an C++ oder Java unterbleiben kann. Durch den an RMI angelehnten Kommunikationsmechanismus ist zu einem späteren Zeitpunkt ein Übergang zum mächtigeren CORBA nicht ausgeschlosssen.

Der Leistungsumfang dieser "objektorientierten RPC" Implementierung bleibt zwar hinter der von CORBA und den kommerziellen RPC Implementierungen zurück[23], die Leistungsfähigkeit reicht für eine prototypische $\mathcal{W}*FLOW$ Implementierung jedoch voll aus, da beispielsweise Performance– und Sicherheitsaspekte, wie sie bei kommerziellen Systemen vorhanden sind, hier keine große Rolle spielen und einzelne nicht vorhandene Dienste, sofern benötigt, rudimentär implementiert werden können.

Ein Argument, das gegen eine CORBA basierte Lösungen spricht, ist, daß das TCP/IP Protokoll weitaus verbreiteter ist, als dies momentan noch bei CORBA der Fall ist. Der Einsatz von CORBA setzt auf den beteiligten Rechnern oftmals die Installation von kommerzieller Software voraus. Im Falle, daß auf einem Rechner keine CORBA Implementierung verfügbar ist bzw. nicht installiert ist, kann dieser Rechner nicht genutzt werden, was speziell bei den Client–Rechnern zu einer Einschränkung beim Einsatz als Applikationsrechner führt. Der Einsatz eines "leichtgewichtigen" Protokolls, wie es das oben beschriebene *Perl* basierte RPC darstellt, das so gut wie auf jedem Rechner ablauffähig und einfach zu installieren ist, stellt hier einen Vorteil dar.

Abbildung 2.5 zeigt die drei oben besprochenen Kommunikationsvarianten auf Basis des Schichtenmodells aus Abbildung 2.4. Variante (a) ist auf Basis des Java RMI Mechanismus realisiert, der mittels dem JPL Interface an *Perl* angebunden wurde. Variante (b) nutzt die *Perl* XS Schnittstelle um über CORBA zu kommunizieren und Variante (c) zeigt die im Rahmen dieser Arbeit eingesetzte Realisierung, bei der ein in *Perl* entwickeltes, objektorientiertes RPC Modul zur Kommunikation genutzt wird.

2.3 Basisfunktionalität der Komponenten

Bevor in den nachfolgenden Kapiteln die einzelnen Komponenten einzeln im Detail besprochen werden, soll in diesem Abschnitt die Basisfunktionalität aller Komponenten, d. h. diejenigen

[22] Ein Vergleich dieser drei Implementierungen findet sich in [RS93].

[23] So gibt es keinerlei Mechanismen zur Ortstransparenz und es sind auch keinerlei zusätzliche Dienste wie Naming–, Property–, Event Service, etc. realisiert.

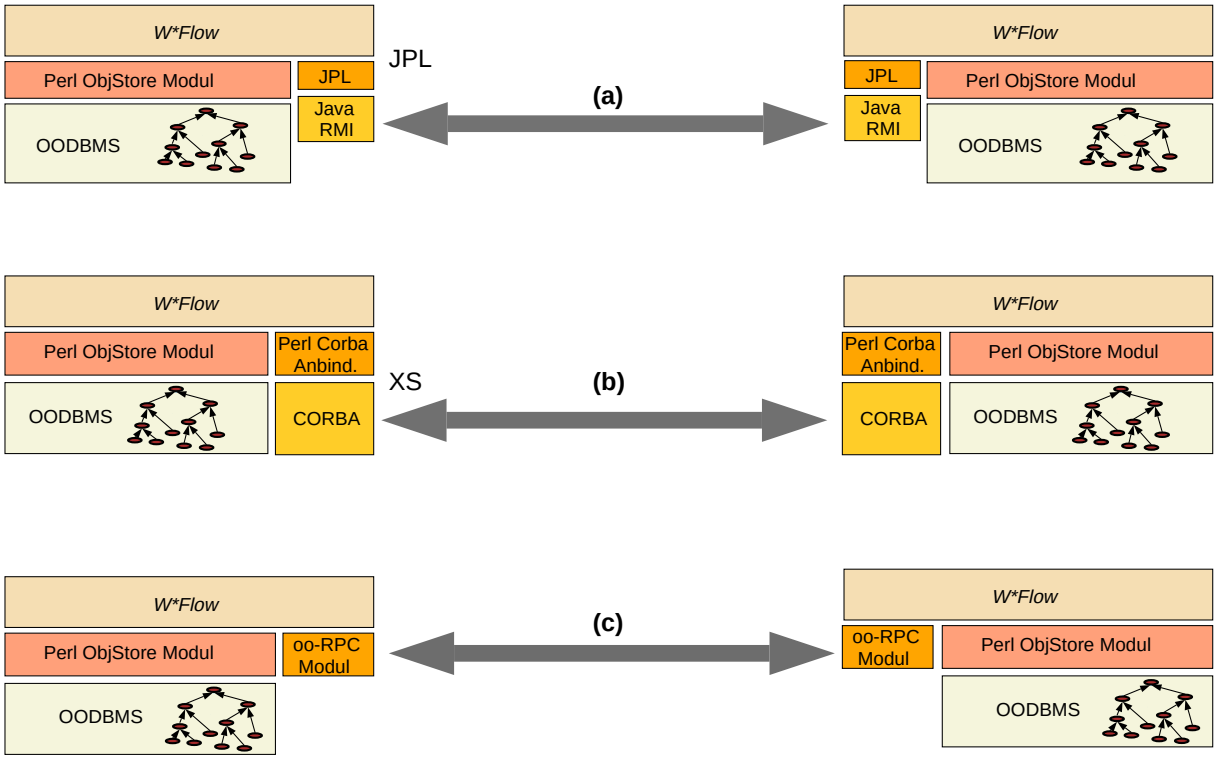


Abbildung 2.5: *Kommunikationsvarianten in $\mathcal{W}$*FLOW*

Eigenschaften und Leistungsmerkmale, die allen Komponenten gemein sind, vorgestellt werden. Durch die objektorientierte Realisierung von $\mathcal{W}$*FLOW kann die Funktionalität am besten in einer oder mehreren Oberklassen der anschließend vorgestellten Komponentenklassen untergebracht werden. Abbildung 2.6 zeigt die Basisklasse Wildflow::Object, von der die anderen Klassen abgeleitet werden und somit die Funktionalität der Oberklasse erben.
Die Basisfunktionalitäten können in drei Bereiche gegliedert werden: Attribute, Kommunikations– und Persistenzmechanismen.

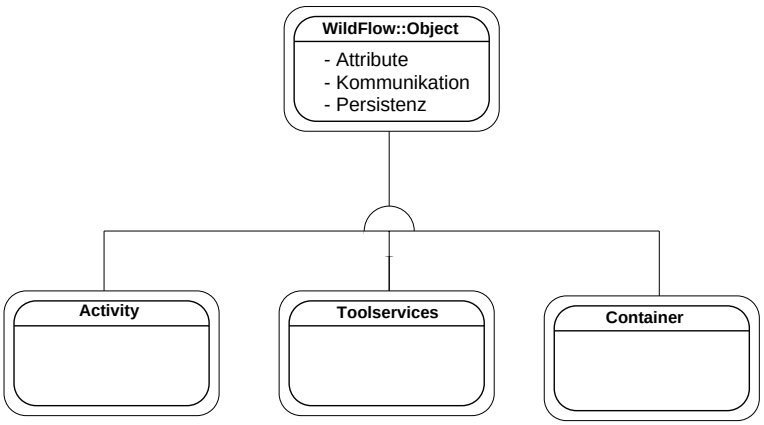


Abbildung 2.6: *Bereitstellung der Basisfunktionalität in einer gemeinsamen Oberklasse*

2.3.1 Attribute

Jede Komponente besitzt die Möglichkeit, eine beliebige Anzahl von Attributen frei zu definieren. Den Attributen können Werte zugewiesen werden. Im Gegensatz zu den *Property Services* der OMG [OMG96] sind die $\mathcal{W}*FLOW$–Attribute jedoch untypisiert und erlauben die Aufnahme beliebiger Datentypen. Mittels Attributen ist es möglich, eine Komponente mit den jeweils notwendigen Informationen zu versehen und sie so an spezielle Gegebenheiten anzupassen. Neben skalaren Werten ist es auch möglich, Referenzen auf andere $\mathcal{W}*FLOW$ Objekte zu speichern. Der Zugriff auf die Attributwerte erfolgt durch vom System bereitgestellte Methoden. Hierdurch wird eine vollständige Kapselung der Komponenten erreicht.

Weiterhin besteht noch die Möglichkeit, spezielle, sogenannte *berechenbare Attribute* zu definieren. Diese Attribute speichern keinen Attributwert, sondern enthalten eine Ausführungsanweisung mittels der ein Wert berechnet bzw. extrahiert werden kann. Im Gegensatz zu normalen Attributen können diese natürlich nur ausgelesen werden. *Berechenbare Attribute* sind somit mit Methoden vergleichbar, können aber dynamisch zur Laufzeit einer Komponente zugeordnet werden. Der Grund, warum hier von einem berechenbaren Attribut und nicht allgemein einfach von einer Methode gesprochen wird ist der, daß es sich bei der Berechnungsvorschrift, die mittels der $\mathcal{W}*FLOW$–API formuliert werden kann, um den Wert des Attributes handelt, d. h. beim lesenden Zugriff auf das Attribut wird die Berechnungsvorschrift ausgeführt, während das Schreiben des Attributes dem Speichern einer neuen Berechnungsvorschrift entspricht.

2.3.2 Kommunikationsmechanismen

Neben der Kommunikation durch Nachrichten mittels API Aufrufen, welche einen synchronen Mechanismus zur Kommunikation darstellen, unterstützen die Komponenten von $\mathcal{W}*FLOW$ noch einen Mechanismus, der auch den Austausch von asynchronen Nachrichten erlaubt.

In diesem Abschnitt soll ein Mechanismus vorstellt werden, der es erlaubt, auf flexible Art und Weise auf externe Ereignisse zu reagieren, ohne daß die Kapselung der dabei beteiligten Komponenten verletzt wird.

Der Mechanismus, welcher hierbei eingesetzt wird, wurde zum ersten Mal im *Qt*–Toolkit [FJ98, Eng96] der Firma Troll Tech eingesetzt und ersetzte dort den fehleranfälligen Einsatz von sogenannten *Callbackfunktionen*[24]. Bei *Qt* handelt es sich um ein Multi-Plattform-Toolkit zum Aufbau von graphischen Oberflächen (GUI). Es erlaubt die völlige Entkopplung von Signalquelle und –senke und somit eine komponentenorientierte Programmierung.

Die Hauptvorteile von *Qt* sind die einfache Handhabbarkeit und die damit geringe Entwicklungszeit im Vergleich zu anderen GUIs, die Geschwindigkeit, der voll objektorientierte Aufbau und die Plattformunabhängigkeit [LS98b]. Inzwischen existieren eine Anzahl von Public Domain– und kommerziellen Anwendungen auf diesem Toolkit [Wei98].

Im folgenden sollen die Basismechanismen, wie sie in [Tro98] beschrieben sind, vorgestellt werden:

Signal

Ein *Signal* wird von einem Objekt ausgesendet, wenn es seinen Zustand ändert und diese Zustandsänderung für andere Objekte von Interesse sein könnte. Einem Signal kann eine beliebige Anzahl von Parametern mit übergeben werden.

[24] Eine Callbackfunktion kann als eine Funktion angesehen werden, die einer anderen Funktion oder Methode als Parameter mit übergeben wird.

Das Objekt sendet das Signal ohne Kenntnis darüber, ob es von irgend jemand empfangen wird. Dadurch, daß keinerlei Informationen über den oder die Empfänger eines Signals im Senderobjekt vorliegt, ist eine komponentenorientierte Programmierung möglich.

Slot

Ein Slot kann als Empfänger eines Signals angesehen werden. Ein Slot ist gleichzeitig eine normale Methode eines Objektes. Er kann mit einer beliebigen Anzahl von Signalen verbunden sein. Ein Slot weiß nicht, ob irgendwelche Signale mit ihm verknüpft sind. Das impliziert, daß das Objekt keinerlei Kenntnisse über den zugrundeliegenden Kommunikationsmechanismus hat, was seinen Einsatz in der komponentenbasierten Programmierung erlaubt.

Die Verbindung zwischen Signalen und Slots geschieht durch einen orthogonalen Kommunikationsmechanismus, z. B. durch einen Ereigniskanal, wie er in [OMG95c] beschrieben wird oder dem im *Java* Komponentenframework existierenden InfoBus [SUN99]. Diese Mechanismen erlauben eine n:m Verbindung zwischen Signal und Slot. Abbildung 2.7 macht diesen Sachverhalt deutlich. So wird das Signal 4 des Objektes 1 von Objekt 4/Slot 1, Objekt 5/Slot 2 und Objekt 6/Slot 5 empfangen, während Slot 2 von Objekt 5 gleichzeitig mit Objekt 1/Signal 4 und Objekt 6/Signal 7 verbunden ist.

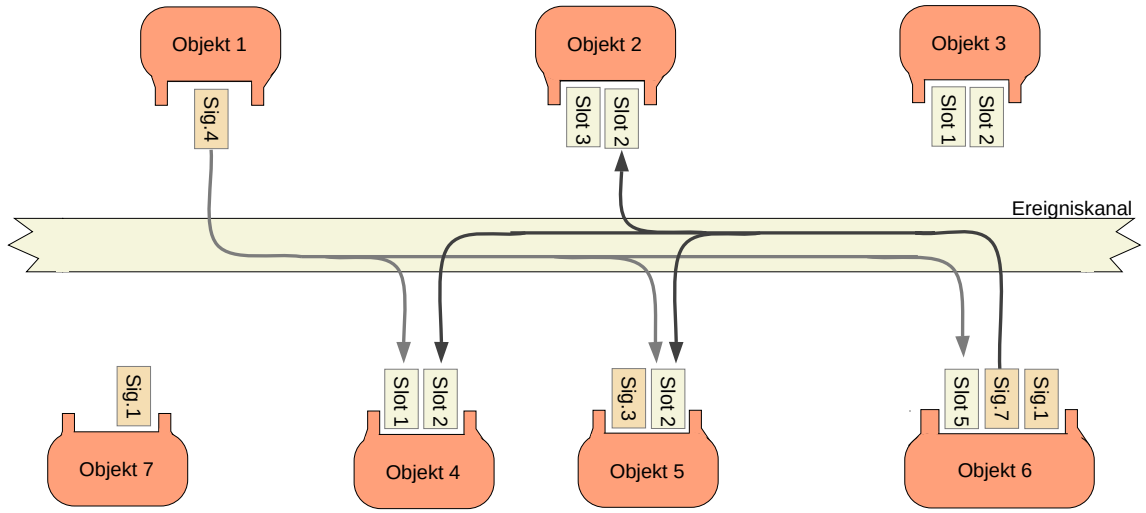


Abbildung 2.7: *n : m Kommunikation auf einem Ereigniskanal*

Anders ausgedrückt lassen sich Objekte als *Interessenten* von bestimmten Signalen anderer Objekte registrieren. Sendet ein Objekt ein Signal, so führt dies zur Ausführung der entsprechenden Methode im *interessierten* Objekt.

Ein solcher Mechanismus ist in allen $\mathcal{W}$*FLOW* Komponenten ebenfalls enthalten und wird von den einzelnen Komponenten in unterschiedlicher Weise genutzt, etwa von der Aktivitätenkomponente in Abschnitt 3.2, um auf externe Ereignisse reagieren zu können.

Das Codefragment in Beispiel 2.3.1 zeigt den zugrundeliegenden Basismechanismus.

```
#
# $activity1, $activity2 sind an dieser Stelle bereits definiert
# my $actvity1 = ...
# my $actvity2 = ...

#
# Aktivitaet 2 soll im Falle des Scheiterns von Aktivitaet 1
# gestartet werden:
#
$activity1->connect('S_ABORTED', $activity2, start());       # (1)
```

Beispiel 2.3.1: Codebeispiel zur Funktionsweise des Signal–Slot Mechanismus

Anweisung (1) verknüpft die Aktivität 1 ($activity1) mit Aktivität 2 ($activity2) derart, daß beim Auftreten des Ereignisses (Signals) `S_ABORTED`[25] von Aktivität 1 die Methode start() von Aktivität 2 aufgerufen wird. Das Signal selbst wird durch Aktivität 1 beim Übergang in den Zustand ABORT durch die Methode emit() ausgesandt. Die Anweisung hierzu sieht etwa wie folgt aus:

```
$self->emit('S_'.$new_state, @para);
```

Dies führt dann zur Ausführung der Anweisung:

```
$activity2->start(@para);
```

Einem Signal kann eine beliebige Anzahl von Parametern (Array @para) mitgegeben werden, die an den Slot, d. h. die auszuführende Methode, weitergegeben werden.

2.3.3 Persistenz

Persistenz ist eine der Eigenschaften, die in Abschnitt 2.1.4.1 als notwendige Funktionalität einer Softwarekomponente genannt wurde. Die Persistenzeigenschaft ist das "Langzeitgedächtnis" einer Komponente. In den meisten Fällen wird der interne Zustand einer Komponente, d. h. alle internen Datenobjekte, die den Zustand einer Komponente manifestieren, serialisiert[26] und in einer Datenbank oder dem Dateisystem abgelegt. Indem die interne Repräsentation einer jeden Komponente nicht in einer zentralen Datenbank gespeichert wird, sondern es möglich ist, daß jede Komponente in einer eigenen Datenbank abgelegt werden kann, wird eine vollständige Verteilung der zu erstellenden Anwendung, ohne zentrale Datenbankkomponente und den damit verbundenen Nachteilen (siehe auch Abschnitt 1.2.1), erreicht. Hierdurch kann eine hohe Skalier– und Verfügbarkeit des Gesamtsystems erzielt werden.

2.4 Zusammenfassung

In diesem Kapitel wurde ein neues Konzept eines komponentenorientierten, verteilten Workflow-Management-Systems vorgestellt. Dazu wurden die verschiedenen Basistechnologien, die im

[25] Die Namenskonvention lautet hier derart, daß bei Annahme eines neuen Zustandes als Ereignis der Name des neuen Zustand mit dem Prefix S_ propagiert wird.

[26] d. h. die interne Hauptspeicherrepräsentation einer Struktur in eine Folge von Bytes umzuwandeln.

vorgestellten Baukasten zum Einsatz kommen, kurz beschrieben und eine Integrationsstrategie, wie diese Technologien zusammen eingesetzt werden können, entwickelt. Das Ziel, das bei der Integration der Konzepte angestrebt wurde, war die optimale Ausnutzung der vorteilhaften Eigenschaften der jeweiligen Technologie. So wird in $\mathcal{W}*FLOW$ ein Workflowkonzept zur Strukturierung der Arbeitsschritte, der Modellierung von Beziehungen zwischen diesen und der Koordination von Arbeitsschritten, Daten und Personen eingesetzt. Eine eigenständige Datenhaltungskomponente zur Speicherung und Verwaltung von Anwendungsdaten stellt die notwendigen Datenbankdienste wie Transaktionsverwaltung, Modellierung von Beziehungen, etc. bereit. Das integrierte Applikations-Framework stellt eine flexible Schnittstelle zur Verwaltung und zum Starten von Programmen in heterogenen Umgebungen zur Verfügung.

Die Integration dieser Technologien im Rahmen eines komponentenbasierten Baukastens erfolgt durch eine netzwerktransparente Skriptsprachenschnittstelle, die den "Klebstoff" zum Aufbau von verteilten Anwendungen darstellt. Dadurch kann eine Verteilung oder Duplizierung der einzelnen Komponten im Rahmen eines heterogenen Rechnernetzes erfolgen, was insbesondere in Bezug auf Skalierbarkeit und Verfügbarkeit wichtig ist.

Nach der Einführung in die Architektur von $\mathcal{W}*FLOW$ und der Vorstellung der zu realisierenden Komponenten, werden diese nun in den folgenden Kapiteln näher besprochen.

Kapitel 3

Entwicklung der neuen Basiskomponenten

3.1 Komponente Container

3.1.1 Einführung

Die Hauptaufgabe eines Workflowsystems ist die Koordination von Programmen, Daten und Benutzern gemäß einer Ausführungsanweisung. In diesem Abschnitt sollen nun Fragen der Verwaltung, Bereitstellung, Zuordnung und Speicherung von Daten, wie sie im Rahmen einer Workflowbearbeitung anfallen, erläutert werden. Das Ziel besteht dabei darin, eine eigenständige Komponente zur Bereitstellung dieser Funktionalität zu konzipieren und anschließend zu realisieren.

Die WFMC unterscheidet drei Arten von Daten, die im folgenden kurz erläutert werden sollen. Abbildung 3.1 zeigt die Unterteilung in graphischer Darstellung.

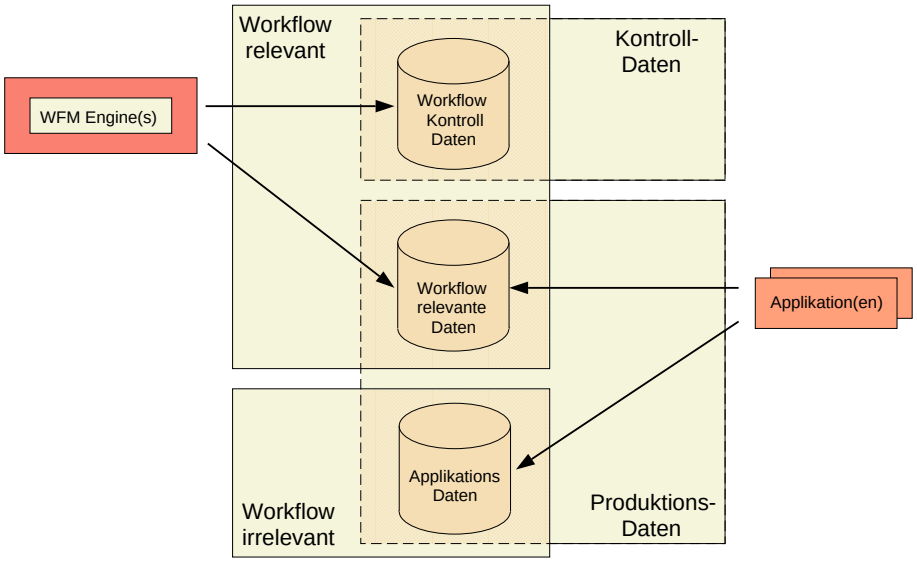


Abbildung 3.1: *Unterschiedliche Datentypen in einem Workflowsystem*

Zum einen gibt es die *Workflow-Kontroll-Daten*. Hierbei handelt es sich um workflowinterne

Daten, die zur Steuerung des Workflowsystems notwendig sind. Beispiele sind der aktuelle Zustand eines Bearbeitungsvorgangs oder der aktuelle Bearbeiter. Außerhalb des Workflowsystems haben diese Daten keine Bedeutung. Daneben gibt es noch die *Workflow-Relevanten-Daten* und die *Applikations-Daten*. Beide Arten werden von den Anwendungen und/oder Benutzern genutzt oder erzeugt. Sie werden deshalb auch als *Produktionsdaten* bezeichnet [JB96]. Applikationsdaten werden in der Regel nicht vom Workflowsystem verwaltet, sondern ihre Verwaltung obliegt der jeweiligen Anwendung, einem externen Dokumentenverwaltungs– oder Datenbanksystem. Die Unterteilung in Applikations– und Workflow-Relevante-Daten liegt in der unterschiedlichen Nutzung der Daten durch das Workflowsystem. Während Applikationsdaten für das Workflowsystem nicht von Interesse und meistens auch nicht interpretierbar sind, werden Workflow-Relevante-Daten vom Workflowsystem, beispielsweise zur Ablaufsteuerung, genutzt. Workflow Designer benötigen häufig für die Ablaufsteuerung oder andere Verwaltungszwecke im Workflow zusätzliche Informationen zu den Applikationsdaten, deren Semantik dem Workflowsystem bekannt sein muß. Diese Daten werden gerade als Workflow-Relevante-Daten bezeichnet. Der vorliegende Abschnitt befaßt sich ausschließlich mit der Verwaltung der *Produktionsdaten*, während die Workflow-Kontroll-Daten im Rahmen der $\mathcal{W}*FLOW$–Aktivitäten in Abschnitt 3.2 behandelt werden.

Zur Speicherung von Workflow-Relevanten-Daten innerhalb eines WFMS führen einige Workflowhersteller den Begriff *Container* ein. Andere verwalten diese Metadaten in sogenannten *Umlaufmappen*. Wie auch immer die Begriffsbildung hier sein mag, so enthält jedes Workflowsystem ein systemeigenes Konstrukt zur Speicherung der Workflow-Relevanten-Daten. Applikationsdaten werden innerhalb dieses Konstrukts referenziert oder gleich mit eingebunden. Hierzu zwei Beispiele:

FlowMark von IBM unterscheidet zwischen Ein– und Ausgabedaten von Aktivitäten, die in Ein– und Ausgabecontainern liegen. Bei den *FlowMark-Containern* handelt es sich um keine eigenständigen Objekte, sondern sie sind jeweils mit einer Aktivität, einem Block von Aktivitäten oder einem Prozeß verknüpft und dienen diesem als Speicher für Eingabeparameter bzw. zur Ablage von erzielten Ergebnissen. Ein Container besitzt eine aus einer geordneten Liste bestehende Struktur, die unterschiedliche Typen wie Integer– und Floating Point Zahlen, Strings oder wiederum Strukturen als Workflow-Relevante-Daten aufnehmen kann.

Als zweites Beispiel soll *ProMInanD* von der IABG[1] beschrieben werden. Im Unterschied zu *FlowMark*, bei dem Arbeitsschritte im Mittelpunkt stehen, modelliert *ProMInanD* den Datenaustausch zwischen Bearbeitungsvorgängen über migrierende Objekte in Form von Umlaufmappen. Eine Umlaufmappe (*Electronic Circular Folder – ECF*) besteht aus zwei Teilen. Der erste Teil enthält Informationen über die weitere Migration der Mappe sowie Historiendaten und den aktuellen Zustand. Der zweite Teil enthält die verschiedenen applikationsspezifischen Daten sowie Felder für informelle Nachrichten an nachfolgende Bearbeiter und allgemeine Angaben über die zu tätigenden Aufgaben.

Im folgenden wird der Begriff *Container* für die Komponente zur Speicherung der Produktionsdaten benutzt. Wie später noch genauer beschrieben wird, enthält ein $\mathcal{W}*FLOW$–Container Workflow-Relevante-Daten und gegebenenfalls auch die Applikationsdaten oder Referenzen darauf.

Im folgenden soll nun näher auf die Anforderungen und die Funktionalität eines Containers im Rahmen von $\mathcal{W}*FLOW$ eingegangen werden.

[1] **I**ndustrie**a**nlagen–**B**etriebs**g**esellschaft mbH, Ottobrunn

3.1.2 Anforderungen

Eine Reihe von Anforderungen werden aufgrund des primären Einsatzgebietes von $\mathcal{W}*FLOW$, dem wissenschaftlich–technischen Umfeld, vorgegeben. Mit diesen Anforderungen beschäftigt sich ein eigenständiger Forschungszweig (*Scientific Database Systems*). Im Rahmen eines Workshops der NSF[2] an der Universität von Virginia in Charlottesville im Jahr 1990 wurden von der NSF eine Reihe von Wissenschaftlern aus den Bereichen Informatik, Ozeanographie, Klimakunde, Geologie, Mikrobiologie, Astrophysik und Astronomie eingeladen, um die Anforderungen an Datenbanksysteme im wissenschaftlichen Bereich mit dem Ziel zu formulieren, neue Forschungaktivitäten in diesem Bereich voranzutreiben [FJP90].

Hauptthema des Forschungsgebietes *Scientific Databases* ist die Verwaltung der bei wissenschaftlichen Versuchen und Experimenten anfallenden Daten. Kernpunkte, die im Rahmen des Workshops diskutiert wurden, waren etwa die Rolle von Metadaten, das Auffinden von Daten, Benutzerschnittstellen, flexible Repräsentationsstrukturen, entsprechende Analyseoperatoren und Standardisierungsbemühungen. Zur Veranschaulichung der Problematik wird das Beispiel der Fluidiksimulation aus Abschnitt 1.2.3.1 betrachtet.

In einem der vorbereitenden Schritte zu einem Simulationslauf einer Fluidiksimulation wird aus der geometrischen Modellbeschreibung des Objektes eine Gitterzerlegung des Modells erstellt, so daß anschließend bei Durchführung der Simulation einzelne physikalische Größen der Simulation für jede Gitterzelle berechnet werden können. Die Parameter zur Zerlegung werden vom Benutzer interaktiv innerhalb eines Programms angegeben. Diese Informationen gehen zwar in die Gitterdatei mit ein, sind aber für Menschen und andere Programme nicht mehr extrahierbar, da das interne Format der Gitterdatei nicht bekannt ist. Damit die Informationen auch später noch zur Verfügung stehen, ist es notwendig, sie an gesonderter Stelle mit der erzeugten Gitterdatei abzuspeichern (Abbildung 3.2).

Abbildung 3.2: *Zusatzinformation zur Gitterdatei*

Solche Informationen werden im folgenden als *Metadaten* oder *Metainformationen* zu Applikationsdaten bezeichnet. In manchen Fällen kann es sogar vorkommen, daß die Metainformationen wichtiger sind als die eigentlichen Daten. Dies ist beispielsweise dann der Fall, wenn der Einsatz eines Workflowsystems zu Dokumentationszwecken erfolgt und im nachhinein nur noch die zugehörigen Verwaltungsinformationen und Dokumentationen, nicht aber die erfaßten Daten, von Interesse sind.
Aber auch zur Unterstützung der Suche nach Datensätzen spielen Metainformationen eine wichtige Rolle. Normale Dateisysteme erlauben für gewöhnlich die Suche nur nach dem Dateinamen, bzw. die Suche nach Textstrings, die innerhalb einer Datei vorkommen. Dies mag bei Textdateien noch eine mögliche Vorgehensweise darstellen, bei Bilddateien oder anderen binären Daten ist sie aber schon nicht mehr anwendbar. Die Unterstützung durch zusätzliche Informationen ist hier Grundvoraussetzung für ein effektives Suchen.

[2]Die **N**ational **S**cience **F**oundation ist eine unabhängige US amerikanische Behörde, deren Aufgabe die Entwicklung von Programmen zur Förderung von Wissenschaft und Technik, mit einem Gesamtrahmen von über 3.3 Milliarden US Dollar pro Jahr, ist.

Ein Ansatz zur Speicherung von Zusatzinformationen zu Dateien innerhalb eines Dateisystems wird z. B. vom *Semantic File System* (*SFS*) [GJSO91] realisiert. *SFS* erlaubt die Hinzunahme einer Reihe von Attribut–Wert Paaren zu Dateien. Über eine oberhalb der alten Dateischnittstelle aufsetzende Zugriffsschnittstelle ist ein assoziativer Zugriff auf die Dateien möglich. Daneben kann ein Benutzer Operatoren (sog. *Transducer*) definieren, die, je nach Typ der Datei, beim Eintrag eine automatische Indexierung im *SFS* durchführen.

Bestimmte, innerhalb der Applikationsdaten vorliegende, Informationen sind nicht nur für die Wissenschaftler, sondern auch für die Ablaufsteuerung des Workflows oder für nachfolgende Arbeitsschritte von Bedeutung. Hierbei handelt es sich um Daten, die durch das Workflowsystem interpretiert werden müssen. Da das Workflowsystem in der Regel solche Informationen nicht automatisch aus den Applikationsdaten, deren Format dem Workflowsystem nicht bekannt ist, extrahieren kann, ist es sinnvoll diese Informationen den Applikationsdaten als Metadaten hinzuzufügen. Solche Steuerdaten werden im Workflow-Kontext als Workflow-Relevante-Daten bezeichnet.

Metadaten und Workflow-Relevante-Daten stellen bei genauer Betrachtung lediglich zwei verschiedene Blickwinkel auf ein und denselben Sachverhalt dar. Während im ersten Fall der Anwender für **ihn** relevante Informationen zu den Applikationsdaten hinzufügt, damit er sie zu einem späteren Zeitpunkt wieder nutzen kann, oder sie zumindest ein späteres Auffinden von Datensätzen erleichtern, fügen im zweiten Fall **Workflow-Designer** den Applikationsdaten Workflow-Relevante-Datenfelder hinzu, die vom Anwender oder der Applikation auszufüllen sind, damit das Workflowsystem im späteren Verlauf auf diese Informationen zugreifen kann. Es liegt nahe, Metadaten und Workflow-Relevante-Daten durch ein und denselben Mechanismus zu realisieren.
Ob die hinzugefügten Informationen aus den Daten automatisch extrahiert werden, oder ob der Bearbeiter aufgrund seiner Erfahrung eine Klassifikation des Datensatzes vornimmt, spielt hierbei keine Rolle. Der entscheidende Punkt ist der, daß es sich um Informationen handelt, die einerseits den Menschen als zusätzliche Informationsquelle dienen können, andererseits syntaktisch so beschrieben werden, daß sie automatisch bearbeitbar und damit von Workflowsystemen beispielsweise zur Steuerung oder zur Unterstützung von Suchfunktionalität eingesetzt werden können. Im weiteren Verlauf der Arbeit werden deshalb die Begriffe **Metadaten** und **Workflow-Relevante-Daten** synonym verwendet.

Zusammengefaßt kann dies in folgender Anforderung an die $\mathcal{W}$*FLOW* Container ausgedrückt werden:

Anforderung 1:

Ein Container muß die Speicherung von Metainformationen unterstützen. Die Speicherung der Informationen muß in einer Art erfolgen, daß diese sowohl von Menschen, als auch von einem Workflowsystem auswertbar sind.

Als nächstes sollen die Dateninhalte und -formate betrachtet werden, die beim Einsatz und dem Zusammenspiel verschiedener wissenschaftlicher Softwaresysteme zum Einsatz kommen und welche Funktionalität von den Containern zur Speicherung und Verwaltung dafür bereitgestellt werden muß.

Wissenschaftliche Software bzw. Softwaresysteme arbeiten mit einer ganzen Palette verschiedener Speichermethoden, die hier kurz vorstellt werden sollen.

An erster Stelle stehen hier Dateien, die im Dateisystem abgelegt werden. Diese weisen unterschiedliche, oftmals proprietäre Datenformate auf. Aufgrund der Vielzahl der unterschiedlichen Datenformate ist kein Workflowsystem in der Lage, all die unterschiedlichen Formate interpretieren zu können. Eine Konvertierung in ein workflow-einheitliches Format, wie etwa STEP [Owe93], bedeutet, daß für alle vorhandenen Datenformate entsprechende Konverter erstellt werden müssen, was einen nicht zu vertretenden Aufwand darstellt. Dies ist jedoch nicht notwendig, da ja das Workflowsystem nur mit den Metadaten (Workflow-Relevante-Daten) und nicht mit den Anwendungsdaten selbst umgehen muß.

Die Dateien können von der Workflow-Engine als unstrukturierter Datenstrom betrachtet werden, d. h. ein Container muß keinerlei Strukturkenntnisse über die Daten besitzen, da diese in den zugehörigen Anwendungen vorhanden sind und nur gewährleistet werden muß, daß die Anwendungen Zugriff auf den Datenstrom haben. Eine Klassifikation des Inhaltes der Datensätze innerhalb des Workflows ist aber sinnvoll. Sie erlaubt dem Workflowsystem an anderer Stelle (z. B. bei den *Toolservices*) zu entscheiden, von welchen Applikationen die einzelnen Datenströme bearbeitet werden können.

Neben den Dateien stellen Daten, die direkt in Datenbanksystemen gespeichert sind, eine weitere wichtige Informationsquelle für Anwendungsprogramme dar. Hierbei erlaubt die Datenbankschnittstelle den Zugriff auf die benötigten Daten. Es kann es sich dabei sowohl um standardisierte Schnittstellen wie die ODBC Schnittstelle für relationale Datenbanken, als auch um proprietäre Formate, beispielsweise bei verschiedenen Netzwerk– oder hierarchischen Datenbanksystemen, handeln. Liegen die Daten in Datenbanksystemen bereit, so erwarten die Anwendungsprogramme die Daten an der ihnen bekannten Datenbankschnittstelle und in dem speziellen Format, das von der jeweiligen Datenbank bestimmt wird. Solche Daten sind nicht so einfach innerhalb des Workflowsystems zu speichern. Dies gilt auch für Daten von verteilten objektorientierten Systemen.

Durch die wachsende Verbreitung verteilter, objektorientierter Systeme, etwa durch den Einsatz von CORBA, spielen in Netzwerken verteilte Objekte eine zunehmende Rolle im Rahmen von Anwendungen. Beispielsweise können Objekte den Zugriff auf beliebige Datenbanken kapseln, oder wiederum andere Objekte nach Informationen fragen. Der Kernpunkt hierbei ist, daß über die Schnittstelle des Objekts der Zugriff auf die Informationen erfolgt. In welchem Format und wo die Daten gespeichert sind, ist innerhalb der Objektimplementierung gekapselt und von außen nicht sichtbar, so daß oft keinerlei Aussagen über die Datenquellen getätigt werden können. Ähnliches gilt auch für WWW–Anwendungen.

WWW–basierte Informationssysteme werden als Quelle für Daten zunehmend wichtiger. Hierbei stellen WWW-Server die Schnittstelle zu den Daten dar. Bei den Inhalten kann es sich um statische Informationen, die auf den Servern in Form von Dateien liegen, oder aber um dynamisch erzeugte Informationen handeln. Im letzteren Fall generieren sogenannte CGI-Programme[3] , Servlets[4] oder mittels speziellen APIs in einen WWW-Server eingebettete Programme die Informationen und geben sie in Form von WWW-Inhalten zurück. Der Zugriff auf diese Informationen erfolgt mittels URLs.

Der Unterschied der letzten drei Datenquellen zu Dateien besteht darin, daß die Datenquellen nicht als einfacher Datenstrom angesehen werden können, sondern der Zugriff auf die eigentlichen Informationsobjekte über die jeweilige Schnittstelle erfolgt. Kann ein Container einen uninterpretierten Datenstrom noch problemlos einer Anwendung als Ganzes zur Verfügung stellen, ist

[3] **C**ommon **G**ateway **I**nterface (siehe Glossar Seite 180)
[4] siehe Glossar Seite 183

es in den anderen Fällen von einem Container notwendig, die Schnittstelle der Datenquellen nachzubilden. Das ist aber im allgemeinen nicht möglich, da die notwendigen Informationen über Struktur und Herkunft der Daten nicht bekannt sein müssen. Da weiterhin beliebige Anwendungsprogramme eingebunden werden sollen, ist es auch nicht möglich, die Schnittstelle der Anwendungsprogramme zu den Daten zu modifizieren. Aus diesem Grund muß die ursprüngliche Datenquelle erhalten bleiben und eine Art Referenzierung auf diese Datenquelle durchgeführt werden.

Anforderung 2:

Der Container muß in der Lage sein, Daten unterschiedlicher Inhaltstypen zu verwalten und einen einheitlichen Zugriff darauf zu gewährleisten. Dies macht sowohl eine Speicherung von Daten innerhalb des Containers, als auch eine geeignete Referenzierung externer Datenquellen erforderlich, für die eine einheitliche Typklassifikation vorhanden sein muß. Dabei muß das ursprüngliche Format der Daten und der Zugriffsschnittstelle erhalten bleiben.

Neben der Anreicherung einzelner Datensätze mit zusätzlichen Informationen spielen auch noch Beziehungen zwischen Datensätzen eine große Rolle, was wiederum am Beispiel aus der Einleitung erläutert werden soll.

In dem Beispiel zur Fluidik-Simulation benötigen die Werkzeuge unter anderem eine Steuerungsdatei für den Simulationsablauf, Dateien mit dem Geometriemodell, Fortran-Quelldateien mit zum Teil modellabhängigem Fortran-Code und eine Datei, in der die Gitterzerlegung des Modells enthalten ist.

Diese Dateien sind aber nicht unabhängig voneinander. Vielmehr gibt es folgende Abhängigkeiten [CFX97]:

- Eine $1 : n$ Beziehung zwischen der Modell– und Gitterzerlegungsdatei. Das rührt daher, daß aus einer Modelldatei unterschiedliche Zerlegungen generiert werden können.
- Eine $1 : 1$ Beziehung zwischen Steuerungsdatei und dem Modell. Das liegt darin begründet, daß sich die Anweisungen in der Steuerungsdatei, welche den Ablauf einer Simulation regeln, speziell auf das erstellte Modell beziehen.
- Eine $1 : m$ Beziehung zwischen Fortran Quell-Code Dateien und dem Modell. Ein und derselbe Fortran Quell-Code kann eventuell in verschiedenen Modellen genutzt werden.
- Jeweils eine $1 : 1$ Beziehung zwischen Steuerungsdatei, Gitterdatei und den einzelnen Ausgabedateien.

Solche Beziehungen lassen sich mittels Betriebssystemfunktionalität auf der Basis hierarchischer, baumartiger Dateisystemstrukturen kaum modellieren und viele Anwendungssysteme, wie z. B. auch *CFX*, unterstützen den Anwender hierbei nicht. So ist bei den *CFX* Werkzeugen selbst die Namensgebung der Dateien so gewählt, daß nach mehreren Simulationsläufen mit unterschiedlichen Gitterzerlegungen im Arbeitsverzeichnis nicht mehr klar ersichtlich ist, welche Gitterzerlegung zu welchem Modell und zu welchen Ergebnissen gehören. Da aber die obigen Beziehungen

einen wichtigen semantischen Aspekt der Daten repräsentieren, müssen zusätzliche Mechanismen eingeführt werden, um die Beziehungen festhalten. Dieses Problem ist in der Fachwelt bekannt. Einen Lösungsansatz hierzu bietet z. B. das *Rufus* System [SLS+93] zur Verwaltung von semistrukturierten Daten. *Rufus* bietet, neben dem Hinzufügen von Metainformationen zur Annotation von Daten, noch Mechanismen zur Mengenbildung und Referenzierung an, um zunächst zusammenhanglose Daten in Beziehung zueinander zu setzen. Allgemein findet man in der Informatik im wesentlichen zwei Mechanismen, um Beziehungen zwischen Daten auszudrücken: einen Mechanismus zur **Gruppierung** (im *Rufus*-System Mengen) und einen Mechanismus zur **Vernetzung** (im *Rufus*-System Referenzen).

Anforderung 3:

Die Container müssen die Möglichkeit bieten, die einzelnen Datensätze zu einer größeren Struktur zu verbinden, um Beziehungen zwischen den einzelnen Datensätzen ausdrücken zu können.

Die Anforderung von Modellierungsmöglichkeiten zwischen Datensätzen wird durch den Bereich *Scientific Databases* noch dahingehend verschärft, daß die Flexibilität bei der Modellierung von Datenschemata enorm wichtig ist, da sich die Struktur aufgrund des experimentellen Charakters der Workflows sehr schnell ändern kann. Diese Forderung wird durch ein Zitat von John Pfaltz, einem der Autoren des NSF–Workshop Abschlußberichtes über "Scientific Database Management", untermauert:

> *"In scientific database applications it is more important to be able to modify the structure of the database than the data itself."*[5] [Pfa95]

Daraus ergibt sich folgende Anforderung:

Anforderung 4:

Die Datenschemata der Container müssen so flexibel sein, daß sie jederzeit, auch zur Laufzeit, modifiziert werden können.

Nachdem in den vorherigen Ausführungen hauptsächlich Datenmodellierungsaspekte im Vordergrund standen, soll im folgenden nun auf die Funktionalität der Container als Datenspeicher eingegangen werden. Je nach Applikation bzw. Anwendung werden, in Bezug auf Zugriffsmethodik und Speicherungseigenschaften, bestimmte Anforderungen an die Containerkomponente gestellt.

So kann es je nach Anwendung sinnvoll sein, daß ein Container nur ein oder mehrere gleichartige Datenobjekte aufnimmt. Repräsentiert ein Containerobjekt z. B. die eindeutig bestimmte Konfigurationsdatei einer Anwendung, die im System nur einmal existiert, so wird es in einem zugehörigen Workflow eventuell einen Container geben, der nur **ein** Datenobjekt enthält, das diese Konfigurationsdatei **referenziert**. Eine Modifikation des Anwendungsinhaltes des Datenobjektes führt dann zu einer Modifikation der Konfigurationsdatei. Andererseits kann im gleichen Workflow ein Container existieren, der mehrere vorgefertigte Konfigurationsdateien der

[5] Übersetzung: *"In wissenschaftlichen Datenbankanwendungen ist es wichtiger in der Lage zu sein, die Struktur der Daten zu verändern, als die Daten selbst".*

Anwendung als Anwendungsinhalt der Datenobjekte enthält. Der Workflow kann dann in einem Arbeitsschritt die Auswahl einer geeigneten Konfigurationsdatei aus der Menge der vorliegenden Vorlagen erlauben.
Bei den Modellobjekten aus dem Beispiel der Fluidiksimulation, werden im allgemeinen inkrementelle Änderungen an einem Modell vorgenommen, die am geeignetsten durch eine Versionierung beschrieben werden. Die Versionierung von Objekten ist in vielen wissenschaftlichen Anwendungsbereichen sinnvoll, nicht zuletzt im Bereich der Informatik.
Im Falle von Containern die anstatt eines Objektes eine Vielzahl von Objekten verwalten, stellt sich die Frage nach den Zugriffsmethoden auf die Objekte. Im einfachsten Fall kann jeweils auf ein Objekt über eine eindeutige Objektidentität (OID), einen Namen oder dergleichen, zugegriffen werden. Häufig ist jedoch auch eine Mehrfachselektion, in der eine Gruppe von Objekten oder alle Objekte ausgewählt werden können, wünschenswert. In der Regel fordert der Anwender hierbei, daß Daten nicht nur über einen Namen oder eine OID selektiert werden können, sondern daß intelligente Zugriffsmechanismen vorhanden sind, die Objekte über Angabe geforderter Eigenschaften selektieren. Dies legt die Konzipierung eines völlig allgemein gehaltenen Selektionsmechanismus, basierend auf den Metainformationen zu den Anwendungsdaten (ähnlich einer Datenbank-Query-Sprache), nahe. Die verschiedenen Ausprägungen von Containern und entsprechende Zugriffe darauf sind in Abbildung 3.3 und 3.4 zu sehen. Die erste Abbildung zeigt den Zugriff auf genau ein Objekt. Im einfachen Fall (links) wird auf das einzig vorhandene Objekt zugegriffen, andernfalls erfolgt die Auswahl genau eines Objektes aus einem Versionsbaum (rechts).
Abbildung 3.4 zeigt verschiedene Zugriffsarten auf mengenwertige Container. Links ist der selektive Zugriff auf genau ein Element einer Menge dargestellt, in der Mitte der Zugriff auf alle zur Verfügung stehenden Elemente und rechts auf eine Teilmenge davon.

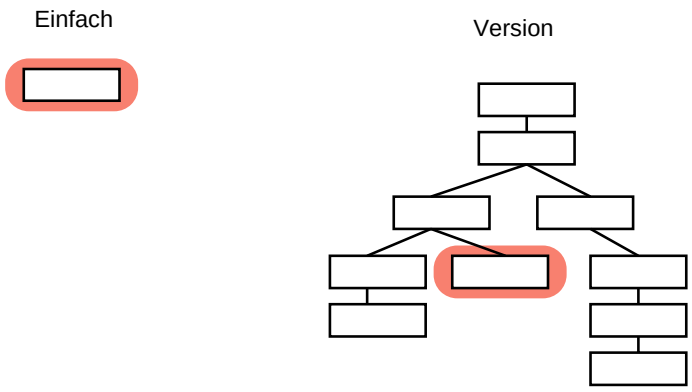


Abbildung 3.3: *Zugriff auf einfachen und versionierten Container*

Anforderung 5:

Ein Container muß, je nach Anwendungsfall, verschiedene Speichermodi unterstützen sowie allgemein gehaltene Zugriffsmethoden auf die einzelnen Datensätze bereitstellen.

Spätestens an dieser Stelle wird ersichtlich, daß die zu realisierende Containerkomponente über Datenbankfunktionalität verfügen muß. Datenbanken bieten nicht nur effiziente Unterstützung bei Suchanfragen, sondern auch Transaktions– und Recoverymechanismen.

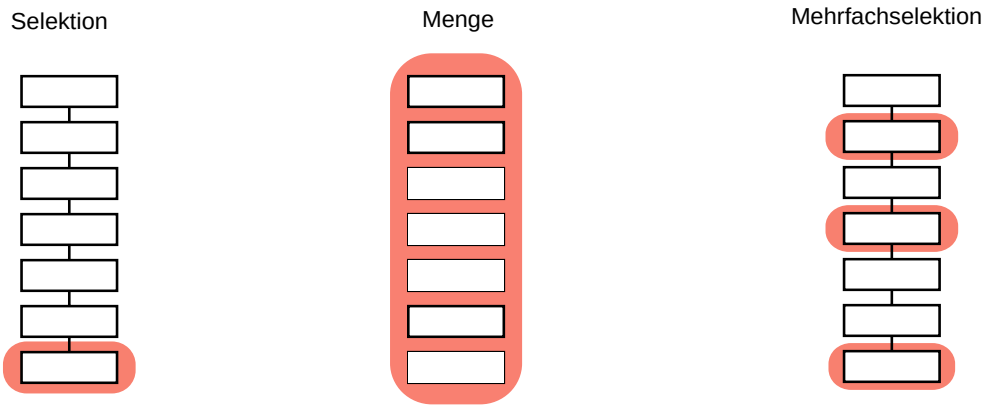


Abbildung 3.4: *verschiedene Zugriffe auf mengenwertige Container*

Insbesondere Transaktionsmechanismen spielen in dem Fall, daß mehrere Datensätze zurückgeschrieben werden müssen und ein paralleler Zugriff verschiedener Benutzer auf die Datenbank erfolgt, eine große Rolle, sichern sie doch die Konsistenz der Daten zu. Bei den hier benötigten Transaktionen handelt es sich ausschließlich um *kurze Transaktionen*, die den Schreibvorgang in eine atomare, isoliert ablaufende Aktion kapseln. Im Gegensatz dazu handelt es sich bei den Mechanismen, die einen oder mehrere Datensätze für eine längere Bearbeitung während eines Arbeitsschrittes kapseln, um sogenante *lange Transaktionen*, die mit anderen Methoden wie die hier auftretenden kurzen Transaktionen behandelt werden müssen. An dieser Stelle wird für die Datenhaltungskomponente Container nur gefordert, daß sie über klassische (kurze) Transaktionen, die zumeist den *ACID*[6]–Eigenschaften genügen, verfügt. Ein Mechanismus von *langen Transaktionen* oder ein erweitertes Transaktionskonzept, wie es **auch** in einem Workflowsystem benötigt wird, wird an anderer Stelle (Abschnitt 3.2.2) von den *Aktivitäten* bereitgestellt.

Die Forderung nach Datenbankmechanismen bei der Verwaltung externer Daten, wie Transaktionen Suchanfragen etc., wird von *CONCERT* [RB95, BRS96] erfüllt. Die Autoren dieses Systems propagieren ein "datenloses" Datenbanksystem, dessen Aufgabe es ist, in externen Dateien abgelegten Daten Datenbankdienste, wie Indizierung, Replikation, Query Processing etc., zur Verfügung zu stellen. Das hier beschriebene System erlaubt es, im Gegensatz zu *CONCERT*, sowohl "datenlos" zu arbeiten, als auch Anwendungsdaten selbst im Container zu speichern.

Anforderung 6:

Von der Containerkomponente müssen allgemeine Datenbankdienste, wie Transaktionsmechanismen, Suchanfragen, etc. zur Verfügung gestellt werden.

Eine weitere Anforderung, welche durch den Einsatz der Komponente im Rahmen von $\mathcal{W}*FLOW$ gestellt wird, ist die Möglichkeit der dezentralen Verwaltung und Speicherung der anfallenden Daten. Hierbei soll die Möglichkeit bestehen, die Daten auf verschiedene, eigenständige Datenbanken, die auf unterschiedlichen Rechnern laufen, zu verteilen. Dies erhöht die Fehlertoleranz des Systems und führt zu einer Lastverteilung.

[6] **A**tomicity, **C**oncurrency, **I**solation, **D**urability

Anforderung 7:

Die Container müssen verteilt im Netzwerk, ohne zentrale Datenbank, einsetzbar sein.

Zusammengefaßt kann gesagt werden, daß die Aufgabe einer Speicherkomponente für Produktionsdaten eines Workflows darin besteht, die Daten zu verwalten, mit Zusatzinformationen zu versehen und den Zugriff auf die unterschiedlichsten Datenquellen zu gewährleisten. Der Container repräsentiert hierbei eine Zwischenschicht[7], die von den betriebs– oder datenquellenspezifischen Besonderheiten abstrahiert und sowohl dem Benutzer, als auch dem Workflowsystem, eine einheitliche Schnittstelle für den Zugriff und die Verwaltung der Daten bereitstellt.

3.1.3 Container

Im folgenden soll nun die konkrete Realisierung der $\mathcal{W}*FLOW$–Containerkomponente vorgestellt werden. Abbildung 3.5 zeigt die Rolle der Containerkomponente zwischen Anwender/Workflowsystem auf der einen Seite und externen, verteilten, heterogenen Daten auf der anderen Seite. Wie in der Abbildung graphisch dargestellt, stellt ein Container eine Zwischenschicht dar, die Konzepte wie Metainformationen, Strukturierungsinformationen und Zugriffsmethoden bereitstellt.

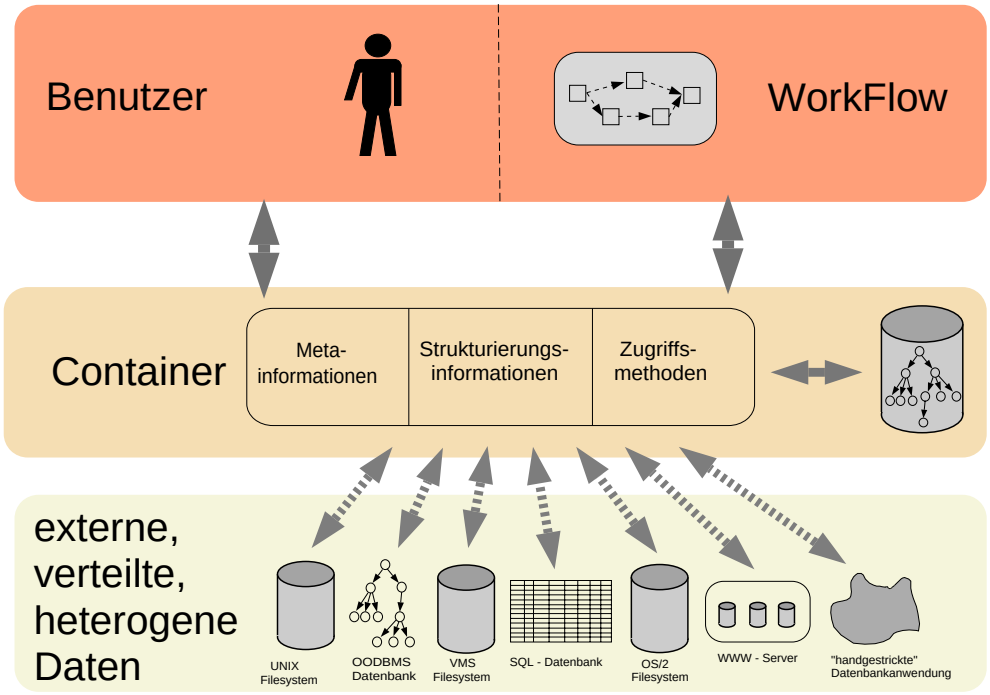


Abbildung 3.5: *Container als Schnittstelle zwischen Benutzer, Workflowsystem und Daten*

Nach der Festlegung der Anforderungen im vorherigen Abschnitt sollen nun die im Rahmen der vorliegenden Dissertationsschrift entstandenen neuen Definitionen bezüglich des $\mathcal{W}*FLOW$–Containers (Definitionen 3.1 bis 3.3) vorgestellt werden. Abbildung 3.6 zeigt dazu die komplette Struktur eines $\mathcal{W}*FLOW$–Containers, auf die in den folgenden Definitionen näher eingegangen werden soll.

[7] In der englischsprachigen Literatur findet man hierfür das Wort *Middle–Tier* (siehe Glossar Seite 181), das sich inzwischen auch im deutschsprachigen Raum eingebürgert hat.

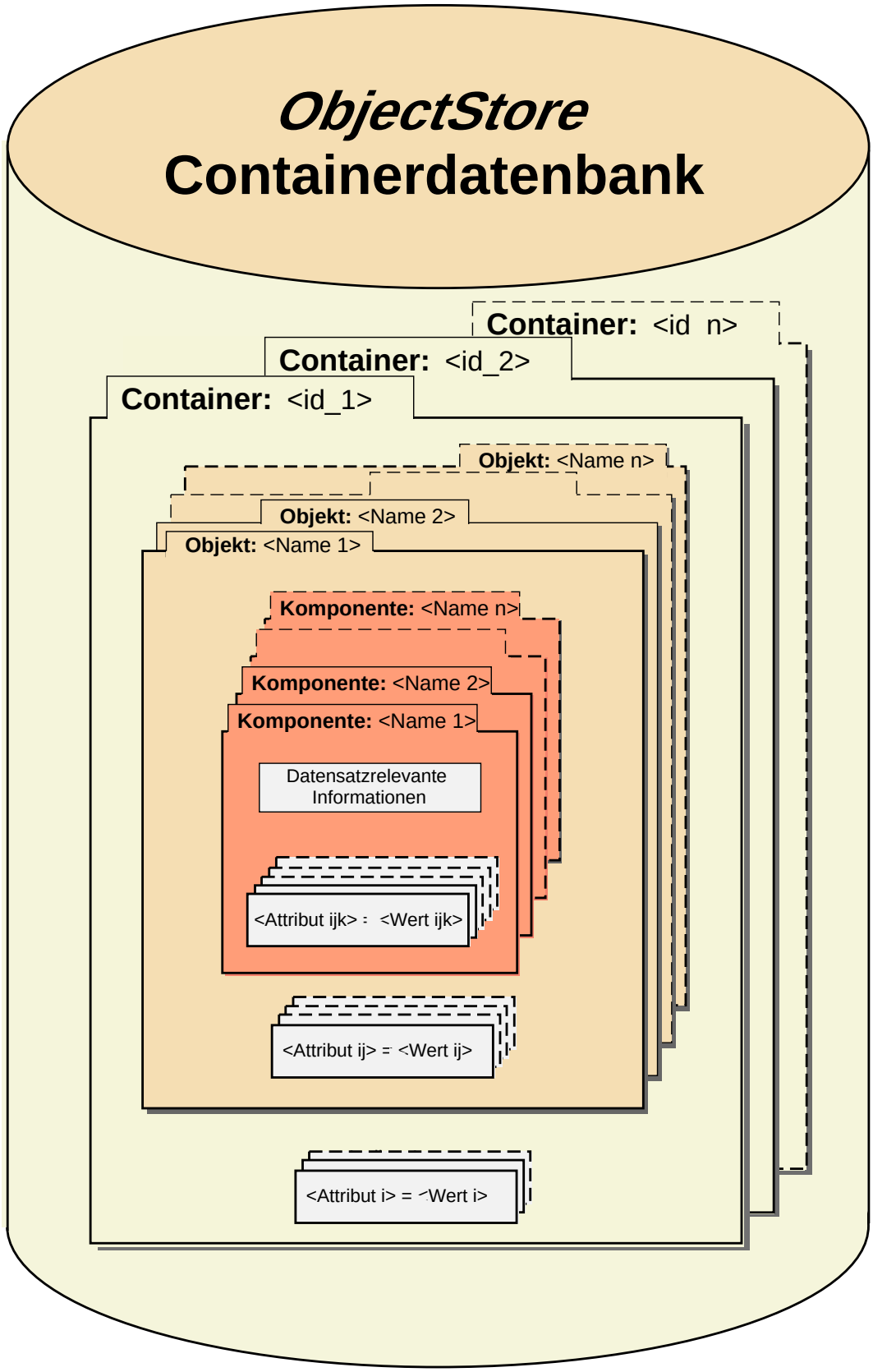


Abbildung 3.6: *Gesamtstruktur einer Containerdatenbank*

Definition 3.1 ($\mathcal{W}$*FLOW–Container) *Ein Container ist ein Speicher zur Verwaltung der Metadaten von Anwendungen und gegebenenfalls der zugehörigen Applikationsdaten auf semantisch hohem Niveau.*

1. *Einem $\mathcal{W}$*FLOW–Container kann eine beliebige Anzahl von $\mathcal{W}$*FLOW–Attributen zugeordnet werden.*
2. *Ein $\mathcal{W}$*FLOW–Container besitzt ein Objekt–Dictionary zur Aufnahme von einem oder beliebig vielen $\mathcal{W}$*FLOW–Container-Objekten.*
3. *Ein $\mathcal{W}$*FLOW–Container besitzt eine API, welche das Eintragen, Modifizieren und Löschen von Containerobjekten und Attributen erlaubt, sowie die Formulierung von Suchanfragen auf den Containerinhalt gestattet.*
4. *Ein oder mehrere $\mathcal{W}$*FLOW–Container werden auf eine ObjectStore Datenbank abgebildet, über die der Container Transaktionen, Recovery– und Suchoperationen zur Verfügung stellt.*

Beliebige $\mathcal{W}$*FLOW–Attribute (siehe Abschnitt 2.3.1) können für die Speicherung von Verwaltungs- und Metainformationen des Containers eingesetzt werden. Über solche Attribute können dem Container z. B. eine Beschreibung, Kurzbeschreibung oder ein Name zugeordnet werden, sowie Eigentümer oder Zugriffsrechte auf den Container festlegt werden. Es soll an dieser Stelle schon erwähnt werden, daß Containerobjekte (und auch deren Komponenten) ebenfalls beliebige $\mathcal{W}$*FLOW–Attribute zugeordnet werden können. Die $\mathcal{W}$*FLOW–API selbst sieht keinerlei feste Attribute, wie Name, Beschreibung oder ähnliches für Container (und auch für alle anderen Konstrukte) vor, da solche festen Attribute am besten auf der Anwendungsebene (also oberhalb der $\mathcal{W}$*FLOW–API) übergreifend definiert werden sollen, um ein allgemeines *Look and Feel* für Workflows vorzugeben. Einen Ansatzpunkt, welche Attribute zur Beschreibung einer Datenressource relevant sind, bietet die Dublin Core (DC) Metadata Initiative [DC99], die einen Satz von 15 Metadaten Elementen (z. B. Title, Subject, Publisher, ...) definiert hat, welche elektronische Dokumente beschreiben können.

Containerobjekte werden innerhalb eines Containers in einem *Dictionary* gespeichert. Ein *Dictionary* ist eine Tabelle, in die Schlüssel–Wert Paare eingetragen werden können. Der Zugriff auf die Werte erfolgt jeweils über den zugehörigen Schlüsseleintrag. Dabei wird nicht etwa ein Containerobjekt-Name, sondern beim Eintrag automatisch eine eindeutige Objekt-ID als Schlüssel benutzt. Trotzdem lassen sich Container-Objekte über $\mathcal{W}$*FLOW–Attribute Namen zuordnen. Die $\mathcal{W}$*FLOW–API ermöglicht dann ein Suchen von Objekten anhand dieser Attribute.

Dadurch, daß ein oder mehrere Container durch eine eigene Datenbank repräsentiert werden, ist eine flexible Modellierung der Datenverteilung eines Workflows möglich. Die Einschränkungen einer zentralen Datenbankkomponente in Bezug auf Skalier– und Verfügbarkeit werden durch die Möglichkeit, mehrere Datenbanken mit Containern verteilt im Netzwerk zum Einsatz kommen zu lassen, vermieden. Ein Workflow-Designer kann, wie er es braucht, Container gruppieren und in gemeinsame Datenbanken packen. Dies kann so geschehen, daß Daten räumlich lokal gehalten werden oder semantisch zusammengehörende Daten in gemeinsamen Datenbanken liegen.

Abbildung 3.7 zeigt, wie die Modell– und Steuerungsdaten des Fluidik-Simulations-Workflows aufgeteilt sind. Der linke Container Modelle enthält die zwei Objekte (Filter und Kanal), während sich im Steuerungsdaten-Container drei Objekte befinden. Zusatzattribute sind den Containern in diesem Beispiel nicht zugeordnet. Die Objekte im linken Container repräsentieren die unterschiedlichen Modelle, die simuliert werden können. Analog befinden sich im rechten Container

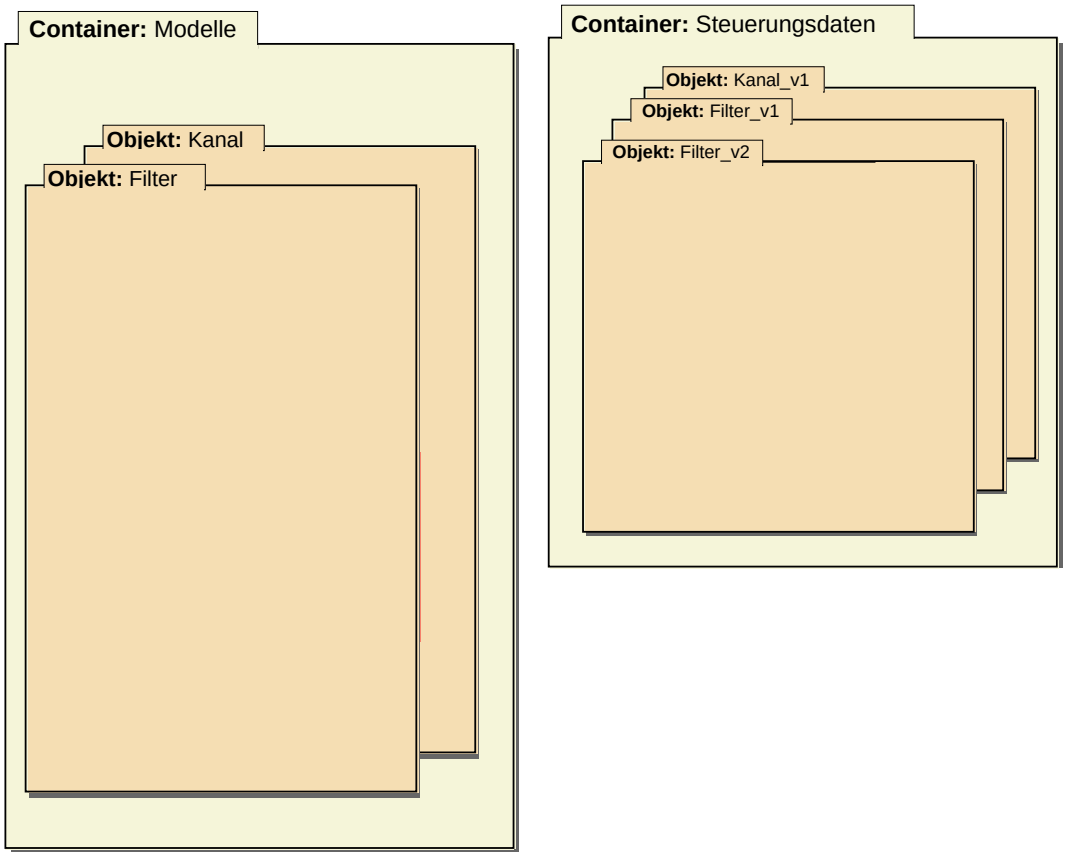


Abbildung 3.7: Beispiel CFX: *Struktur der Container für die Eingangsdaten einer Simulation*

unterschiedliche Steuerungsanweisungen, die für einen Simulationslauf benutzt werden können. Der Code zur Erzeugung der zwei Container ist in Beispiel 3.1.1 zu sehen. Als erste Anweisung (0) wird die `WildFlow::API` Bibliothek eingebunden, welche die Funktionalität zum Aufbau der Container enthält. Die Anweisung (1) kapselt die nachfolgenden Anweisungen (2) bis einschließlich (6) innerhalb einer Schreibtransaktion, d. h. die Ergebnisse der Anweisungen werden entweder alle ausgeführt und die Modifikationen sichtbar oder, im Fehlerfalle, wird keine der Modifikationen der Anweisungen innerhalb der Transaktion sichtbar. In den Anweisungen (2) und (3) wird der Container Modelle innerhalb eines bestimmten Repositories (*/data/CFX*) neu angelegt. Die Anweisungen unter (4) tragen die beiden Containerobjekte Filter und Kanal in den Container ein. Die Anweisungen (5) und (6) erzeugen in Analogie zu den oberen Schritten den Steuerungsdaten Container mit den entsprechenden Containerobjekten. Die Anweisung (7) behandelt den Fall, daß ein Fehler im Rahmen der Transaktionsbearbeitung aufgetreten ist. In diesem Fall wird das Programm beendet und eine entsprechende Fehlermeldung ausgegeben.

Es folgt nun die Definition des Containerobjekts.

Definition 3.2 (*W∗FLOW*–Container-Objekt) *Ein Containerobjekt erlaubt die Gruppierung, Vernetzung und Metabeschreibung von Daten. Es besteht aus folgenden Einzelteilen:*

1. *Ein W∗FLOW–Container-Objekt kann beliebig viele W∗FLOW–Attribute enthalten, die das Objekt näher beschreiben.*

```
use WildFlow::API;                                                    # (0)

my $modell_cont;
my $steuer_cont;

#
#  Start Transaktion
#
begin 'update', sub {                                                 # (1)
  #
  #  Container 'Modelle' erzeugen
  #
  my $rep = WildFlow::ContainerRepository->create('/data/CFX');       # (2)
  $rep->clear();
  $modell_cont = WildFlow::Container->new(NAME=>'Modelle',            # (3)
                                          REPOSITORY=>$rep);
  #
  # 2 Objekte in diesem Container erzeugen
  #
  my $filter_objekt = $modell_cont->new_object(NAME=>"Filter");       # (4)
  my $kanal_objekt  = $modell_cont->new_object(NAME=>"Kanal");        #


  $steuer_cont = WildFlow::Container->new(NAME=>'Steuerungsdaten',    # (5)
                                          REPOSITORY=>$rep);          #
  #
  # 3 Objekte in diesem Container erzeugen
  #
  my $filter_1 = $steuer_cont->new_object(NAME=>"Filter_v1");         # (6)
  my $filter_2 = $steuer_cont->new_object(NAME=>"Filter_v2");         #
  my $kanal_1  = $steuer_cont->new_object(NAME=>"Filter_v1");         #

  #
  #  Weiterer Code folgt hier; siehe Beispiele 3.1.2 und 3.1.5
  #


};
die "$@\n" if $@;                                                     # (7)
...
```

Beispiel 3.1.1: Codebeispiel zur Erzeugung der Container aus Abbildung 3.7

2. *Ein $\mathcal{W}$*FLOW–Container-Objekt besitzt ein Komponenten–Dictionary zur Aufnahme beliebig vieler Daten(ströme) eines Containerobjektes.*
3. *Ein $\mathcal{W}$*FLOW–Container-Objekt besitzt eine API zum Anlegen und Löschen von Attributen und Komponenten sowie zum Zugriff auf Objekte.*

Die $\mathcal{W}$*FLOW–Attribute können für die Speicherung von Metainformationen, Referenzen auf andere $\mathcal{W}$*FLOW–Objekte (ermöglicht die Vernetzung von Objekten), Referenzen auf externe Datensätze sowie zur Formulierung berechenbarer Attribute, die Workflow-Relevante-Daten aus den Datensätzen extrahieren, eingesetzt werden, was die Anforderungen (1), (2) und (3) aus Abschnitt 3.1.2 abdeckt.

Ein Containerobjekt erlaubt es, Datenobjekte zu gruppieren. Dies wird dadurch realisiert, daß ein Containerobjekt eine beliebige Anzahl von namentlich gekennzeichneten Komponenten in einem *Dictionary* aufnehmen kann. Es kann somit als Strukturierungsmittel, vergleichbar einem *Record* in `Pascal`, bzw. einem *struct* in C/C++ angesehen werden. Im Gegensatz dazu erlaubt der Einsatz von *Dictionaries* aber eine weitaus größere Flexibilität, da die Struktur nicht fix ist, sondern zur Laufzeit geändert werden kann. Dadurch ist es auch möglich, daß beispielsweise die einzelnen Datensätze eines $\mathcal{W}$*FLOW–Containers eine unterschiedliche Struktur besitzen können.
Durch das Komponenten-*Dictionary* und die Attributreferenzen sind die zwei elementaren Strukturierungsmittel, **Gruppierung** und **Referenzierung** vorhanden (Anforderungen (4) und (5)).

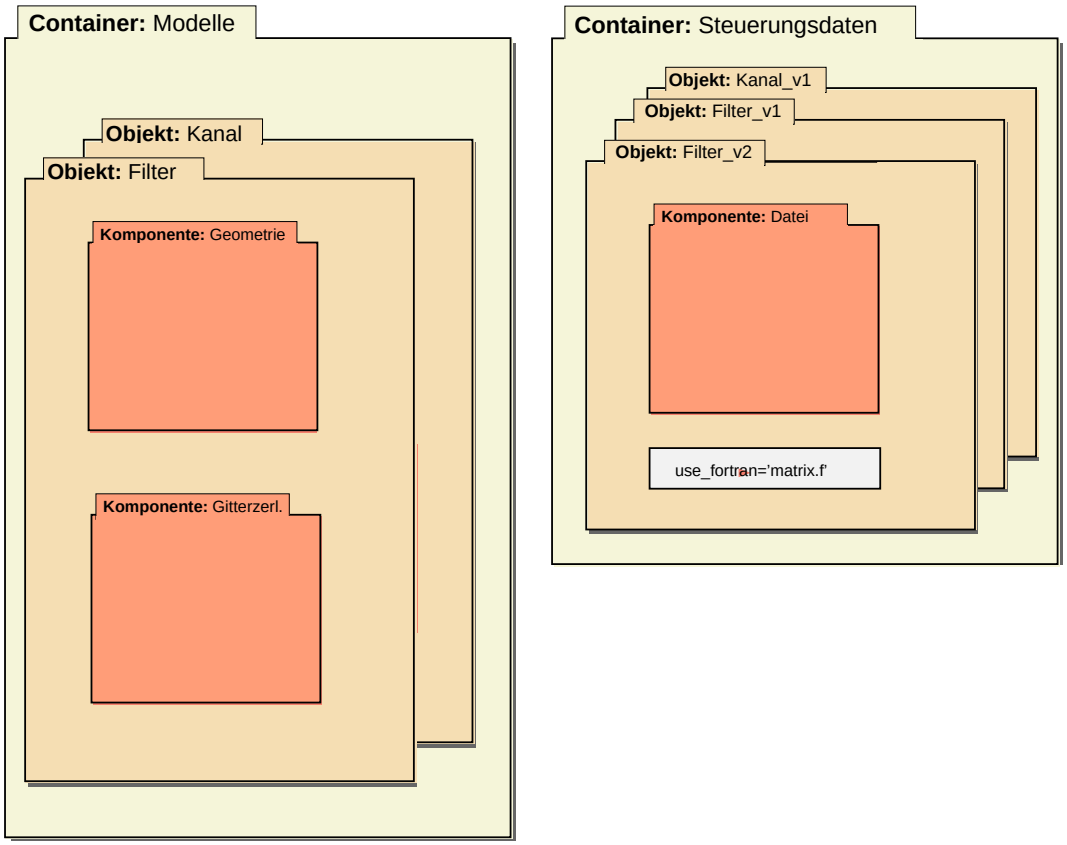


Abbildung 3.8: Beispiel CFX: *Erweiterte Struktur der Container für die Eingangsdaten einer Simulation*

Abbildung 3.8 zeigt als Erweiterung von Abbildung 3.7 die Modellierung der $\mathcal{W}$*FLOW–

Container-Objekte in dem Fluidik-Simulations-Workflow. Das Container-Objekt Filter, im Container Modelle, besteht aus den beiden Komponenten Geometrie und Gitterzerlegung. Das Container-Objekt Filter_v2 im Steuerungsdaten-Container besitzt, neben der einzelnen Komponente Datei, noch ein zusätzliches Attribut use_fortran, das den Namen der verwendeten Fortran-Code-Datei enthält.

Der Code hierzu ist in Beispiel 3.1.2 dargestellt. Die Anweisungen unter Punkt (9) definieren für die in Beispiel 3.1.1 erzeugte Containervariable `$filter_2` das zusätzliche Attribut *use_fortran* und fügen eine neue Komponente mit der Bezeichnung Datei hinzu. Anweisung (10) liefert das Filter–Objekt aus der Containervariablen `$modell_cont` und in (11) wird in das extrahierte Objekt eine neue Komponente Geometrie angelegt. Die Anweisung in (12) erzeugt die mengenwertige Komponente Gitterzerlegung und Anweisung (13) trägt ein Element in die Komponente ein.

```
#
# ... (Fortsetzung von Beispiel 3.1.1, Seite 60)
#


$filter_2->insert_attribute(use_fortran=>'matrix.f');              #  (9)
my $cmp = $filter_2->new_component(NAME=>"Datei");                 #

my $filter_obj = $modell_cont->get_object('Filter');               # (10)

my $cmp1 = $filter_obj->new_component(NAME=>"Geometrie");          # (11)

                                                                   # (12)
my $set_cmp = $filter_obj->new_set_component(NAME=>'Gitterzerlegung');
my $cmp_item = $set_cmp->new();                                    # (13)
...
```

Beispiel 3.1.2: Codebeispiel zur Erzeugung der Containerobjekte aus Abbildung 3.8

Als letzte Definition im Rahmen der $\mathcal{W}$*FLOW*–Container wird nun festgelegt, wie Daten in den Container-Objekt-Komponenten gespeichert werden.

Definition 3.3 ($\mathcal{W}$*FLOW*–Komponente) *Eine $\mathcal{W}$*FLOW*–Container-Objekt-Komponente beschreibt einen Datenstrom und speichert diesen Datenstrom oder eine Referenz auf ihn. Im einzelnen enthält eine $\mathcal{W}$*FLOW*–Container-Objekt-Komponente die folgenden Teile:*

1. *Eine $\mathcal{W}$*FLOW*–Komponente kann eine beliebige Anzahl von $\mathcal{W}$*FLOW*–Attributen enthalten, die den Datenstrom näher beschreiben.*
2. *Eine $\mathcal{W}$*FLOW*–Komponente speichert Datenströme (als Bytestream) oder enthält eine Referenz auf ihn in Form einer* URL.
3. *Jedem Datenstrom ist eine Typinformation im MIME Format (siehe Erklärung unten) zugeordnet.*
4. *Eine $\mathcal{W}$*FLOW*–Komponente kann als einfache, versionierte oder mengenwertige Komponente vorliegen.*
5. *Eine $\mathcal{W}$*FLOW*–Komponente besitzt eine API zum Eintragen, Auslesen, Zugriff, Löschen und Erzeugen von Attributen und Datensätzen.*

6. *Die Methoden zum Zugriff auf die Datenelemente können überladen werden, d. h. es können mehrere Methoden mit dem selben Namen existieren, von denen zur Laufzeit anhand der Kontextinformationen (der übergebenen Parameter) die jeweils richtige Implementierung ausgewählt wird.*

Die $\mathcal{W}$*FLOW*–Attribute werden für die gleichen Aufgaben wie bei einem $\mathcal{W}$*FLOW*–Container-Objekt eingesetzt.

Zusätzlich zu dem Datensatz oder der Referenz wird noch eine Typinformation im MIME-Format mit gespeichert. MIME [BF93] Typen wurden ursprünglich entwickelt, um die Beschränkungen bei der Übertragung von Daten per e-Mail zu überwinden. Das im Internet Bereich eingesetzte Protokoll SMTP [Cro82] zur Übertragung von e-Mail erlaubt nur die Übertragung von 7 bit ASCII Zeichen. Um diese Einschränkungen zu umgehen und beliebige Dateien übertragen zu können, wurde der MIME Standard entwickelt, der die ASCII codierte Übertragung und Beschreibung beliebiger Arten von Daten erlaubt. Der MIME Standard beinhaltet ein erweiterbares Typisierungskonzept für Datenströme, das momentan über 100 vordefinierte Inhaltstypen unterstützt. Neben diesen ist es möglich, beliebig weitere, nicht standardisierte Inhaltstypen zu definieren, die alle mit dem Präfix 'application/x–' beginnen. Ein Inhaltstyp spaltet sich in einen Typ und Untertyp auf. Während der Typ allgemein angibt, um was für einen Inhaltstyp es sich handelt (z. B.: Text, Video, Audio, ...), gibt der Untertyp die spezielle Kodierung der Daten an (GIF, JPEG, ...). So lautet die Beschreibung einer Bilddatei im Graphic Interchange Format etwa image/gif.
Neben dem Einsatz innerhalb der Mailprogramme fand der MIME–Standard auch außerhalb der reinen e-Mail Anwendung, z. B. wenn es darum geht, den Inhaltstyp von Daten zu beschreiben. So werden die Daten, die von einem WWW–Server an einen Browser geschickt werden, im MIME–Format übertragen. Weiterhin benutzen die heutigen gängigen Desktop-Systeme, wie *Windows 95/98/NT*, *UNIX CDE* und *Linux KDE* MIME-Typen zur Typklassifikation ihrer Daten.

Da die Daten sowohl von den Containern verwaltet, als auch den Anwendungen an der ursprünglichen Schnittstelle zur Verfügung gestellt werden müssen, wird im Rahmen dieser Arbeit dem Ansatz aus [BRS96] und [GJSO91] gefolgt, der eine Datenhaltung auch außerhalb der Datenbank ermöglicht. Im Gegensatz zu den obigen Ansätzen erlaubt es der Ansatz hier jedoch auch, daß die Daten innerhalb der Datenbank abgelegt werden können.

Wird der Datensatz innerhalb der Containerdatenbank gespeichert, so stellt die Komponente geeignete Methoden zum Eintragen, Auslesen, Löschen und Erzeugen zur Verfügung. Im Falle, daß die Daten außerhalb der Container gespeichert werden, müssen dem Container geeignete Methoden zum Zugriff bereitgestellt werden. Das erfolgt, indem die Containermethoden für die jeweiligen externen Daten redefiniert werden können. Im folgenden Beispiel wird dies, anhand des Zugriffs auf eine SQL–Datenbank, demonstriert.
Die $\mathcal{W}$*FLOW*–Komponente stellt insgesamt sechs Methoden, die redefiniert werden können, zur Verfügung. Die Methoden sind in Tabelle 3.1 aufgelistet. Diese werden dann vom Container für den Zugriff auf die entsprechenden Daten genutzt.

Als Beispiel sollen die Redefinitionen der Zugriffsmethoden INITIALIZE (Beispiel 3.1.3) und GET (Beispiel 3.1.4) für den Zugriff auf eine relationale Datenbank gezeigt werden.
Im vorliegenden Beispiel verwaltet ein $\mathcal{W}$*FLOW*–Container die Informationen einer kompletten Datenbank. Das Beispiel stellt gleichzeitig beispielhaft die Anbindung eines relationalen Datenbank an das $\mathcal{W}$*FLOW*–Containerkonzept dar. Diese Sachverhalt ist ganz allgemein in Abbildung 3.9 graphisch dargestellt. Ein Container (links) repräsentiert eine Datenbank (rechts).

Tabelle 3.1: *Einträge in der Operatorendatenbank der Container*

Methode	Beschreibung
INITIALIZE	Initialisierungsaufgaben, wie beispielsweise das Öffnen einer Datenbank.
CREATE	Anlegen eines externen Datenobjektes.
GET	Auslesen eines Datenobjektes.
DELETE	Löschen eines Datenobjektes.
PUT	Eintragen eines Datenobjektes.
DESTROY	Aufräumaktivitäten, wie z. B. das Schließen einer Datenbank.

Als Metainformationen sind im Container der Name der Datenbank und der Rechner auf dem sie läuft abgelegt. Diese Informationen werden innerhalb der Methode INITIALIZE (Beispiel 3.1.3 (1)) ausgelesen und zum Aufbau der Verbindung mit der Datenbank (2) genutzt. Mögliche weitere Informationen können z. B. Benutzername und Passwort zur Authorisierung bei der Datenbank sein. Jede Tabelle der Datenbank (⟨tab i⟩) wird durch ein Containerobjekt repräsentiert. Relevante Metainformationen sind in diesem Fall z. B. der Name der Tabelle, das Schlüsselattribut[8] oder auch eine Auswahl von Tabellenspalten, die für die Anwendung von Belang sind. Die eigentlichen Datensätze werden durch mengenwertige Komponenten repräsentiert, wobei die Identifikation des jeweiligen Datensatzes durch den Wert des Schlüsselattributs (siehe auch (1) in Methode GET, Beispiel 3.1.4) erfolgen kann.

```
sub INITIALIZE {
    my $self  = shift;

    # Der Name der Datenbank ist im
    # Container in Attribut 'id' abgelegt
    #
    my $db_name = $self->object->container->id;                          # (1)
    my $db_host = $self->object->container->host;                        #

    # Datenbank oeffnen, falls noch kein
    # gueltiges Handle im Container vorhanden
    #
    $self->object->container->dbh ||= Msql->Connect($db_host, $db_name);  # (2)
}
```

Beispiel 3.1.3: Codebeispiel für die Methoden INITIALIZE

Auf diese Art und Weise ist es sehr einfach möglich, einem Container neue, externe, ihm zuvor unbekannte Datentypen beizubringen. Blott et. al. [BRS96, RB95], welche sich ebenfalls mit der Verwaltung externer Daten im Rahmen von Datenbanken (*CONCERT*) befassen, wählen hier einen ähnlichen Ansatz, der dahin geht, daß sie mittels externer Funktionen neue Datentypen auf bereits bekannte Typen abbilden. Der Mechanismus wird dort als '*likeness*' bezeichnet. Der Unterschied der beiden Ansätze liegt darin, daß bei *CONCERT* die Abbildungsfunktionen in

[8] Im allgemeinen können auch mehrere Attribute den Schlüssel bilden, der Einfachheit halber wird im Beispiel aber nur von einem Schlüsselattribut ausgegangen.

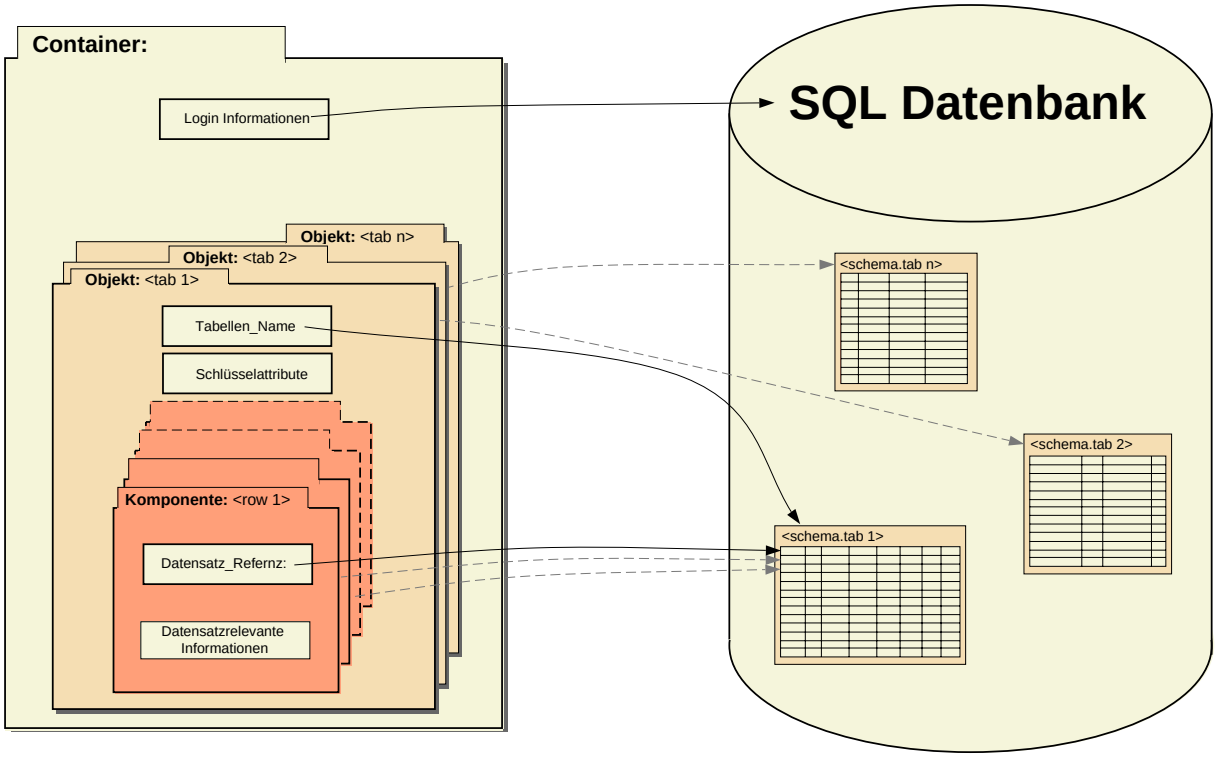


Abbildung 3.9: *Abbildung zwischen einem Container und einer SQL–Datenbank*

```
sub GET {
    my $self  = shift;
    my $result;
    # Tabellenname aus Objektattribut auslesen
    my $table = $self->object->id;

    # id des Datensatzes aus Komponente auslesen
    my $id    = $self->id;                                          # (1)

    # Schluesselfeld des Datensatzes aus Objekt auslesen
    my $key   = $self->object->key;

    # Anfrage formulieren
    my $query = "SELECT * FROM $table WHERE $key=$id";

    # Anfrage absenden
    #
    $dbh->Query($query);
    # Resultate abholen
    #
    my @row = $sth->fetchrow()
    return @row;
}
```

Beispiel 3.1.4: Codebeispiel für die Methoden GET

einer Library zur Verfügung gestellt werden müssen, was einen separaten Übersetzungslauf erfordert, während bei den Containern eine Schnittstelle zur Skriptsprache *Perl* zur Verfügung gestellt wird. Hierdurch wird eine einfachere und schnellere Integration, ohne zusätzlich notwendige Entwicklungswerkzeuge wie Compiler, Linker, etc. , ermöglicht.

Andererseits geht die Funktionalität von *CONCERT* über die der Container hinaus, da es sich, im Gegensatz zu den Containern, auf wohlstrukturierte, applikationsspezifische Daten stützen kann [BV95a, BV95b]. Aus diesem Grund kann die Struktur externer Dateien mittels applikationsspezifischer Parser genauer analysiert und eventuell Listenoperationen zur Verfügung gestellt werden. Außerdem werden die Informationen über die externen Datentypen von *CONCERT* zur Zugriffsoptimierung genutzt [RB95].

Abbildung 3.10 zeigt das endgültige Aussehen der Container aus dem Fluidikbeispiel. Bei der Komponente Gitterzerlegung handelt es sich hierbei um ein mengenwertiges Objekt. Dies erlaubt die Modellierung der $1 : n$ Beziehung zwischen Geometrie und Gitterzerlegung, wie sie in Abschnitt 3.1.2 gefordert wurde. Die mengenwertige Komponente erlaubt die Aufnahme von beliebig vielen Datensätzen. Ein zusätzliches Attribut *Gittergröße* in der Komponente Gitterzerlegung bestimmt die Feinheit einer Gitterzerlegung[9] eines Modells. Weiterhin ist ein Attribut in der Komponente von Objekt Filter_v2 definiert, das eine Referenz auf das Filter-Objekt im Container Modelle darstellt. Hiermit wird die $1 : 1$ Beziehung zwischen Steuerungsdatei und Modell repräsentiert.

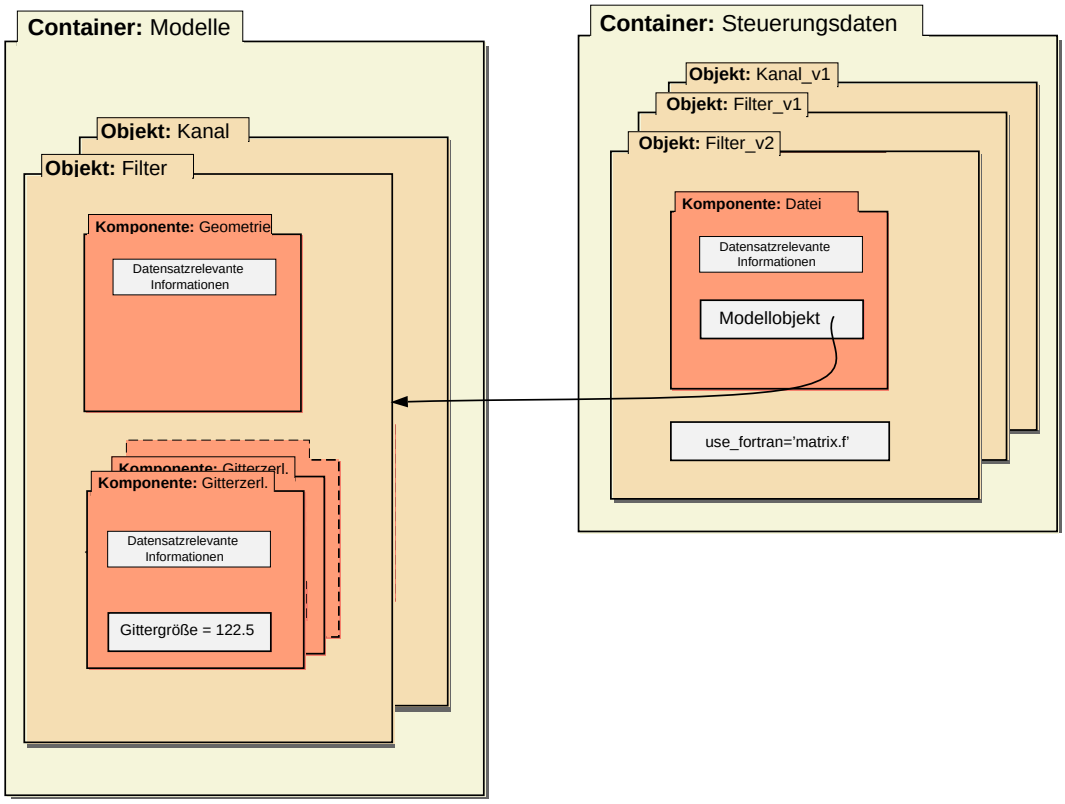


Abbildung 3.10: Beispiel CFX: *Entgültige Struktur der Container für die Eingangsdaten einer Simulation*

[9] Der Einfachheit halber wird angenommen, daß es sich bei der Zerlegung um eine äquidistante Zerlegung in einzelne Würfel handelt.

Ein Codefragment zur Bearbeitung von Komponenten ist in Beispiel 3.1.5 dargestellt. Dieses Beispiel schließt sich an die beiden Beispiele 3.1.1 und 3.1.2 an. In Anweisung (14) werden die Objekte Filter_v2 und Filter aus ihren jeweiligen Containern extrahiert (Variablen `$steuer_obj` und `$filter_obj`). In Anweisung (15) werden diese Variablen dazu benutzt, ein zusätzliches Attribut für das Steuerobjekt zu definieren, das eine Referenz auf das zugehörige Objekt im Container Modelle enthält. Anweisung (16) liefert die mengenwertige Komponente Gitterzerlegung aus dem Filter–Objekt zurück und Anweisung (17) trägt in das erste Element (Index 0) das zusätzliche Attribut *Gittergroesse* mit dem Wert 122.5 ein.

```
#
# ... (Fortsetzung von Beispiel 3.1.2, Seite 62)
#


my $steuer_obj = $steuer_cont->get_object('Filter_v2');            # (14)
my $filter_obj = $modell_cont->get_object('Filter');               #

$steuer_obj->insert_attribute(Modellobjekt=>$filter_obj);          # (15)

my $gitter_comp = $filter_obj->Gitterzerlegung();                  # (16)
$gitter_cmp[0]->insert_attribute(Gittergroesse=>122.5);            # (17)
```

Beispiel 3.1.5: Codebeispiel zur Erzeugung der Containerkomponenten aus Abbildung 3.8

Abschließend soll noch eine Möglichkeit beschrieben werden, wie aus den eigentlichen Daten relevante Informationen automatisch extrahiert werden können. Neben der Möglichkeit für das Workflowsystem relevante Informationen als Metainformationen in Form von Attribut–Wert Paaren abzulegen, besteht noch die Möglichkeit, statt einem Attributwert eine in *Perl* realisierte Methode oder Funktion zu definieren, die eine programmatische Extraktion der benötigten Information vornimmt. Dieser Mechanismus basiert, wie die Metainformationen, auf dem in Abschnitt 2.3.1 vorgestellten Basismechanismus der $\mathcal{W}*FLOW$ Attribute. Dazu muß zu einem Attribut eine Perlmethode definiert werden, welche die gewünschte Information zurückliefert. Die Methode wird beim Zugriff auf das Attribut mit festgelegten Parametern aufgerufen und das Ergebnis muß in Form eines Strings zurückgeliefert werden. Ein Methodenrumpf (engl. Stub) hat den in Beispiel 3.1.6 gezeigten Rahmen:

```
sub {
    my $self = shift;
    my $result;
    #
    # the spefific code follows here ...
    #
    return $result;
}
```

Beispiel 3.1.6: Methodenrumpf für eine Zugriffsmethode

Hierbei ist `$self` die jeweilige Container-, Containerobjekt- oder Komponenten-Instanz, die zur Laufzeit als Parameter jeder so definierten Routine übergeben wird. Der Zugriff auf die einzelnen Informationen erfolgt dann mittels der zur Verfügung gestellten API.

Die nachfolgende Routine in Beispiel 3.1.7 extrahiert, mittels regulärer Ausdrücke, aus dem

eigentlichen Datensatz eine Schlagwortliste[10]. Hierbei erfolgt eine Unterscheidung, ob der Datensatz direkt oder als URL–Referenz vorliegt. Im Falle, daß eine URL nicht aufgelöst werden kann, ist eine einfache Fehlerbehandlung (1) mittels des *Perl*-Operators die(...) realisiert. Als Parameter wird jeder so definierten Routine die jeweilige Instanz übergeben (Variable $self).

```
sub {
    my $self         = shift;
    my $content      = $self->content;
    my $type         = $self->type;
    my $content_type = $self->content_type;
    if ($type eq "REF") {
          #
          # Referenz aufloesen
          #
          use LWP::Simple;       # Einbinden der WWW Bibliothek
          $value = LWP::Simple::get $value;
          die "ERROR: URL kann nicht aufgeloest werden"
                                  if ! defined $value;             # (1)
    }
    $value =~ /\@keylist:(.*)/;
    return $1;
}
```

Beispiel 3.1.7: Beispielmethode zur Extraktion einer Schlüsselwortliste

Da es sich hierbei um eine Methode handelt, die an ein $\mathcal{W}*FLOW$–Attribut angehängt wurde, ist es völlig transparent, ob der Wert mittels einer Perlroutine berechnet wird, oder nur der gespeicherte Wert zurückgeliefert wird.

Im Rahmen der Anforderungen an eine Datenhaltungskomponente im wissenschaftlich–technischen Umfeld wurde auch die Problematik des Wiederauffindens und der Extraktion von Daten angesprochen. Durch die große Datenmenge ist es besonders wichtig, mittels geeigneter Anfragen relevante Daten extrahieren zu können. Hier gibt es in der Literatur eine Fülle von Ansätzen, die verschiedene Anfragestrategien in unterschiedlichen Datenbanken beschreiben. Ein Vertreter ist z. B. SQL [ANS86], das sich im Bereich relationaler Datenbanken durchgesetzt hat, sowie eine Reihe von Erweiterungsvorschlägen hierzu [Kul94, HG94, Bus90, KC93]. Weitere Anfragesprachen sind Query By Example (QBE) [DE89], deduktive Ansätze [RS87] sowie eine Vielzahl von proprietären Ansätzen in nichtrelationalen Datenbanksystemen. Ein Überblick über die verschiedenen Konzepte ist z. B. in [KRB85] und [LL95] gegeben.

Die Anforderungen, die im Rahmen von wissenschaftlich–technischen Datenbanken an eine Anfragesprache gestellt werden, lassen sich aus den Anforderungen aus Abschnitt 3.1.2 ableiten und finden sich auch in der Literatur (z. B. [GRS94b]) wieder. Die Hauptmerkmale hierbei sind:

1. Unterstützung komplexer Datentypen
2. Umgang mit häufigen Schemaänderungen
3. Unterstützung von Ad-Hoc Anfragen

[10] In dem unteren Codebeispiel wird angenommen, daß im Datensatz den eigentlichen Schlagwörtern die Kennung @keylist: vorangeht.

Die erste Anforderung wird oft durch domänenspezifische Anfragesprachen gelöst, die semantisch oberhalb der zuvor aufgezählten Sprachen liegen. Hierbei wird die Diskrepanz zwischen der Anfragesprache und den zu modellierenden Zusammenhängen gering gehalten, was jedoch auf Kosten der Allgemeingültigkeit geht. Der zweite Punkt erfordert die Integration von Metainformationen, da keine einheitlichen Datenstrukturen vorliegen und diese mittels Metainformationen zur Laufzeit erfragt werden müssen. Der letzte Punkt behandelt einen eher technischen Aspekt vieler objektorientierter Datenbanken. Sie erlauben zwar die Formulierung komplexer Suchanfragen, die dem Datenbanksystem aber in kompilierter Form (z. B. als Objektdatei) zugänglich sein müssen. Dies bringt zum einen Performanceprobleme mit, da gestellte Anfragen in einem ersten Schritt jeweils erst übersetzt werden müssen und zum anderen können Fehler in so formulierten Anfragen verheerende Folgen auf das Laufzeitsystem haben. Das Vorhandensein einer interpretativen Anfragekomponente vermeidet/vermindert die geschilderten Probleme und erlaubt die Unterstützung aller drei oben genannten Punkte.

Der Ansatz, der im Rahmen von $\mathcal{W}*FLOW$ gewählt wurde, zielt dahingehend, daß eine Reihe von domänenunabhängigen Basismechanismen zur Anfrageunterstützung (siehe nächster Absatz) bereitgestellt werden. Diese Basismechanismen erfüllen die oben gestellten Anforderungen nach der Unterstützung komplexer Datentypen, der Schemaunabhängigkeit sowie der Behandlung von Anfragen auf Ad-Hoc Basis.

Die Realisierung im Rahmen von $\mathcal{W}*FLOW$ geht dahin, daß eine Reihe von Zugriffsmethoden von der API bereitgestellt werden. Die Methoden nutzen zum Teil bestimmte, von *ObjectStore* aus angebotene Elemente der DML[11] [Obj96b], wodurch ein optimierter Zugriff auf die zugehörigen Datensätze ermöglicht wird. Die bereitgestellten Basismethoden erlauben beispielsweise die Suche von Datensätzen nach Attributen und Attributwerten (Suche in Metainformationen) und nach bestimmten Mustern (Inhalt). Da es sich um beliebige Inhalte handeln kann, besteht die Möglichkeit eigene Suchmuster zu definieren. Dies geschieht, indem sehr komplexe Bedingungen und Anfragen in *Perl* Notation formuliert werden können. Die Eignung von *Perl* für eine derartige Vorgehensweise wird von Tom Christiansen, dem Autor des Buches Perl Cookbook [CTW98] ausdrücklich in einem Interview [Ama99] betont. Die von ihm entwickelte *MoxPerl* Datenbank [Chr98] basiert vollkommen auf *Perl* und *Perl* wird auch als mächtige Anfragesprache eingesetzt.
Basierend auf den bereitgestellten Basismethoden und der Schnittstelle zu *Perl* können in konkreten Applikationen entsprechende Anfrageschnittstellen realisiert bzw. zur Verfügung gestellt werden.

[11] **D**ata **M**anipulation **L**anguage

3.2 Komponente Aktivität

Der vorliegende Abschnitt behandelt die Aktivitäten. Dies sind die Komponenten eines Workflowsystems, die zum einen die konkrete Funktion eines Workflows festlegen und zum andern auch ein wichtiges Hilfsmittel zur Strukturierung des Workflows darstellen.

In der Literatur findet sich keine einheitliche Definition des Begriffs der Aktivität (engl.: *activity*). Um ein Bild von den verschiedenen Interpretationen des Aktivitätsbegriffs zu vermitteln, sollen zunächst eine Reihe von Definitionen aus der Literatur zitiert und in Bezug auf Funktionalität und Einordnung innerhalb eines Gesamt-Workflows diskutiert werden.

3.2.1 Einführung in den Aktivitätenbegriff

Die Workflow Management Coalition (WFMC) definiert eine Aktivität wie folgt:

> **Definition 3.4 (Activity [WFM99])** *A description of a piece of work that forms one logical step within a process. An activity may be a manual activity which does not support computer automation, or a workflow (automated) activity. [...]. An activity is usually the smallest unit of work which is scheduled by a workflow engine during process enactment [...].*[12]

Daneben existiert bei der WFMC noch der Begriff des Aktivitätenblocks:

> **Definition 3.5 (Activity Block [WFM99])** *A set of activities within a process definition which share one or more common properties which cause the workflow management software to take certain actions with respect to the block in total. [...].*[13]

Die WFMC beschreibt eine Aktivität also als logischen Arbeitsschritt innerhalb eines Prozesses, der sowohl von Hand als auch mit Unterstützung eines Computersystems durchgeführt werden kann. Dabei wird eine Aktivität als die kleinste mögliche Arbeitseinheit angesehen, die nicht weiter unterteilt wird. Zusammengehörende Aktivitäten können weiterhin zu einem Aktivitätenblock zusammengefaßt werden, wodurch ein Mittel zur Hierarchisierung von Workflows zur Verfügung steht.

Die Definition einer Aktivität bei der Object Management Group lautet:

> **Definition 3.6 (Activity [OMG98])** WFActivity *is a step in a process that is associated, as part of an aggregation, with a single* WFProcess. *It represents a request for work in the context of the containing* WFProcess. *There may be many active* WFActivity *objects within a* WFProcess *at a given point of time.*[14]

[12] Übersetzung: *"Die Beschreibung eines Arbeitsstückes, das einen logischen Schritt innerhalb eines Prozesses darstellt. Eine Aktivität kann eine manuelle Aktivität, die keine Unterstützung durch den Computer vorsieht oder eine Workflow (automatisierte) Aktivität sein. [...]. Eine Aktivität ist gewöhnlich die kleinste Arbeitseinheit die durch ein Workflowsystem während der Prozessausführung vorgesehen ist."*

[13] Übersetzung: *"Eine Menge von Aktivitäten innerhalb einer Prozeßdefinition mit ein oder mehreren gemeinsamen Eigenschaften, welches das Workflow Management Programm dazu veranlassen, bestimmte Schritte hinsichtlich des Blocks als Ganzes auszuführen."*

[14] Übersetzung: *"Eine* WFActivity *ist ein Schritt innerhalb eines Prozesses, der als Teil einem einzelnen* WFProcess *zugeordnet ist. Diese repräsentiert eine Arbeitsanweisung im Kontext des übergeordneten* WFProcess. *Zu einem gegebenen Zeitpunkt kann es viele aktive* WFActivity*-Objekte innerhalb eines* WFProcesses *geben."*

Die OMG bezeichnet die Aktivität als einen Teil-Schritt im Rahmen genau eines Workflow-Prozesses. Sie repräsentiert eine konkrete Arbeitsanforderung im Rahmen des Workflow-Prozesses.

IBM's Workflow Produkt *FlowMark* ist zum Teil konform zur Spezifikation der WFMC und bietet folgende Interpretation der WFMC Definition:

> **Definition 3.7 (Activity [IBM96])** *An activity is a step within a process. It represents a piece of work that the assigned person can complete by starting a program or another process.*[15]
>
> *A FlowMark workflow model consists of the following types of activities:*
>
> **Process Activity:** *An activity to which a separate process is assigned. Starting this activity creates an instance of the referred process and starts it.*[16]
>
> **Program Activity:** *An activity to which a registered program is assigned. Starting this activity invokes the program.*[17]
>
> **Block Activity:** *A modelling construct that enables the grouping of related activities in a lower–level diagramm, and the modeling of loops.*[18]

Laut dieser Definition repräsentiert eine Aktivität einen Arbeitsschritt, der einem konkreten, für diesen Arbeitschritt zuständigen Bearbeiter zugeordnet ist. Es gibt folgende drei Ausprägungen für eine Aktivität:

Prozeßaktivität: Repräsentiert einen, eventuell aus mehreren Programmaktivitäten bestehenden, eigenen Workflow.

Programmaktivität: Repräsentiert den Umgang eines Benutzers mit einem konkreten Programm.

Blockaktivität: Strukturierungsmittel, das es erlaubt hierarchische Aktivitäten aufzubauen. Weiterhin ist es damit möglich, die enthaltenen Aktivitäten wiederholt auszuführen.

Den Abschluß sollen Definitionen bilden, wie sie im Rahmen des *NSF Workshops on Workflow and Process Automation in Information Systems* [NSF96] geprägt wurden. Diese sind insofern interessant, da viele der Teilnehmer dieses Workshops zu den international anerkannten Experten auf dem Gebiet des Workflow Management gezählt werden können[19].

> **Definition 3.8 (Activity [NSF96])** *A unit of work that an individual, a machine, or a group can perform in an uninterrupted span of time. The execution of one*

[15] Übersetzung: *"Eine Aktivität ist ein Schritt innerhalb eines Prozesses. Sie repräsentiert eine Aufgabe, die der zugeteilte Bearbeiter durch Ausführen eines Programms oder eines anderen Prozesses vollenden kann."*

[16] Übersetzung: *"Eine Aktivität der ein separater Prozeß zugeordnet ist. Durch Starten der Aktivität wird eine neue Instanz des zugeordneten Prozesses erzeugt und gestartet."*

[17] Übersetzung: *"Eine Aktivität, der ein registriertes Programm zugeordnet ist. Durch Starten der Aktivität wird das zugeordnete Programm aufgerufen."*

[18] Übersetzung: *"Ein Modellierungskonstrukt, das die Gruppierung von in Beziehung stehenden Aktivitäten und die Modellierung von Schleifen auf einer niedrigeren Ebene erlaubt."*

[19] z. B.: Gustavo Alonso (ETH Zentrum Zürch und IBM Almadon Research Center, CA), Dimitrios Georgakopoulus (GTE Labs, MA), Stef Joosten (Georgia State University, GA und University of Twente, Niederlande), Marek Rusinkiewicz (University of Houston, TX), Amit Sheth (LSDIS Lab, University of Georgia, GA), ...

activity consists of a sequence of interactions (called events) between the performer and the workflow management system, and a sequence of actions that change the state of a particular instance (or case).[20]

Definition 3.9 (Process [NSF96]) *A set of activities (also termed as tasks or steps) and precedence relations.*[21]

Die Teilnehmer des Workshops bezeichnen eine Aktivität als eine Arbeitseinheit, die von einer Person, Gruppe oder Maschine in einer unterbrechungsfreien Zeitspanne durchgeführt werden kann. Die Durchführung der Aktivität erfolgt hierbei in Form einer Reihe von Interaktionen zwischen Anwender(n)/Maschine und dem Workflowsystem und einer Reihe von Aktionen, welche den Zustand der Aktivität beeinflussen. Weiterhin wird ein Prozeß als Menge von Aktivitäten, zwischen denen Ablaufbeziehungen bestehen, beschrieben.

Die vorgestellten Definitionen beleuchten die verschiedenen Sichtweisen der Autoren. Bei den ersten beiden Definitionen handelt es sich um abstrakt gehaltene Beschreibungen von Standardisierungsgremien, die große Freiheitsgrade[22] bei der Umsetzung in ein konkretes System bieten. Das dritte Beispiel zeigt die Interpretation des Standards durch einen konkreten Hersteller, der mit dem von der WFMC verabschiedeten Standard konform geht. Die vierte Definition wurde im Rahmen eines Workshops geprägt, an dem eine Reihe von Experten teilnahmen.

Gemeinsam ist diesen Definitionen, daß eine Aktivität mit konkret durchzuführenden Arbeitsschritten verknüpft ist. Weiter haben alle Definitionen die Eigenschaft, daß Aktivitäten zu einer größeren Struktur zusammengefaßt werden können. Hierbei kann es sich um den Begriff Prozeß (WFMC, OMG, NSF–Workshop) oder weitere Strukturierungsmittel wie Aktivitätenblöcke (WFMC, *FlowMark*), etc. handeln.

Der Hauptunterschied zwischen den oben aufgeführten Definitionen liegt in der Wahl des Kriteriums, nach dem Arbeitsabfolgen in einzelne *Aktivitäten* unterteilt werden. Hier gibt es, durchaus begründbar, verschiedene Philosophien. Während die WFMC von einem "logischen Schritt" und der "kleinst möglichen Arbeitseinheit", die von einer Workflow-Engine ausgeführt werden kann, spricht, ist bei der OMG ganz einfach von einem "Arbeitsschritt im Rahmen eines Prozesses" die Rede. Beide Organisationen beziehen sich bei ihrer Sichtweise mehr auf die (organisatorische) Modellierung der Arbeit und der damit einhergehenden Zerlegung in kleinste Arbeitsschritte **unabhängig** von Randbedingungen, die ein konkretes Workflowsystem als Software-System ins Spiel bringen kann. Hersteller von Workflowsystemen werden hier schon spezifischer. So wird bei *FlowMark* der Begriff Programmaktivität mit einem Programm verknüpft, das bei der Durchführung der Aktivität benutzt wird.

Generell ist bei der Entscheidung, welche "kleinsten Arbeitsschritte" gewählt werden, die Frage nach den Ressourcen, die zur Durchführung der Arbeitsschritte benötigt werden, wichtig. Dies gilt umgekehrt auch für die Entscheidung, welche "kleinsten Arbeitsschritte" zu größeren

[20] Übersetzung: *"Eine Arbeitseinheit die von einem Individuum, einer Maschine oder einer Gruppe innerhalb einer unterbrechungsfreien Zeitspanne verrichtet werden kann. Die Ausführung einer Aktivität besteht aus einer Reihe von Interaktionen (Ereignissen) zwischen Ausführendem und dem Workflow-Management-System und einer Reihe von Aktionen welche den Zustand der einzelnen Instanz ändern."*

[21] Übersetzung: *"Eine Menge von Aktivitäten (auch als 'Aufgaben' oder 'Schritte' bezeichnet) und Ablaufbeziehungen"*

[22] Diese Freiheitsgrade rühren nicht von ungefähr her, haben doch die Workflowhersteller, die Mitglieder in den obigen Organisationen sind, ganz bestimmte Vorstellungen, die im allgemeinen mit deren eigenem Produkt übereinstimmen.

Einheiten (wie z. B. Aktivitätenblöcke oder Prozesse) zusammengefaßt werden sollen. Hierbei spielen gemeinsam genutzte Ressourcen oder auch Eigenschaften, die Arbeitsschritte gemeinsam haben, eine wichtige Rolle (vergleiche auch WFMC Blockdefinition).

Nach eigenen Erfahrungen und Beobachtungen bei der Modellierung von Workflows im wissenschaftlich–technischen Bereich [EDG+96, BDE+96, Qui98, Mei98] haben erstellte Workflows zumeist eine ähnliche Struktur. Es entstehen eine Reihe von Clustern aus einzelnen Arbeitsschritten, die sich durch komplexe Beziehungen innerhalb der Cluster auszeichnen, während zwischen den Clustern einfache Kontroll– und Datenflußbeziehungen vorherrschen. Dies liegt daran, daß ein Modellierer intuitiv dazu neigt, Arbeitsschritte, die mit denselben Daten und Ressourcen arbeiten, zur Vereinfachung der Synchronisation, zu Clustern mit **gemeinsamen** Eigenschaften und Ressourcen zusammenzufassen, während gleichzeitig versucht wird, die Schnittstelle zwischen den Clustern minimial zu halten. Bei der Betrachtung des Beispiels zur Fluidik-Simulation aus Abschnitt 1.2.3.1 (siehe auch Bild 3.11) sind Arbeitsschritte zu Clustern zur Bearbeitung der Simulations-Steuerungsdatei (Steuerdaten bearbeiten), zur Modellerstellung (Modelldaten bearbeiten), zur Simulation (Simulation) und zur Auswertung der Simulation (Auswertung) zusammengefaßt worden.

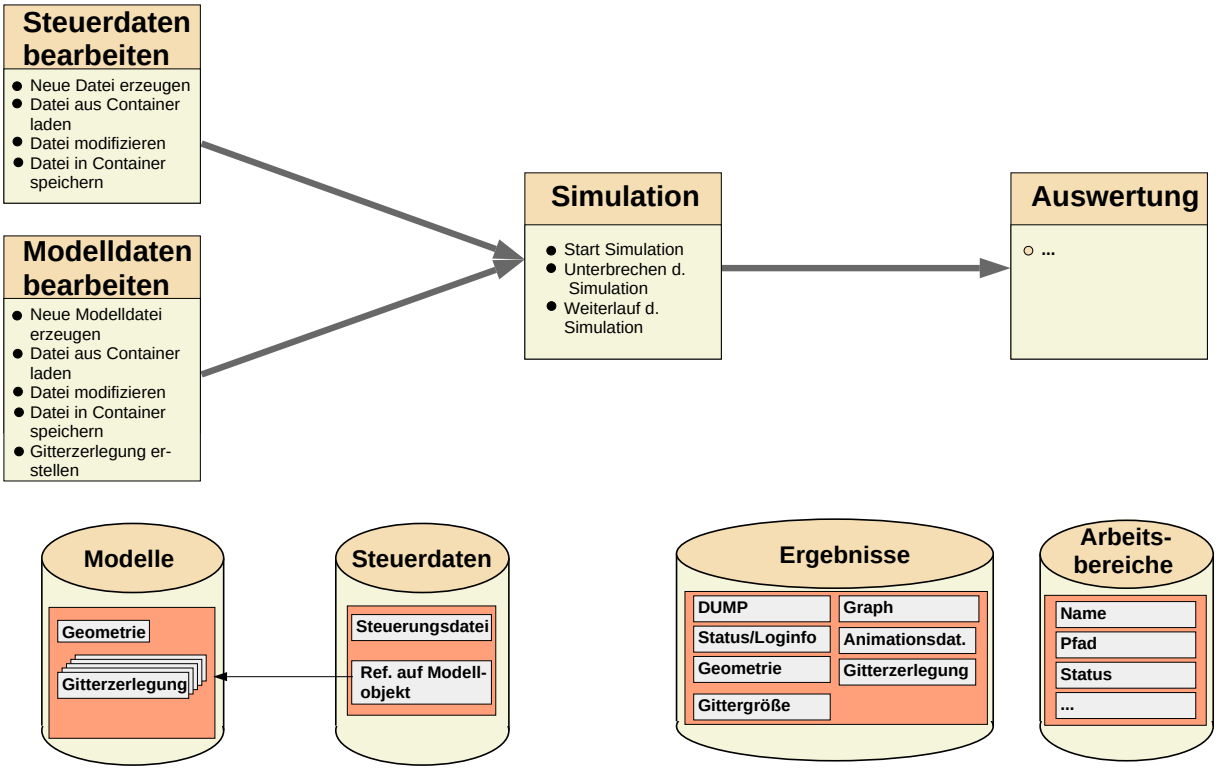


Abbildung 3.11: Beispiel CFX: *Arbeitsfluß und Datenorganisation*

So enthält das Cluster Modelldaten bearbeiten die folgenden Arbeitsschritte:

Neue Modelldatei erzeugen: Bei diesem Arbeitsschritt muß eine neue Datei in einem Arbeitsverzeichnis angelegt werden, die eventuell bereits ein bestimmtes Format aufweist. Weiterhin muß die neue Datei als aktuelle Modelldatei registriert werden.

Datei aus Container laden: Der Arbeitsschritt kopiert eine bereits bestehende Modelldatei, welche sich in einem Container befindet, in das aktuelle Arbeitsverzeichnis. Die Datei wird

dann als aktuelle Modelldatei registriert.

Modelldatei modifizieren: Der Schritt erfordert den Aufruf eines speziellen Editors zur Modifikation der Modelldatei.

Modelldatei in Container speichern: Die aktuelle Modelldatei wird in einem Containerobjekt abgespeichert.

Gitterzerlegung erstellen: Hierbei wird, ausgehend von der aktuellen Modelldatei, ein Werkzeug aufgerufen, das es dem Benutzer erlaubt eine neue Gitterzerlegung basierend auf dem aktuellen Modell zu erstellen.

Mit jedem der einzelnen obigen Punkte ist eine konkrete Aktion verknüpft, bei der es sich entweder um den Aufruf eines Programms handelt oder Aktionen im Rahmen der Workflow-Engine (z. B.: Eintragen von Datensätzen in Containerobjekte) durchgeführt werden.

Weiterhin wird deutlich, daß sich die Aktionen alle auf die selben Datenobjekten beziehen, in diesem Fall auf die Modelldatei.

Ein entscheidender Aspekt im Umgang mit den Modelldaten ist dabei, daß fast alle Arbeitsvorgänge nicht unmittelbar auf Containerobjekten, sondern auf temporären Arbeitsobjekten (hier Dateien in einem Arbeitsverzeichnis) durchgeführt werden. Nur einige wenige Aktionen (Datei aus Container laden und Datei in Container speichern) führen eine Synchronisation der Containerinhalte mit den temporären Datenobjekten, auf denen innerhalb des Clusters gearbeitet wird, durch.

Die Gruppierung von Aktionen, basierend auf der Benutzung gemeinsamer, privater Daten, stellt ein intuitives Konzept dar, das dem Konzept eines Objektes im Sinne einer objektorientierten Sprache entspricht. Bei der Objektorientierung werden eine Reihe von Operationen[23] auf die zugehörigen privaten Daten eines Objektes angewandt. Jedes Objekt hat einen internen Zustand, der sich durch Anwendung von Operationen auf das Objekt ändert. Diese objektorientierte Interpretation der Clusterung von Aktionen auf gekapselte, persönliche Daten innerhalb eines solchen Clusters führt zu der folgenden interessanten Interpretation des Aktivitätenbegriffs:

Eine Aktivität besitzt eine Menge von Aktionen (Arbeitsschritte) innerhalb eines Workflows, die auf den **gleichen** privaten Ressourcen (Daten) einer Instanz der Aktivität durchgeführt werden können. Eine Aktivität entspricht somit einer Klassendefinition. Sie definiert Aktionen und private Ressourcen für mögliche Objekte dieser Klasse, die Klasseninstanzen genannt werden. Im Rahmen der Abarbeitung eines Workflows werden *Aktivitäteninstanzen* erzeugt, welche dann die Ausführung der jeweiligen Arbeitsschritte, entsprechend den Vorgaben der Aktivität, regeln. Gemäß den Prämissen der Objektorientierung führt jede der Aktivitäteninstanzen Arbeitsschritte (Operationen) auf den privaten Ressourcen aus. Tabelle 3.2 zeigt die Entsprechungen zwischen den Begriffen aus dem Bereich der Objektorientierung und ihren Entsprechungen im Aktivitätenmodell.

Hiermit wird die einführende Diskussion des Begriffs Aktivität beendet. Im folgenden Abschnitt sollen einige weitere Randbedingungen an den Aktivitätenbegriff erarbeitet werden, bevor dann anschließend die im Rahmen dieser Dissertation erarbeitete Semantik des Aktivitätenbegriffs im Detail vorgestellt wird.

[23] zumeist Methoden genannt

Tabelle 3.2: *Struktur eines Komponentenobjekts*

OO	Aktivitätenmodell	Erläuterung
Klasse	Aktivität	Definition von Verhalten und Struktur einer Menge von Instanzen.
Instanz (Objekt)	Aktivitäteninstanz	Konkrete Ausprägung einer Klasse mit innerem Zustand, der durch die Daten des Objekts repräsentiert wird.
Operation (Methode)	Arbeitsschritt	Operationen werden auf den lokalen Daten einer Instanz durchgeführt.

3.2.2 Randbedingungen

Wenn auch jede Instanz einer Aktivität instanzeigene Datenressourcen verwendet, so gibt es dennoch in der Regel bestimmte, innerhalb der Aktivität definierte Aktionen, die auf gleiche globale Datenressourcen eines Workflows zugreifen müssen. Dies führt im allgemeinen zu Synchronisationsproblemen, die vom System gelöst werden müssen. Um zu verhindern, daß mehrere Instanzen auf die gleichen Datensätze zugreifen, müssen Mechanismen, wie sie aus dem Bereich der Datenbanksysteme bekannt sind, eingesetzt werden. Hier kommen Mechanismen wie unterschiedliche Sperren, Versionierung, Checkin/–out und Transaktionen [LS87, LL95] zum Einsatz.

Transaktionen wurden bereits im Rahmen der $\mathcal{W}*FLOW$–Container in Abschnitt 3.1.2 eingesetzt. Sie stammen aus dem Bereich der Datenbanktechnologie und sichern dort die Konsistenz der Daten bei paralleler Bearbeitung verschiedener Benutzeraufträge (*Concurrency Control*) sowie der automatischen Behandlung von Fehlern in Ausnahmesituationen (*Recovery*). Diese Merkmale von Datenbank-Transaktionen machen auch ihren Einsatz im Rahmen von Workflowsystemen interessant, sind doch der Mehrbenutzerbetrieb sowie Fehlerbehandlung wichtige Anforderungen an ein Workflow-Management-System. Es zeigt sich jedoch, daß der Einsatz von Transaktionen im Workflowbereich ganz anderen Anforderungen unterliegt, als im klassischen Datenbankbereich. So sind Datenbank-Transaktionen von zumeist kurzer Dauer (im Sekundenbereich) [Lie98], besitzen eine "flache" Struktur, die in der Regel aus wenigen, sequentiell auszuführenden Operationen auf die Datenbasis besteht, und ihr Verhalten im Fehlerfalle ist durch fest vorgegebene technische Korrektheitskriterien, auf Basis der Konflikt-Serialisierbarkeit (Tabelle 3.3), festgelegt. Die Atomizität einer Transaktion besagt, daß entweder alle Operationen im Rahmen einer Transaktion ausgeführt werden müssen, oder keine der Operationen Änderungen am Datenbestand hinterlassen darf (*"all or nothing"*–Eigenschaft). Weiterhin garantiert eine Datenbank-Transaktion, daß die erzielten Ergebnisse erst nach erfolgreichem Abschluß einer Transaktion (Commit) nach außen sichtbar sein dürfen (Isolation). Die Grundidee bei Datenbank-Transaktionen liegt somit in der Konsistenzsicherung der Daten in der Datenbank.
Im Gegensatz dazu ist der Grundgedanke bei Transaktionen im Workflowbereich die Sicherung der Konsistenz des Geschäftsprozesses. Da Workflow-Transaktionen im Rahmen von Workflow-Instanzen stattfinden, werden die Anforderungen durch den zugrundeliegenden Geschäftsprozeß definiert. Hierbei handelt es sich um langandauerende Prozesse, beispielsweise um einen komplexen CAD–Entwurf, die sich über Monate erstrecken können. Im Rahmen solcher Workflows spielen aber nicht nur die Daten eine zentrale Rolle, sondern auch die dabei beteiligten Anwender und Programme, die innerhalb einer verteilten, heterogenen Umgebung in den Workflow integriert sind. Ein Workflow–Transaktionskonzept muß beschreiben und sicherstellen, **wer** mit

welchen Werkzeugen, **wie** und zu welchem Zeitpunkt (**wann**) auf Daten zugreifen darf. Da hierbei der Mensch unmittelbar als Ressource beteiligt ist, ergeben sich völlig neue Anforderungen an die Transaktionssemantik. So kann ein Mensch durch Krankheit, Arbeitsplatzwechsel und dergleichen als Ressource ausfallen, oder er kann bei der Abarbeitung eines Arbeitsvorgangs Fehler machen, die ein Zurücksetzen einer Transaktion notwendig machen.

Durch den verteilten Charakter einer Workflow Anwendung ist es weiterhin oft nicht ausreichend, eine Transaktion über Operationen auf einer einzelnen lokalen Datenbasis zu spannen, sondern es ist teilweise erforderlich, eine Transaktion über mehrere verschiedene Datenbasen zu modellieren.

Um Probleme wie die hier geschilderten zu lösen, wurden zu Beginn der neunziger Jahre – unabhängig vom Einsatzgebiet Workflow Technologie– eine Reihe von Erweiterungen an den ursprünglichen Transaktionsmodellen durchgeführt (sogenannte *Erweiterte Transaktionsmodelle*). Die Erweiterungen erlauben beispielsweise das Schachteln von einzelnen Transaktionen (*Nested Transactions*), die die Modellierung von komplexeren Szenarien, wie etwa von hierarchischen Workflows, erlauben. Weitere Erweiterungen versuchen die strenge Isolationseigenschaft klassischer Transaktionen aufzuheben, indem sie es erlauben, Teilergebnisse bereits vor erfolgreichem Ende einer Transaktion zur Verfügung zu stellen. Dies wird beispielsweise vom *SAGA*–Modell [GMS87] realisiert, das geschachtelte Transaktionen vorsieht, wobei die Ergebnisse der einzelnen Transaktionen nicht erst nach erfolgreicher Beendigung der Top–Level Transaktion freigegeben, sondern nach erfolgreicher Beendigung jeder einzelnen Transaktion sichtbar gemacht werden. Dadurch wird jedoch auch die Gefahr erhöht, daß Transaktionen, welche die frühzeitig freigegebenen Ergebnisse nutzen, bei Abbruch der Top-Level Transaktion ebenfalls zurückgesetzt werden müssen. Eine wichtige Rolle spielen in diesem Fall sogenannte Kompensationsaktivitäten, die im Falle eines Abbruchs einer atomaren Transaktion durchgeführt werden müssen, um bereits getätigte Änderungen wieder rückgängig zu machen.

Komplexe Geschäftsmodelle erfordern zudem mächtigere Kontrollflußkonstrukte als die lineare Abarbeitung von Operationen. So integriert das *ConTract*–Modell [WR91], neben den klassischen Transaktionen, noch eine explizite Ablaufsteuerung mittels Skripten. Bewirkt bei klassischen Transaktionen das Scheitern einer Operation das komplette Zurücksetzen aller bisher durchgeführten Operationen, so ist dies beim Einsatz im Workflowbereich nicht möglich, da hierbei möglicherweise die Arbeit von vielen Monaten verworfen werden müßte. Vielmehr sind hier Mechanismen gefragt, die ein kontrolliertes und partielles Zurücksetzen, möglicherweise auch unter Einbeziehung eines oder mehrerer Anwender, vorsehen. *ConTracts* besitzt hier Ansätze, die die explizite Modellierung von Fehlersituationen erlaubt.
Der Korrektheitsansatz auf Basis der Konfliktserialisierbarkeit, wie er im klassischen Transaktionsmodell eingesetzt wird, d. h. eine Unterscheidung in Lese- und Schreiboperationen mit zugehöriger Kompatibilitätsmatrix (Tabelle 3.3) zeigt sich in komplexen Modellen oft als zu starr und rigide. Um ein höheres Maß an Nebenläufigkeit zu erreichen, existieren Ansätze, die Semantik der Operationen mit ins Spiel zu bringen [RC97]. In diesem Fall wird die Nebenläufigkeit verschiedener Operationen an deren Verträglichkeit zueinander bestimmt.

Tabelle 3.3: *Kompatibilitätsmatrix bei Lese– und Schreiboperationen*

	Lesen	Schreiben
Lesen	x	-
Schreiben	-	-

Die unterschiedlichen Anforderungen an Transaktionen zwischen dem klassischen Einsatzgebiet Datenbanken und dem Bereich Workflow machen deutlich, daß klassische Transaktionen nicht zur Unterstützung von Geschäftsprozessen eingesetzt werden können, sondern verschiedene Erweiterungen und Änderungen vorgenommen werden müssen. So gibt es, wie im Text bereits angesprochen, verschiedene Erweiterungen, die sich speziell mit einzelnen oder mehreren der oben angesprochenen Probleme befassen. Der Einsatz solcher *Erweiterten Transaktionen* im Rahmen von Workflowanwendungen wurde u. a. in [AAEA+96] und [Lie98] untersucht. Ein Kritikpunkt dieser Transaktionsmodelle liegt darin, daß sie sich zu sehr an der Datensemantik orientieren und nicht auf die Anwendungssemantik eingehen, was eine Reihe von Einschränkungen mit sich bringt. Die aktuellen Forschungen auf dem Gebiet der Workflow–Transaktionen umfassen zwei Richtungen. Zum einen existieren Ansätze, die aufbauend auf existierenden *Erweiterten Transaktionsmodellen* versuchen, Workflowsysteme zu realisieren [SR93, Vos97, She94, BDS+93] bzw. der umgekehrte Ansatz, der ausgehend von einem Workflowmodell, transaktionale Eigenschaften zu integrieren versucht [Lie98]. Forderungen, wie sie von beiden Ansätzen formuliert werden, umfassen im einzelnen die folgende Punkte:

Sicherung der Konsistenz des Geschäftsprozesses: Im Gegensatz zu klassischen Transaktionen, bei denen technische Korrektheitskriterien die Konsistenz der Daten sichern und fest vorgegeben sind, soll die Konsistenz auf Grundlage semantischen Korrektheitskriterien, wie sie vom Geschäftsprozeß abgeleitet werden können, realisiert werden. Das bedeutet, daß es keine fest vorgegebenen Kriterien in Bezug auf die Serialisierbarkeit der Abläufe gibt, sondern die Serialisierbarkeit wird aus dem Geschäftsprozeß abgeleitet.

Der Hauptunterschied zwischen beiden Ansätzen liegt darin, daß bei klassischen Datenbank-Transaktionen die Konsistenz an die Daten geknüpft ist, während sie im Falle der Workflow-Anwendung an das weitaus abstraktere Ziel des Erreichens eines "gültigen Endzustandes" [Lie98] angelehnt ist.

Systemgesteuerte Isolation: Die Freigabe von Teilergebnissen soll frei definierbar sein.

Kontroll- und Datenfluß: Der Kontroll- und Datenfluß soll vom Benutzer bzw. der Anwendung definierbar sein.

Fehlerhandhabung: Es sollen definierbare Fehler- und Ausnahmesituationen modellierbar sein.

Transaktionstopologie: Die Transaktionstopologie muß sich strukturell an den zu modellierenden Workflow anpassen lassen.

Abhängigkeiten zwischen Transaktionen: Im Gegensatz zu Datenbank–Transaktionen, die hauptsächlich konkurrierend agieren, werden Workflow-Transaktionen auch kooperierend eingesetzt, da im Rahmen eines Workflows ein gemeinsames Ziel erreicht werden soll. Dies erfordert zusätzliche, intertransaktionale Kontrollflußkonstrukte.

Atomizität: Die *"All or Nothing"*–Eigenschaft klassischer Datenbank-Transaktionen muß zugunsten einer benutzerdefinierbaren Atomizität aufgeweicht werden.

Persistenz: Die Persistenz umfaßt neben den Daten auch den Prozeßkontext.

Nach diesen Ausführungen wird klar, daß die transaktionale Unterstützung von Workflowsystemen ein äußerst komplizierter Themenbereich ist, die aber eine enorme Relevanz, nämlich der Zusicherung der Korrektheit der Abarbeitung von Workflows, hat. Transaktionale Unterstützung

von Workflowsystemen ist zur Zeit ein hochaktueller Forschungsbereich und kein kommerziell verfügbares System verfügt momentan über derartige Mechanismen [AAEA+96, Lie98]. Aus diesem Grund soll im Rahmen von $\mathcal{W}*FLOW$ kein eigenes Transaktionsmodell für alle Workflowanwendungen vorgestellt werden, sondern es sollen Basismechanismen zur Realisierung einer auf der Semantik einer bestimmten Anwendung abgestimmten Transaktionslösung zur Verfügung gestellt werden.

Kernkomponente jedes Workflow–Transaktionsmodells ist die Aktivität. Vom Moment der Erzeugung einer Aktivitäteninstanz bis zu ihrer Beendigung und Zerstörung durchläuft eine Aktivitäteninstanz eine Reihe von Zuständen wie dies in Abbildung 3.12 verdeutlicht wird.

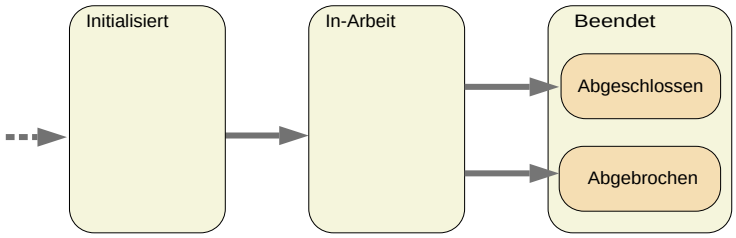


Abbildung 3.12: *Zustände bei Workflowsystemen. Die Pfeile definieren Zustandsübergänge zwischen den einzelnen Zuständen der Aktivitäteninstanz.*

Zunächst muß die Aktivitäteninstanz erzeugt und initialisiert werden (Übergang in den Zustand Initialisiert). Anschließend geht die Instanz in den Zustand In-Arbeit über. In diesem Zustand werden die notwendigen, im Rahmen der Aktivität definierten Arbeitsschritte ausgeführt. Nach Vollendung der Arbeitsschritte muß die Aktivitäteninstanz beendet werden. Je nach Situation kann dabei eine Aktivitäteninstanz mit positivem Ergebnis (Abgeschlossen) oder negativem Ergebnis (Abgebrochen) beendet werden.

Dieser dreiteilige, grobe Ablauf einer Aktivitäteninstanz ist in fast allen Workflowsystemen zu finden, bzw. die Kernfunktionalität kann durch diese Zustände repräsentiert werden [BMWCJ98]. Hierbei ist es unwesentlich, ob die konkreten Systeme explizit mit Zuständen realisiert sind oder ob dem System andere Realisierungskonzepte zugrunde liegen.

Speziell die Zustandsübergänge in die Zustände Initialisiert und Beendet (Verfeinerungen Abgeschlossen und Abgebrochen) liefern einen ersten Ansatzpunkt für ein Workflow–Transaktionskonzept. Der oben beschriebene, dreiteilige Ablauf einer Aktivitäteninstanz kann aus Transaktionssicht als Ressourcenallokation (benötigte Daten, Ressourcen und Werkzeuge), Bearbeitung und Ressourcenfreigabe gedeutet werden (siehe Abbildung 3.13).

Zu Beginn einer Aktivitäteninstanz werden benötigte Datenressourcen ermittelt, aus zentralen Datenrepositories in temporäre Arbeitsbereiche kopiert, zusätzliche Ressourcen, wie benötigte Werkzeuge, auf Vorhandensein/Verfügbarkeit geprüft und dem gesamten Arbeitsvorgang ein Bearbeiter zugeordnet. Nachdem der Bearbeiter (eventuell verschiedene) Arbeitsschritte im Rahmen der Aktivitäteninstanz durchgeführt hat, müssen die Ergebnisse seiner Arbeit (wieder) in zentralen Datenrepositories gesichert werden und eine Freigabe der reservierten Ressourcen erfolgen.

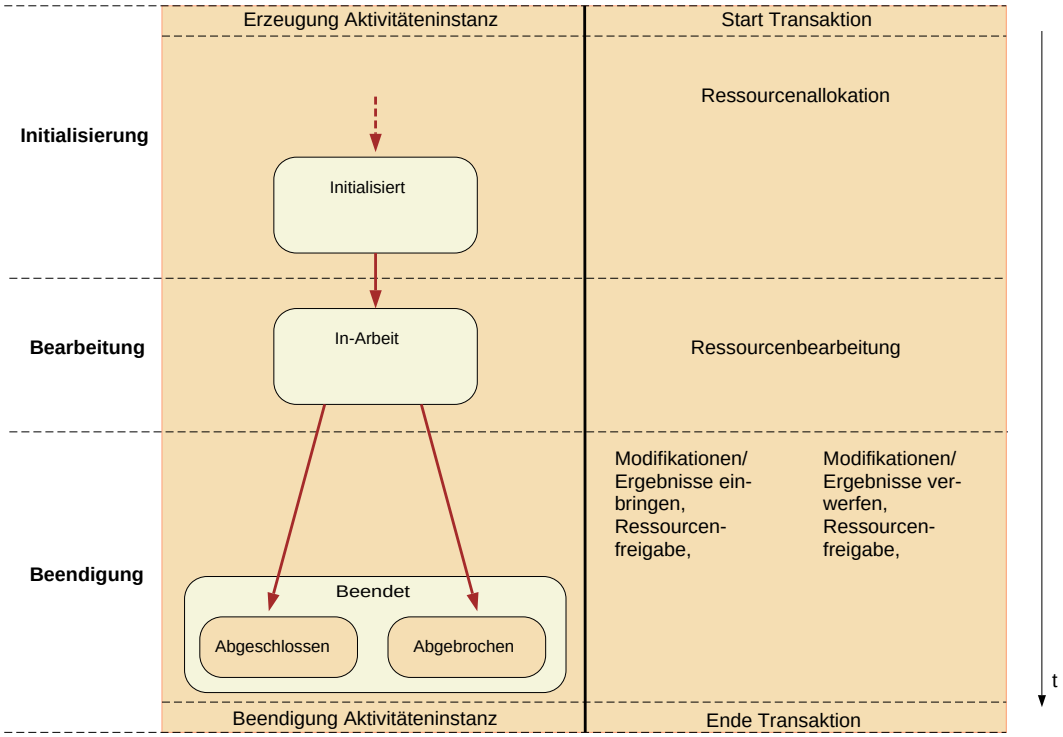


Abbildung 3.13: *Phasen einer Aktivität aus transaktionaler Sicht*

Anforderung 1:

Die Aktivität stellt das Basiskonzept zum Aufbau transaktionsunterstützter Workflows bereit. Eine Aktivitäteninstanz durchläuft während ihrer Existenz die drei Phasen Ressourcenallokation, Ressourcenbearbeitung und Ressourcenfreigabe, welche zur Modellierung von Transaktionen herangezogen werden können.

Dieses Aktivitätenkonzept bietet alle Hilfsmittel, um eine transaktionale Unterstützung durch Sperren und Freigeben von Ressourcen, unter Einbeziehung der konkreten Anwendungssemantik, zu Beginn und Ende einer Aktivitäteninstanz zu realisieren. Durch Verfeinerung des Zustandes In-Arbeit können verzweigte, aus mehreren einzelnen Arbeitsschritten bestehende, Ablaufszenarien realisiert werden, die unter anderem auch eine Teil–Resynchronisation von temporären Arbeitsergebnissen mit zentralen Datenrepositories vornehmen können. Die im Rahmen einer Aktivitäteninstanz durchzuführenden Aktionen werden hierbei als Zustandsübergänge zwischen den definierten Zuständen einer Aktivität modelliert. Zusätzlich können noch Vor– und Nachbedingungen mit den Übergängen verknüpft werden. So läßt sich beispielsweise situationsbezogen entscheiden, ob ein Zustandsübergang überhaupt stattfinden kann, d. h. eine Aktion durchgeführt, bzw. die damit verbundene Aktion erfolgreich durchgeführt worden ist. Dies erlaubt sowohl auf einfache Art und Weise die Modellierung komplexer kausaler Abhängigkeiten und Ablaufsteuerungen als auch die Realisierung von um Kontrollflußelemente erweiterten Transaktionsmechanismen, wie sie beispielsweise in *ConTracts* existieren.

Anforderung 2:

Zur Definition der Semantik einer Aktivität bzw. Transaktionslogik kann der von der Aktivität vorgegebene Zustandsautomat verfeinert werden. Die Aktionen, welche im Rahmen einer Aktivitäteninstanz zu tätigen sind, werden mit den Zustandsübergängen des endlichen Automaten verknüpft. Zusätzlich können zu den Übergängen noch Vor– und Nachbedingungen formuliert werden.

Wurde bislang die Vorstellung des Transaktionsbegriffs auf eine Instanz einer Aktivität eingeschränkt, so steigt die Komplexität an, wenn Instanzen mehrerer Aktivitäten betrachtet werden. Zwischen unterschiedlichen Aktivitäten eines (oder mehrerer) Workflows bestehen häufig Abhängigkeiten der Art, daß ihre Ausführungsreihenfolge kausal voneinander abhängen. So kann die Auswahl einer Aktivität B eventuell erst dann erfolgen, wenn die Ergebnisse von Aktivität A vorliegen. In diesem Fall wird von einer sequentiellen Ausführungsreihenfolge gesprochen (erst A, dann B; $A \rightarrow B$). Diese Aufgaben eines Workflowsystems werden als *verhaltensbezogene Aspekte* bezeichnet (Abschnitt 2.1.1). Sie beschäftigen sich mit Fragen über die Abarbeitungsrandbedingungen von Aktivitäten in Workflowsystemen, wie Reihenfolge, Bedingung für die Ausführung, Zulässigkeit von Auslassung und Wiederholung, Nebenläufigkeit, alternatives Vorgehen, Entscheidung über den nächsten Arbeitsschritt, Reaktion auf Ereignisse, etc. Für ihre Beschreibung gibt es eine Reihe mathematischer Modellierungsansätze, die über die Beschreibung der kausalen Strukturen durch Kontroll– und Datenflüsse, z. B. durch State–/Activitycharts [HPSS87, Har88] oder erweiterte Petrinetze [DG94, OS96], bis zur Beschreibung von Kommunikations– und Aktionssemantik über formale Spezifikationssprachen, aus dem Bereich der Kommunikationstheorie [MMWFF92, WF87] oder Koordinationstheorie [MC94], reichen.

Workflow Hersteller und Experten im Bereich *Business–Engineering* haben solche Ansätze und Beschreibungsmethoden mit Methoden zur Modellierung von Organisationsstrukturen für Arbeitsplanungssysteme vermischt. Dies hat jedoch den Nachteil, daß die hieraus resultierenden Beschreibungen der verhaltensbezogenen Aspekte nicht mehr vorgehensneutral sind. Vielmehr schreibt der gewählte Modellierungsansatz eine bestimmte Vorgehensweise fest.

Da es sich bei $\mathcal{W}*FLOW$ um einen Baukasten zum Aufbau von Workflowsystemen handelt, der die Bereitstellung *verfahrensneutraler* Komponenten zum Ziel hat, scheiden gerade solche "höherwertigen" Modellierungskonzepte aus. Vielmehr sind hier Basismechanismen gefordert, auf Grundlage derer solche "höherwertigen" Beschreibungsansätze umzusetzen sind. Die $\mathcal{W}*FLOW$ zugrundeliegende Philosophie ist dabei, daß verhaltensbezogene Aspekte eines Workflows letztendlich über Bedingungen und Beziehungen bezüglich der Daten, Metadaten und Zustandsinformationen des Workflowsystems sowie externen Informationsquellen beschrieben werden müssen. Der im Rahmen einer Aktivitäteninstanz vorgestellte Ansatz, die durchzuführenden Aktionen mit Vor– und Nachbedingungen zu verknüpfen und auf diese Art und Weise zu bestimmen, ob ein Zustandsübergang innerhalb einer Aktivitäteninstanz möglich ist bzw. ein neuer Zustand angenommen werden kann, eignet sich auch für die Modellierung *verhaltensbezogener Aspekte* zwischen unterschiedlichen Aktivitäteninstanzen. So läßt sich beispielsweise die Sequentialisierung von Aktivitäteninstanzen erzwingen, indem die Initialisierung einer Aktivitäteninstanz vom Zustand einer zuvor abzulaufenden Aktivitäteninstanz abhängig gemacht wird.

Anforderung 3:

Für die Modellierung der *verhaltensbezogenen Aspekte* eines Workflows sollen in $\mathcal{W}$*FLOW* Bedingungen auf den Objekten eines Workflows, den Daten, Metadaten, Zuständen, etc., eingesetzt werden können.

Damit soll die Diskussion der Anforderungen an den Transaktionsbegriff bei Workflowsystemen beendet und im folgenden auf den Aspekt der Toolanbindung eingegangen werden.

Häufig ist mit der Durchführung eines Arbeitsschrittes im Rahmen der Abarbeitung einer Aktivitäteninstanz der Aufruf eines externen Softwaretools verbunden. Ein Workflow-Management-System muß in diesem Rahmen gewährleisten, daß das entsprechende Werkzeug mit den richtigen Eingangsdaten versorgt und dann das Werkzeug mit geeigneten Parametern aufgerufen wird. Der Bearbeiter einer Aktivität arbeitet dann in der Regel mit den Softwaretools unabhängig vom WFMS, bis zu dem Zeitpunkt, an dem er seine Arbeitsergebnisse im lokalen Dateisystem gesichert hat. Das WFMS muß dem Bearbeiter dann gegebenenfalls die Möglichkeit bieten, die Ergebnisse vom lokalen Dateisystem zurück in das zentrale vom WFMS verwaltete Datenmanagement-System zu bringen. Diese Form der externen Anbindung nennt man *Blackbox*–Integration. Für eine Reihe von Anwendungen reicht eine solche einfache *Blackbox*–Integration aber nicht aus. So gibt es Softwarewerkzeuge, z. B. Simulatoren (siehe dazu auch Abschnitt 1.2.3.1), die bei ihrem Aufruf eine lokale Arbeitsverzeichnis-Struktur erwarten, in dem sich die für den Aufruf relevanten Daten, beispielsweise *Modell*–, *Steuerungsdatei*, zusätzliche Fortran–Quellcodedateien und Optionseinstellungen an bestimmten, festgelegten Stellen mit z. T. festgelegten Dateinamen, in dem Verzeichnisbaum befinden. Bevor ein Tool in diesem Falle gestartet werden kann, muß zuerst ein entsprechendes Arbeitsverzeichnis angelegt, die notwendigen Daten aus den zentralen Container–Repositories in das Verzeichnis kopiert und gegebenfalls aufbereitet werden. Workflowsysteme lösen die Problematik durch eine Programmschnittstelle, mit deren Hilfe der eigentliche Programmaufruf in ein weiteres Programm einbettet wird, das die zusätzlichen Aufgaben vor dem eigentlichen Programmstart erledigt. Zur Programmierung solcher *Wrapper*–Programme eignen sich insbesondere Skriptsprachen, die eine administrationsnahe Programmierung erlauben. Eine noch stärkere Integration von externen Programmen mit dem WFMS ist möglich, wenn das externe Programm oder *Wrapper*–Programm über eine API Durchgriff auf das WFMS besitzt. In dem Fall kann es beispielsweise den Status einer Aktivität, über die simplen Zustände Tool_xyz-gestartet oder Tool_xyz-beendet hinaus, genau beschreiben (z. B. detaillierte Angabe des aktuellen Status einer langandauernden Simulation) oder Ereignisse im WFMS triggern (Ereignisse innerhalb $\mathcal{W}$*FLOW* werden weiter unten erörtert). Hierbei handelt es sich um eine *Whitebox*–Integration. Der Aktivitätenbegriff eines WFMS sollte all die unterschiedlichen Integrationsformen ermöglichen. Eine detaillierte Betrachtung der unterschiedlichen Anbindungsarten befindet sich in Abschnitt 4.1. Da die konkrete Einbettung von Programmen in spezielle Betriebssysteme sehr stark plattformabhängig ist, wird im Rahmen von $\mathcal{W}$*FLOW* die konkrete Programmbeschreibung in eine eigenständige Komponente, dem *Toolserver* (Kapitel 4), ausgelagert. Das hier vorgestellte Aktivitätenkonzept stellt aber einen abstrakten, generischen (d. h. nicht von der Laufzeitumgebung abhängigen) Mechanismus bereit, mit dem innerhalb von Aktivitätendefinitionen der Aufruf externer Tools im Rahmen einer Aktion beschrieben werden kann. Diese Beschreibung beinhaltet auch, welche Ressourcen (Daten, Zustandsvariable, etc.) für eine Aktion benötigt werden (siehe nachfolgenden Abschnitt 3.2.3).

Anforderung 4:

Die im Rahmen einer Aktivitäteninstanz auszuführenden Aktionen umfassen unter anderem den Start komplexer, externer Anwendungen. Aktivitäten sollten dazu eine umgebungsunabhängige Schnittstelle mit unterschiedlichen Integrationsarten zur Anbindung externer Programme bereitstellen.

Der Aufruf von externen Programmen ist aber nicht die einzige Interpretation, die der Begriff der Aktion innerhalb einer Workflow Aktivität haben kann. Häufig wird hiermit auch ein computergestützter Ablauf assoziiert, der nur eine interne Veränderung im Workflow auslöst. Typische Beispiele sind hier eine Aktion, die die Erzeugung eines neuen Subworkflows auslöst, ein Datenelement löscht oder einfache Parameter abfragt. Dabei sollte auch erlaubt sein, daß interne Aktionen vollkommen ohne Interaktion mit dem Benutzer stattfinden können. Das Eintreffen eines Dokumentes als Arbeitsergebnis einer Aktivitäteninstanz könnte z. B. eine interne Aktion aufrufen, die das erstellte Dokument automatisch an einen festgelegten Verteiler (ohne explizite Aktion durch einen Benutzer) weiterleitet. Auch derartige interne Aktionen sollte eine Aktivitätenrealisierung von $\mathcal{W}*FLOW$ ermöglichen. Die Realisierung solcher Aktionen erfolgt durch Programmierung mittels der von $\mathcal{W}*FLOW$ zur Verfügung gestellten API.

Anforderung 5:

Neben der Anbindung externer Programme ist auch die Anbindung workflow-interner Aktionen, mittels der $\mathcal{W}*FLOW$–API, sinnvoll.

Das nächste Beispiel bringt einen weiteren interesanten Aspekt von WFMS in Bezug auf die Ausführung von Aktionen ins Spiel: das Triggern von Aktionen. Hierunter wird ein genereller Mechanismus verstanden, der es erlaubt, daß beispielsweise die Beendigung einer Aktion bzw. einer Aktivitäteninstanz dazu führt, daß innerhalb einer anderen Aktivitäteninstanz eine Aktion ausgelöst wird. Die Anwendung einer solchen Aktivitäten–Aktivitäten Kommunikation sind vielfältig: z. B. automatische Benachrichtigung von Bearbeitern bei Abschluß einer bestimmten Aktion, automatische Aktivierung von Folgeaktivitäten, falls alle notwendigen Eingabedaten vorliegen, etc.
Im Rahmen der $\mathcal{W}*FLOW$–API werden solche Kommunikationsmechanismen bereits innerhalb der in Abschnitt 2.3.2 beschriebenen Basisklassen für alle Module realisiert und stehen somit komponentenübergreifend zur Verfügung, was beispielsweise auch eine Kommunikation zwischen Containern und Aktivitäten oder Aktivitäten und externen Anwendungen ermöglicht.

Anforderung 6:

Basierend auf einem allgemeinen, komponentenbasierten Kommunikationsmechanismus soll eine einfache asynchrone Kommunikation zwischen den Aktivitäteninstanzen realisiert werden.

Nachdem nun die wesentlichen Randbedingungen erläutert wurden, die mit dem Aktivitätenbegriff in Zusammenhang stehen, sollen nun im folgenden Abschnitt die $\mathcal{W}*FLOW$–Aktivitäten, wie sie hier im Rahmen der Arbeit konzipiert und realisiert wurden, näher beschrieben werden.

3.2.3 $\mathcal{W}$*FLOW*–Aktivität

Ausgehend von den Anforderungen und Erläuterungen aus dem vorherigen Abschnitten wird ein neuer *erweiterter* Aktivitätenbegriff wie folgt eingeführt:

> **Definition 3.10 ($\mathcal{W}$*FLOW*–Aktivität)** *Eine $\mathcal{W}$*FLOW–Aktivität ist eine Beschreibung des Aufbaus und des Verhaltens von Aktivitäteninstanzen, die von einem Workflow-Management-System bei Aktivierung einer Aktivität dynamisch erzeugt werden. Sie besteht aus:*
>
> 1. *einer Beschreibung der Ein– und Ausgabegrößen und internen Variablen einer Aktivitäteninstanz und ihrer Zuordnung zu externen Datenressourcen (Containern).*
> 2. *einer Reihe von Aktionsbeschreibungen (Aktionshandler), die auf den Ein– und Ausgangsgrößen sowie internen Variablen arbeiten, externe Tools aufrufen oder WFMS-interne Schnittstellen bedienen.*
> 3. *der Spezifikation eines erweiterten Zustandsautomaten, der festlegt, welche Zustände eine Aktivitäteninstanz einnehmen kann und welche Zustandssübergänge während der Lebensdauer der Aktivitätsinstanz wann, wie und mit welchem Ergebnis durchgeführt werden können.*

Diese Definition wird im folgenden noch durch eine Reihe von weiteren Definitionen verfeinert werden. Prinzipiell ist die Funktionsweise einer Aktivität (wie schon im Eingang des Kapitels erwähnt) vergleichbar mit der einer objektorientierten Klasse. Sie beschreibt im Sinne einer Schablone, wie vom WFMS bei der Aktivierung der Aktivität dynamisch Instanzen der Aktivität (die Objekte einer Klasse im objektorientierten Sinne) erzeugt werden. Solche Aktivitäteninstanzen haben einen internen Zustand, der durch eine Reihe von innerhalb der Instanz verfügbaren Variablen beschrieben wird. Die Variablen legen instanz-fremde Ressourcen fest, die von der Instanz genutzt werden (Eingangsgrößen), definieren Ressourcen, die von der Instanz erzeugt und in den weiteren Workflow exportiert werden (Ausgangsgrößen) und stellen innerhalb der Instanz als Hilfsgrößen verwendete Variablen bereit.

Die innerhalb einer Aktivitäteninstanz definierten Größen werden dann von Aktionsbeschreibungen (*Aktionshandlern*, siehe Definition 3.12), die den Methoden einer objektorientierten Klasse entsprechen, verwendet. Im Vergleich zur objektorientierten Klasse neu, aber wesentlich, ist der Punkt (3) der $\mathcal{W}$*FLOW*–Aktivitätendefinition. Jede Aktivitäteninstanz besitzt zur Ablaufsteuerung einen internen Zustandsautomaten, der zum einen die Zustände definiert, in denen sich die Aktivitäteninstanz während ihrer Lebenszeit befinden kann, und weiterhin festlegt, wann und unter welchen Bedingungen ein Zustandsübergang (in der Regel mit korrespondierender Ausführung eines zugehörigen *Aktionshandlers*) ausgeführt werden kann.

Im folgenden wird auf die Punkte (1) bis (3) aus Definition 3.10 genauer eingegangen. Den Anfang machen hierbei die internen Variablen einer Aktivitäteninstanz:

> **Definition 3.11 (Zustandsgrößen)** *Einer Aktivität können eine Reihe von Zustandsgrößen zugeordnet werden:*
>
> 1. **Eingangsgrößen** *beschreiben instanzen-fremde Ressourcen, die von der Aktivitäteninstanz benutzt und somit bei der Initialisierung einer Aktivitäteninstanz allokiert werden müssen.*

2. **Ausgangsgrößen** *halten die Ergebnisse der Arbeitsschritte innerhalb einer Aktivitäteninstanz fest. Sie stellen Ressourcen dar, die bei Beendigung der Instanz gegebenenfalls im WFMS gesichert werden müssen.*
3. **Interne Variablen** *beschreiben instanzen-interne Hilfsgrößen, die allen Aktionshandlern einer Aktivitäteninstanz zur Verfügung stehen.*

Werte von Eingangs– und Ausgangsgrößen können sowohl Containerobjekte oder Teile davon, wie Meta-Informations-Attribute oder einzelne Komponenten sein. Sie können einfache Variablenwerte vom Type Integer, Real oder String oder komplette Container darstellen. Ihre wichtigste Eigenschaft ist, daß ihre Werte bei der Initialisierung der Aktivitäteninstanz zugeordnet werden müssen. Dies kann automatisch oder interaktiv unter Einbeziehung eines Benutzers erfolgen. Definiert eine Eingangsgröße Modell beispielsweise ein Containerobjekt in einem Container Modelle, in dem eine Reihe verschiedener Modellobjekte gespeichert sind, so kann das WFMS bei der Initialisierung der Aktivitäteninstanz der Eingangsgröße Modell das aktuelle Defaultobjekt des Containers zuweisen oder aber dem Bearbeiter der Instanz die interaktive Auswahl eines Modells aus der Liste aller Objekte erlauben. Hierbei ist aber zu beachten, daß $\mathcal{W}*FLOW$ als Baukasten für Workflow-Management-Systeme für eine solche Interaktion mit dem Benutzer keinerlei Oberflächenelemente zur Verfügung stellt. Diese müssen von Systemen, die oberhalb von $\mathcal{W}*FLOW$ realisiert sind, selbst bereitgestellt werden. Die $\mathcal{W}*FLOW$–API bietet jedoch Standard-Methoden zum Initialisieren von Eingangsgrößen an, die von den höheren Schichten eines Workflowsystems in der Regel aber redefiniert werden. Das *PRAXIS*–System [DGS98a, DGS98b, DGS98c] stellt hierfür beispielsweise eine WWW-basierte Oberfläche bereit, über die ein Benutzer Eingangsgrößen aus einer Objektliste auswählen oder direkt mit einem Wert besetzen kann.
Ausgangsgrößen werden bei der Abarbeitung einer Aktivitäteninstanz im Laufe der Zeit Werte zugeordnet, die Arbeitsergebnisse der durchgeführten Arbeitsschritte darstellen. Bei erfolgreicher Beendigung der Aktivitäteninstanz überprüft diese, ob alle Ausgangsgrößen mit gültigen Werten belegt sind. Ist das nicht der Fall, versucht die Instanz die noch nicht definierten, aber benötigten Ausgangsgrößen durch Aufruf von vordefinierten Methoden zu ermitteln. Diese Methoden werden, in Analogie zu den entsprechenden Methoden der Eingangsgröße, von den höheren Realisierungsschichten einer WFMS Applikation in der Regel redefiniert. So kann z. B. der Bearbeiter einer Aktivitäteninstanz bei ihrer Beendigung zur Angabe fehlender Ausgabegrößen durch eine geeignete GUI–Oberfläche aufgefordert werden. *PRAXIS* nutzt auch hier eine formularbasierte WWW–Oberfläche, um die fehlenden Werte zu erfragen bzw. den Bearbeiter die Ergebnisdaten mittels Datei–Upload ins Workflowsystem transferieren zu lassen.

Das Codefragment aus Beispiel 3.2.1 zeigt die Zuordnung von einzelnen Größen zu einer Aktivität. Die Anweisung (1) definiert zwei interne Variablen, denen zum Definitionszeitpunkt noch kein Wert zugeordnet ist. Anweisungen (2) bis (4) setzen Perlvariable auf die benötigten Container, und Anweisungen (5) bis (7) ordnen die Container der Aktivität ($activity) zu.

Nach Definition der Eingangs–, Ausgangsgrößen und internen Variablen können die Aktionen definiert werden, die im Rahmen der Abarbeitung einer Aktivitäteninstanz ausgeführt werden.

Definition 3.12 (Aktionshandler) *Aktionshandler beschreiben innerhalb einer Aktivität die computergestützten Abläufe, die nötig sind, um einen Bearbeiter bei der Durchführung von Arbeitsschritten zu unterstützen. Es gibt sie in zwei Ausprägungen:*

```
use WildFlow::API;
begin 'update', sub {
   ...
   $activity->insert_attribute('Verantwortlicher', undef);           # (1)
   $activity->insert_attribute('Startzeitpunkt, undef);              #

   $modell_container = WildFlow::Container->new(NAME=>'CFX Modelle', # (2)
                                                REPOSITORY=>CFX);

   $steuer_container = WildFlow::Container->new(NAME=>'CFX Steuerdaten',
                                                 REPOSITORY=>CFX);    # (3)

   $dump_container = WildFlow::Container->new(NAME=>'CFX Simulationsdaten',
                                               REPOSITORY=>CFX);      # (4)


   $activity->add_container(CONTAINER=>$modell_container,            # (5)
                            TYPE=>'in');

   $activity->add_container(CONTAINER=>$steuer_container,            # (6)
                            TYPE=>'in');

   $activity->add_container(CONTAINER=>$dump_container,              # (7)
                            TYPE=>'inout');
   ...
}
die "$@\n" if $@;
```

Beispiel 3.2.1: Codebeispiel für die Zuordnung von Zustandsgrößen zu Aktivitäten

1. *Ein interner Aktionshandler ist ein Codefragment, das innerhalb der Workflow-Engine ausgeführt wird.*
2. *Ein externer Aktionshandler wird zur Anbindung einer externen Anwendung eingesetzt. Hierzu wird dem Handler eine abstrakte Methodenbeschreibung (s. Abschnitt 4.4), welche die externe Aktion beschreibt, zugeordnet. Die Ausführung erfolgt über die W*FLOW*–Toolservices *(Kapitel 4).*

Die Ausführung eines *Aktionshandlers* ist in zwei Varianten möglich:

Synchrone Ausführung: In diesem Fall wird das Codefragment oder die abstrakte Methodenbeschreibung gestartet und auf das Ende der Bearbeitung gewartet.

Asynchrone Ausführung: Hierbei wird das Codefragment oder die abstrakte Methodenbeschreibung gestartet, aber nicht auf das Ende gewartet.

Die beiden Arten von Handlern unterscheiden sich in einigen wichtigen Punkten:

Ort der Ausführung: Ein interner Handler wird immer innerhalb der Workflow-Engine ausgeführt. Durch die API hat er Zugriff auf die komplette Engine und kann auf alle Informationen zugreifen. Der Handler kann somit sowohl zur Erweiterung der Funktionalität der

kompletten Workflow-Engine als auch als Schnittstelle zum lesenden oder schreibenden Zugriff auf die einzelnen Aktivitäten– und Containerobjekte eingesetzt werden.

Im Gegensatz hierzu ist ein externer Handler nicht an die Workflow-Engine gebunden und kann auf einem beliebigen Rechner, auf dem ein Toolstarterprozeß (Abschnitt 4.5.3) läuft, ausgeführt werden. Ein Handler wird im Rahmen der Handlerdefinition als abstrakte Methodenbeschreibung definiert, deren Auflösung in ein konkretes Kommando oder Skript erst zur Laufzeit des Workflows mit Hilfe des *Toolservers* (Abschnitt 4.5.2) erfolgt.

Benutzerinteraktion: Interne Handler laufen stets ohne Benutzerinteraktion ab, da sie innerhalb der Workflow-Engine ausgeführt werden. Im Gegensatz dazu können Benutzer, je nach ausgeführtem Programm, mit externen Handlern interagieren.

Ablauf: Die Unterscheidung in synchrone und asynchrone Aufrufe hat bei internen und externen Aktionshandlern unterschiedliche Auswirkungen. Externe Aktionshandler, die durch eigenständige Programme realisiert sind, laufen in beiden Modi in einem eigenen Prozeßkontext des jeweiligen Betriebssystems ab. Im Gegensatz hierzu laufen die Codefragmente der internen Handler im selben Prozeßkontext wie der jeweilige Teil[24] der $\mathcal{W}$*FLOW*–Engine ab. Damit ein Fehler (Syntax– und Laufzeitfehler) die $\mathcal{W}$*FLOW*–Engine nicht zum Absturz bringt, laufen die Codefragmente in einem besonders geschützten Bereich der $\mathcal{W}$*FLOW*–Engine ab, der es erlaubt, solche Fehler abzufangen. Im Falle der synchronen Abarbeitung wird das Codefragment in einem ersten Schritt übersetzt und anschließend innerhalb der Engine ausgeführt. Im Falle der asynchronen Ausführung muß für das Codefragment nach der Übersetzung ein eigener Thread [KSS96] erzeugt werden, innerhalb dem das Codefragment dann parallel zum $\mathcal{W}$*FLOW*–Engine Programmablauf ausgeführt wird. Da Threads erst seit kurzem[25] in *Perl* unterstützt werden, ist eine asynchrone Behandlung von internen Handlern zwar vorgesehen, zur Zeit aber noch nicht realisiert.

Realisierungsschnittstelle: Interne Handler sind Code-Fragmente, die in der Skriptsprache geschrieben werden müssen, die die Workflow-Engine unterstützt. Dadurch haben sie allerdings direkten Zugriff auf sämtliche API's des $\mathcal{W}$*FLOW*–Baukastens. Es handelt es sich somit um eine *Whitebox*–Integration, da der direkte Zugriff auf die Engine, mittels der $\mathcal{W}$*FLOW*–API gewährleistet ist. Externe Handler werden über den *Toolserver* auf Programme, Skripte oder dergleichen abgebildet, die innerhalb der Workflow-Applikationsumgebung vom *Toolstarter* gestartet werden. Externe Handler können sowohl als *Blackbox*–, *Greybox*– als auch als *Whitebox*–Integration[26] vorliegen.

Externe Handler können, wie es beispielsweise in *PRAXIS* realisiert ist, auf WWW-basierte Programme erweitert werden. In diesem Fall wird der Begriff des Programms einfach auf URLs und *Java*–Applets ausgedehnt.

Im folgenden Beispiel 3.2.2 soll die Definition von Aktionshandlern gezeigt werden.

In diesem kurzen Codefragment werden die zwei unterschiedlichen Möglichkeiten, einen Aktionshandler zu definieren, vorgestellt. Bei der Variable $activity handelt es sich um eine zuvor definierte Instanz der Klasse WildFlow::Activity. Der Methodenaufruf define_handler(...) erlaubt die Erzeugung eines Handlers und seine Verknüpfung mit einem Zustandsübergang. Im ersten

[24] Da $\mathcal{W}$*FLOW* verteilt auf mehreren Rechnern ablaufen kann, läuft eine $\mathcal{W}$*FLOW* Anwendung möglicherweise auch in mehreren unterschiedlichen Prozeßkontexten ab.

[25] Threads werden in Perl ab der Version 5.005 unterstützt.

[26] Eine ausführliche Beschreibung der drei Integrationsarten findet sich weiter hinten in Abschnitt 4.1.

```
begin 'update', sub {
  ...
    my $package = "CFX";
    my $method  = "Simulate";

    my $ch = $activity->define_handler('Initialisiert'=>'Laufend',  # (1)
                                       NAME=>'starten',
                                       PACKAGE=>$package,
                                       METHOD=>$method);

    $activity->define_handler('Laufend'=>'Kontrollmodus',           # (2)
                              NAME=>'Beobachten',
                              CODE=>q($self->log_info(TRUE));
    ...
}
die "$@\n" if $@;
```

Beispiel 3.2.2: Codebeispiel zur Erzeugung von zwei *Aktionshandlern*

Aufruf (1) wird ein externer Aktionshandler für den Zustandsübergang von Initialisiert nach Laufend installiert. Der Übergang wird mit dem Namen starten verknüpft, der dann zur Laufzeit als Name der Methode zur Durchührung des Übergangs zur Verfügung steht. Die Definition eines externen Handlers erfolgt durch Angabe des abstrakten Paket– und Methodennamens (CFX, Simulate).
Die zweite Anweisung (2) ist ein Beispiel für eine interne Handler-Definition. In diesem Fall wird kein Paket– und Methodenname angegeben, sondern es erfolgt die Definition eines Codefragmentes oder einer kompletten Methode. Der Code nutzt die $\mathcal{W}*FLOW$–API zum Zugriff auf die Engine (so handelt es sich bei der Variablen $self um eine von der API vordefinierten Variable, die der Aktivitäteninstanz zur Laufzeit entspricht). In dem konkreten Fall wird im Rahmen des *Handler-Codes* die interne Variable log_info der Aktivitäteninstanz mittels der entsprechenden Zugriffsmethode gleichen Namens auf den Wert **TRUE** gesetzt.

Wie die abstrakte Methodenbeschreibung und ihre Umsetzung auf externe Programme im einzelnen aussieht, wird im nachfolgenden Kapitel Toolservices erläutert. Als nächstes wird nun auf Punkt (3) aus Definition 3.10 näher eingegangen, dem Zustandsautomaten, der den Ablauf einer Aktivitäteninstanz steuert.

Definition 3.13 (Zustandsautomat einer Aktiviät) *Die Ablaufsemantik einer Aktivität wird durch einen Zustandsautomaten beschrieben, der die folgenden Eigenschaften hat:*

1. *Der Zustandsautomat besitzt die fünf vorgegebenen Zustände* Instanziiert, Initialisiert, In-Arbeit, Abgeschlossen *und* Abgebrochen. Initialisiert *ist ein Startzustand des Automaten,* Abgeschlossen *und* Abgebrochen *sind Finalzustände. Der Zustand* In-Arbeit *kann entsprechend der noch folgenden Definition 3.14 verfeinert, d. h. in mehrere benutzerdefinierte Zustände und Zustandsübergänge unterteilt werden.*
2. *Es gibt die folgenden, vom System vorgegebenen Zustandsübergänge:*
   i.) →**Instanziiert:** *Im Rahmen des Überganges wird eine neue Aktivitäteninstanz, entsprechend den Vorgaben der Aktivität, erstellt.*

ii.) **Instanziiert→Initialisiert:** *Der Übergang ist für die Initialisierung und Allokation der durch die Ein– und Ausgabegrößen beschriebenen Ressourcen verantwortlich. Weiterhin wird der Instanz ein Benutzer zugeordnet.*

iii.) **Initialisiert→Instanziiert:** *Die allokierten Ressourcen und der zugeordneten Bearbeiter werden wieder freigegeben.*

iv.) **Initialisiert→In-Arbeit:** *Dieser Übergang führt den Zustandsautomat in die eigentliche Bearbeitungsphase.*

v.) **In-Arbeit→Abgeschlossen:** *Die Bearbeitungsphase einer Aktivitäteninstanz wird beendet. Die erzielten Ergebnisse (Ausgangsgrößen) werden ins WFMS eingebracht und anschließend die allokierten Ressourcen wieder freigegeben.*

vi.) **In-Arbeit→Abgebrochen:** *Der Übergang ist analog zum vorherigen, mit dem Unterschied, daß die erzielten Ergebnisse verworfen und nicht ins WFMS eingebracht werden.*

Diese Übergänge können im Rahmen einer konkreten Anwendung überschrieben oder erweitert werden.

3. *Zustandsübergänge können mit Aktionshandlern verknüpft werden, die ausgeführt werden, wenn der Zustandsübergang stattfindet.*

4. *Mit jedem Zustandsübergang kann eine Vorbedingung verknüpft werden, die erfüllt sein muß, damit der Zustandsübergang stattfinden und der Aktionshandler ausgeführt werden kann.*

5. *Mit jedem Zustandsübergang kann eine Nachbedingung verknüpft werden. Die Bedingung wird nach Ausführung eines eventuell vorhandenen Aktionshandlers und vor der Annahme des Zielzustandes ausgewertet. Der Zielzustand wird nur dann angenommen, wenn die Nachbedingung erfüllt ist. Andernfalls wird der Zielzustand verworfen.*

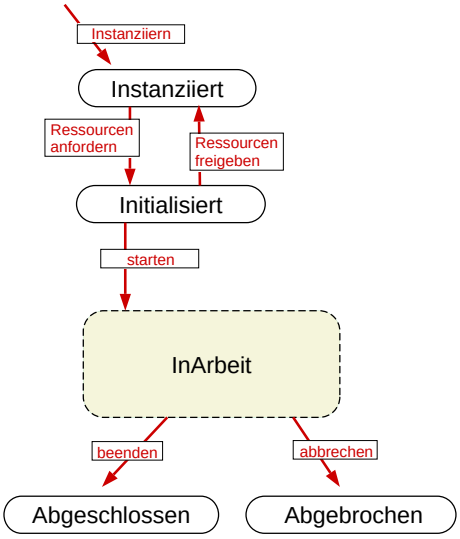


Abbildung 3.14: *Zustandsübergänge bei Aktivitäten*

Bild 3.14 zeigt den Zustandsautomaten, gemäß Definition 3.13. Wird bei der Aktivierung einer Aktivität eine Aktivitäteninstanz erzeugt, so geht diese durch einen initialen Zustandsübergang

instanziieren in den Zustand Instanziiert über. Hierbei wird gegebenenfalls die Vorbedingung zum Zustandsübergang instanziieren evaluiert und nachgeprüft, ob alle benötigten Ressourcen (Eingangsgrößen) in genügender Zahl vorhanden sind. Bevor mit der neuen Instanz gearbeitet werden kann, müssen zunächst die ständig benötigten Arbeitsressourcen allokiert werden. Dies geschieht in einem Zustandsübergang Ressourcen anfordern, der die Instanz in den Zustand Initialisiert überführt.

Beim Übergang in den Zustand Initialisiert wird der Instanz zunächst ein Bearbeiter zugeordnet. Unter eventuell interaktiver Beteiligung dieses Bearbeiters werden dann die benötigten Ressourcen (Eingangsgrößen) ausgewählt und reserviert. Zustand und Art der Reservierung sind hierbei abhängig vom Zugriff zur Laufzeit der Aktivitäteninstanz; wird die Ressource eventuell modifiziert und anschließend zurückgeschrieben, so sind im allgemeinen andere Sperrmodi notwendig, als wenn die Ressource nur gelesen wird. Mittels des Übergangs Ressourcen freigeben kehrt der Automat wieder in den Zustand Instanziiert zurück, eventuell weil die Instanz einen anderen Bearbeiter zugeordnet werden soll oder muß. Der Übergang starten führt den Automaten in den Zustand In-Arbeit. Wird der Zustandsautomat einer Aktivität nicht verfeinert, so ist dieser Übergang für den Start oder die Durchführung einer Anwendung zuständig. Nach erfolgreichem Abschluß der Bearbeitung wird der Benutzer oder das Workflowsystem den Zustandsübergang beenden auslösen, in dessen Rahmen die Sicherung der erzielten Ergebnisse ins WFMS und die Freigabe der allokierten Ressourcen erfolgt. Im Falle eines Abbruchs der Bearbeitung wird der Übergang abbrechen ausgelöst, der, ohne bereits erzielte Ergebnisse zu sichern, die allokierten Ressourcen wieder freigibt. Um das spezielle Verhalten einer Aktivitäteninstanz modellieren zu können, ist es möglich, den Zustand In-Arbeit bei der Definition einer Aktivität zu verfeinern (vergl. Definition 3.14), d. h. er kann durch eine Reihe von Zuständen und Übergängen ersetzt werden. In diesem Fall befindet sich die Aktivitäteninstanz so lange im Zustand In-Arbeit, wie nur Zustandsübergänge innerhalb des verfeinerten Teils des Zustandsautomaten ausgeführt werden.
Wenn ein Zustandsübergang aus dem (verfeinerten) Zustand In-Arbeit herausführt, wird, je nach Aktion des Bearbeiters oder des Workflowsystems, einer der Zustandsübergänge beenden oder abbrechen, die vom Zustand In-Arbeit zu einem der Zustände Abgeschlossen oder Abgebrochen führen, ausgeführt. Dies führt nach Durchführung der notwendigen Arbeitsschritte (Sichern/Verwerfen der Ergebnisse, Freigabe der allokierten Ressourcen und Schreiben von Logdaten) zur Beendigung der Aktivitäteninstanz.

Nachdem die Bestandteile des Zustandsautomates nun dargelegt wurden, soll im folgenden näher auf die Punkte (3) bis (5) aus Definition 3.13 eingegangen werden. Die Punkte befassen sich mit dem Zusammenspiel von Zustandsübergängen, Aktionshandlern sowie Vor– und Nachbedingungen.
Das Zusammenwirken soll anhand eines einzelnen Übergangs genauer betrachtet werden. Abbildung 3.15 zeigt an einem weiteren Zustandsautomaten, wie ein Zustandsübergang zwischen einem Zustand X und einem Zustand Y im einzelnen abläuft.

Neben den benutzersichtbaren Zuständen X und Y existieren noch eine Reihe von internen Zwischenzuständen, die protokollieren, ob eine Vorbedingung erfüllt, die Aktion gestartet (synchron/asynchron, mit oder ohne Warteoption) oder ob die Aktion beendet ist, bzw. ob eine Fehlersituation vorliegt. Es wird deutlich, daß der Zwischenzustand Vorbedingung erfüllt Voraussetzung dafür ist, daß in einem zweiten Schritt der Aktionshandler gestartet werden kann. Hierbei sind drei Fälle zu unterscheiden.

Im synchronen Fall wird der Aktionshandler der Aktivitäteninstanz gestartet und die Aktivitäteninstanz wartet dann auf die Rückgabe des Exit-Status des intern aufgerufenen internen

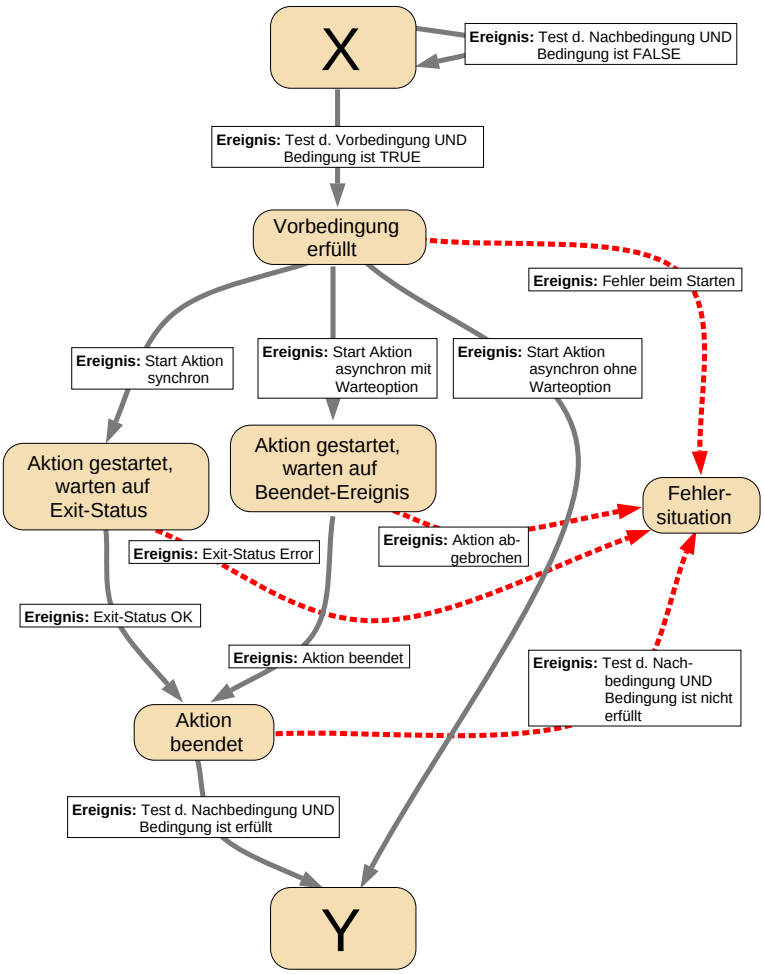


Abbildung 3.15: *W*FLOW: interner Ablauf eines Zustandsübergangs*

Handlers (Codefragment) oder auf die Zurückgabe des Exit-Status einer durch den *Toolstarter* gestarteten externen Aktion. Im letzten Fall gibt der *Toolstarter* den Exit-Status des von ihm ausgeführten Toolaufrufs synchron an die Aktivitäteninstanz zurück.
Im asynchronen Fall startet die Aktivitäteninstanz eine externe Anwendung über den *Toolstarter* und weist ihn an, das externe Tool asynchron zu starten. Der *Toolstarter* wartet in diesem Fall nicht auf die Rückgabe eines Exit-Codes des gestarteten Tools, sondern gibt nur eine Fehlerindikation (Fehler beim Starten) an die Aktivitäteninstanz zurück, wenn das Tool nicht gestartet werden konnte. Je nachdem, ob der asynchrone Aktionshandler mit oder ohne Warteoption gestartet wird, wartet die Aktivitäteninstanz jetzt auf ein Ereignis, das ihr anzeigt, daß der externe Toolaufruf beendet ist oder die Aktivitäteninstanz geht nach erfolgreichem Start direkt in den Zustand Y über. Im ersten Fall wird der externe Toolaufruf in der Regel durch einen Wrapper implementiert, der mit dem Workflowsystem durch eine *Whitebox*–Integration verbunden ist. Der Wrapper kontrolliert nach Beendigung des Tools den korrekten Ablauf der Toolaktion, setzt gegebenenfalls einige Statusvariablen innerhalb der Aktivitäteninstanz und triggert dieser dann das Ereignis Aktion beendet. Die Aktivitäteninstanz prüft im Anschluß daran die Nachbedingung, in der eventuell einige durch den Wrapper gesetzte Statusvariablen mit eingebunden sind. Wenn der Aktionshandler asynchron ohne Warteoption ausgeführt wird, macht die Auswertung einer Nachbedingung im Anschluß an den Start der Toolaktion keinen Sinn, da ja nichts über den

Status des aufgerufenen Tools selber bekannt ist. In diesem Fall wird nach einem erfolgreichen Start direkt der Zustand Y angenommen.

Synchrone Handler, die externe Tools aufrufen, werden in der Regel für eine einfache *Blackbox*–Integration eingesetzt, bei der der Exit-Status des aufgerufenen Tools Aufschluß über Erfolg oder Mißerfolg des Toolaufrufs geben kann. Dies ist bei batchorientierten, nicht interaktiven externen Toolaufrufen (z. B. Datenkonvertierungswerkzeugen, etc.), weniger bei interaktiven Tools der Fall. Synchrone interne Handler laufen sowieso automatisch im System ab und ihr Ergebniswert ist direkt innerhalb des Codes der Workflow-Engine abrufbar. Asynchrone Handler mit Warteoption können immer dann eingesetzt werden, wenn nach Beendigung des externen Tools der damit verbundene Arbeitsvorgang auf Erfolg geprüft werden muß. Das kann automatisch innerhalb eines Wrappers oder manuell durch den Benutzer geschehen. Im ersten Fall löst der Wrapper innerhalb der Aktivitäteninstanz das Ereignis Tool beendet aus. Im zweiten Fall geschieht dies durch den Benutzer.

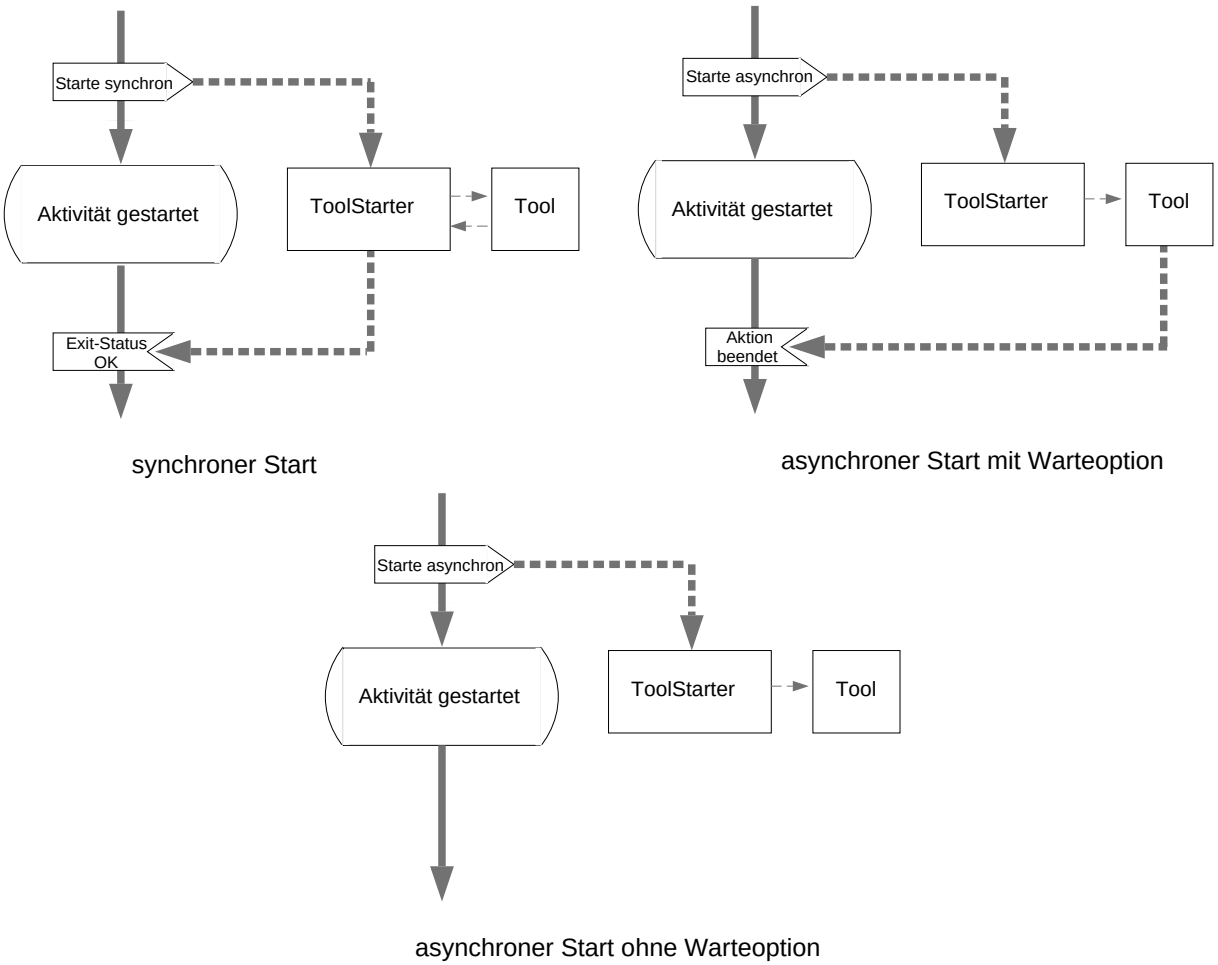


Abbildung 3.16: *Unterscheidung zwischen synchronen und asynchronen Aktionsaufrufen in UML Notation*

Der Unterschied zwischen synchronem Start und asynchronem Start mit und ohne Warteoption von externen Tools ist zur Verdeutlichung des Unterschieds der Ereignisquelle in Abbildung 3.16 in Anlehnung an die UML Notation für Aktivitätendiagramme dargestellt. Explizit ist das Sen-

den und Empfangen von Ereignissen (gestrichelte Linien) dargestellt.

Generell müssen Zustandsübergänge, damit sie stattfinden, getriggert werden. Das kann entweder automatisch durch das System oder manuell durch den Benutzer erfolgen, der durch eine Interaktion mit dem Workflowsystem einen Zustandsübergang auslöst. Bei der Definition der Zustandsübergänge wird angegeben, auf welche "Ereignisse" reagiert werden soll. Aktionshandler aber auch Vor– und Nachbedingungen, die nur spezielle Aktionshandler darstellen, können Ereignisse an Aktivitäteninstanzen senden, die dann einen mit diesem Ereignis verknüpften Zustandsübergang auslösen. Liegt in einem benutzerdefinierten Zustandsautomaten z. B. die Zustandsfolge A $\rightarrow$ B $\rightarrow$ C an, so kann ein Aktionshandler beim Übergang von A $\rightarrow$ B beispielsweise ein Ereignis triggern, das automatisch den Zustandsübergang von B $\rightarrow$ C auslöst. Auf diese Weise können ganze Ketten von Arbeitsvorgängen innerhalb einer Aktivitäteninstanz automatisch angestoßen werden.

Im Falle, daß bei der Abarbeitung ein Fehler auftritt (z. B. Aktionshandler kann nicht korrekt gestartet werden bzw. wird abgebrochen, Nachbedingung scheitert, etc.) wird der interne Zustand Fehlersituation angenommen. Die Intension hierbei ist, einen einheitlichen Punkt zur Bearbeitung von Fehlersituationen, etwa durch die Behandlung mittels zu realisierender hochentwickelter Ausnahmebehandlungsverfahren, z. B. [CFM98, Lie98, HA98b], oder auch nur zur Hinzunahme des Benutzers, etc. zu besitzen.
Indem eine einheitliche Behandlung der Übergänge realisiert wird, unabhängig davon, ob diese mittels externer (Benutzer, externes Programm, Betriebssystem, etc.) oder interner ($\mathcal{W}$*FLOW*–Engine) Ereignisse getriggert werden, läßt sich ein einfach zu konfigurierender Zustandsautomat bauen, der es sowohl erlaubt auf externe Ereignisse zu reagieren, bzw. sie automatisch intern zu generieren, was eine flexible und situationsbezogene Abarbeitung des Zustandsautomats und damit verbunden der auszuführenden Aktionen, erlaubt.

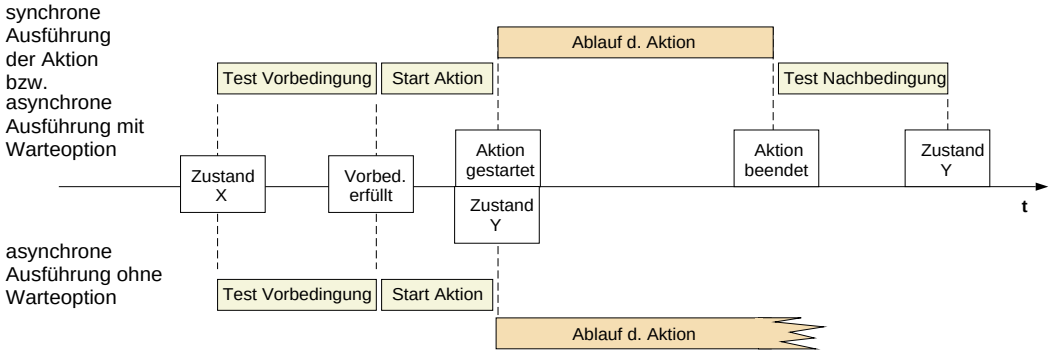


Abbildung 3.17: *Unterscheidung zwischen synchronen und asynchronen Aktionsaufrufen am Beispiel eines erfolgreichen Überganges*

Bisher wurden zwei Arten von Informationen vorgestellt, auf denen Bedingungen evaluieren können. Neben diesen beiden, auch von der WFMC unterschiedenen Informationsquellen [Hol94b], identifizieren Alonso et. al. [AAAM97] noch eine dritte Art von Information, welche in Bedingungen verarbeitet werden können. Hierbei handelt es sich um externe Ereignisse (Events). Im Gegensatz zu Bedingungen, welche auf Applikations– bzw. Workflowdaten basieren, können Bedingungen, welche auf externen Ereignissen basieren, ihren Status mit der Zeit ändern, d. h. sie sind *dynamisch*. Das erfordert vom Workflowsystem zusätzliche Mechanismen und ist ein Grund dafür, daß nur die wenigsten Systeme mit externen Ereignissen umgehen können, obwohl diese einen Schlüsselaspekt bei der Synchronisation von Workflowsystemen spielen können [AAAM97].

So ist es beispielsweise vorstellbar, daß bestimmte Aktionen nur zu bestimmten, durch externe Ereignisse ausgelöste Zeitpunkte, ausführbar werden. Indem es erlaubt ist, daß solche externen Ereignisse, Ereignisse innerhalb der Aktivitäteninstanz triggern, kann erreicht werden, daß diese externen Ereignisse die Ausführung von Aktionshandlern bewirken und somit zur Synchronisation von Workflows beitragen können. Als Beispiel sei hier die Lackierung eines Bauteils mittels computergesteuertem Spritzroboter angeführt. Bevor die Lackierung beginnen kann, muß das Bauteil in der Spritzkabine justiert und der Arbeitsbereich des Roboters gesichert sein, was durch die Verriegelung der Spritzzelle von außen geschieht. Die Betätigung der Verriegelung generiert ein externes Signal, das dann die Workflowengine dazu veranlaßt, die Steuerungssoftware für den Roboter zu laden.

Wie in Definition 3.13 Punkt (4) und (5) festgelegt, werden Vor– und Nachbedingungen ebenfalls mittels *Aktionshandler* repräsentiert. Hierbei werden jedoch noch spezielle Anforderungen an die Aktionshandler gestellt:

- Handelt es sich um einen *internen Aktionshandler*, so muß dieser einen boolschen Wert (**TRUE**, **FALSE**) zurückliefern.
- Bei einem *externen Aktionshandler* muß es sich um ein Skript handeln, das die $\mathcal{W}$*FLOW*–API nutzt und innerhalb des Skripts mit einer bestimmten API Methode den entsprechenden boolschen Wert an die $\mathcal{W}$*FLOW*–Engine schickt.

Die Möglichkeit, Vor– und Nachbedingungen sowohl als interne– als auch externe Handler zu formulieren, bringt folgende Konsequenzen mit sich:

Die Bedingungen können auf Basis von Applikationsdaten, auf Basis von Workflowdaten – oder einer Kombination aus beiden – formuliert werden. Die Möglichkeit Bedingungen, nicht nur auf Basis von Workflowdaten formulieren zu können, stellt einen Schlüsselaspekt bei der Formulierung komplexer Workflowsemantik dar, der bei heutigen Systemen so gut wie nicht realisiert ist [AAAM97].

Die Möglichkeiten, welche sich durch die Kombinationen von Vor–, Nachbedingung sowie den Zugriff auf die unterschiedlichen Daten ergeben, sollen kurz an ein paar Beispielen skizziert werden.

So kann zum Beispiel ein externer Handler definiert werden, der als Vorbedingung die Ausführbarkeit einer Anwendung auf einem externen Rechner überprüft.

Ein externer Handler kann als Nachbedingung überprüfen, ob die erzielten Ergebnisse bestimmten Anforderungen genügen, oder ob die Aktion als fehlgeschlagen angesehen werden muß.

Dadurch, daß die Bedingungen in Form von Perlcode definiert werden und nicht einem Logik-Kalkül folgen, bietet sich zudem noch die Möglichkeit, "Seiteneffekte" zu erzeugen. Diese Seiteneffekte können zum Beispiel im Rahmen einer externen Vorbedingung eine Reihe von Initialisierungen auf dem entsprechenden Rechner vornehmen und nur dann **TRUE** zurückliefern, wenn die Initialisierungen erfolgreich durchgeführt werden konnten. In einem anderen Fall kann im Rahmen einer externen Vorbedingung auf einem Rechner A kontrolliert werden, ob ein bestimmter Serverprozeß läuft. Der Prozeß kann dann gegebenenfalls erzeugt und anschließend auf Rechner B die eigentliche Applikation gestartet werden, welche den Serverprozeß auf Rechner A nutzt. Im Rahmen einer Nachbedingung kann ein nicht mehr benötigter Serverprozeß in analoger Weise gestoppt werden.

Weiterhin können Vor– und Nachbedingungen zur Formulierung von Schleifen benutzt werden oder aber auch eine einfache Ausnahmebehandlung im Rahmen der Nachbedingung durchführen. Durch die Ausdrucksmächtigkeit einer modernen Skriptsprache wie *Perl*, sowie der engen Anbindung der Schnittstelle an die einzelnen Komponenten sind hier nur wenig Grenzen gesetzt [LS98a, LS98c, Ous98, Ste98a].

Im folgenden soll nun betrachtet werden, wie der Zustandsautomat im Zustand In-Arbeit verfeinert werden kann:

Definition 3.14 (Verfeinerung des Zustandsautomates)

Der Zustand In-Arbeit *kann durch einen benutzerdefinierbaren Zustandsautomaten verfeinert werden:*

1. *Es können beliebig viele Unterzustände definiert werden. Eine Teilmenge dieser Zustände kann als Finalzustände gekennzeichnet werden.*
2. *Es können beliebige Zustandsübergänge zwischen den Unterzuständen definiert werden, die mit Aktionshandlern, sowie Vor– und Nachbedingungen versehen werden können.*
3. *Wird der Zustandsautomat im Zustand* In-Arbeit *verfeinert, so müssen die drei vom System vorgegebenen Zustandsübergänge* starten, beenden *und* abbrechen *durch neue Zustandsübergänge ersetzt werden, für die folgendes gelten muß:*
   Es muß mindestens ein Zustandsübergang definiert werden, der vom Zustand Initialisiert *in die Menge der unter Punkt (1) definierten Zustände führt. Weiterhin muß mindestens ein Zustandsübergang von einem Zustand der unter Punkt (1) definierten Zustände zum Zustand* Abgeschlossen *und mindestens ein Übergang zum Zustand* Abgebrochen *führen.*

Zur Illustration dieser Definition soll das einfaches Beispiel aus Abbildung 3.18 vorgestellt werden. Es zeigt die Aktivität Simulation aus Bild 3.11, welche eine Modellierung des einführenden Beispiels aus Abschnitt 1.2.3.1 graphisch darstellt. Im konkreten Fall wurde der Zustand In-Arbeit mittels den beiden Zuständen Laufend und Unterbrochen verfeinert. Die Modellierung trägt dem Umstand Rechnung, daß Simulationen oft sehr langandauernd und CPU-intensiv sind und es deshalb wünschenswert ist, einen Simulationslauf zeitweitig zu unterbrechen und später wieder fortzuführen. Dies wird durch die beiden Zustände ermöglicht, die den aktuellen Zustand eines Simulationslaufs speichern. Neben der Verfeinerung des Zustandes In-Arbeit wurden noch die Zustandsübergänge Start_SIM, Stop_SIM, Restart_SIM, Commit_SIM und Abort_SIM definiert. Die Zustandsübergänge enthalten die Aktionshandler, welche den Start, Stop, die Unterbrechung und den Wiederanlauf einer Simulation veranlassen. Durch die Verfeinerung des Zustandes In-Arbeit müssen die drei Zustandsübergänge starten, beenden und abbrechen gemäß Punkt 3 der Definition 3.14 durch neu zu definierende Zustandsübergänge ersetzt werden. Die Realisation erfolgt durch die Zustandsübergänge Start_SIM, Commit_SIM, Abort_RUN_SIM und Abort_SUSP_SIM. Ein Codefragment zur Definition dieser Aktivität ist in Beispiel 3.2.3 zu sehen. Zu Beginn des Codefragmentes Anweisung (1) werden die zwei Zustände Laufend und Unterbrochen als Verfeinerung des Zustandes In-Arbeit definiert. Anweisung (2) ersetzt den Zustandsübergang starten des ursprünglichen Automaten, während die Übergänge (3) und (4) die Verfeinerung des Zustandes In-Arbeit regeln. Anweisung (5) ersetzt den Übergang beenden. Die Anweisungen (6) und (7) ersetzen den Übergang abbrechen, wobei der Übergang in Anweisung (7) einen leeren *Aktionshandler* besitzt, d. h. es wird keine zusätzliche Aktion ausgeführt.

```
begin 'update', sub {
  ...
  my $activity = $workflow->new_activity(NAME=>'CFX-RUN');

  $act->set_states('Laufend','Unterbrochen');                    # (1)

  my $package = "CFX";

  my $method  = "Start_Simulate";
  my $ch = $act->define_handler('Initialisiert'=>'Laufend',      # (2)
                                NAME=>'Start_SIM,
                                PACKAGE=>$package,
                                METHOD=>$method);

  $method  = "Stop_Simulate";
  $act->define_handler('Laufend=>'Unterbrochen',                 # (3)
                        NAME=>'Stop_SIM,
                        PACKAGE=>$package,
                        METHOD=>$method);

  $method  = "Resume_Simulate";
  $act->define_handler('Unterbrochen'=>'Laufend',                # (4)
                        NAME=>'Restart_SIM',
                        PACKAGE=>$package,
                        METHOD=>$method);

  $method  = "End_Simulate";
  $act->define_handler('Laufend'=>'Abgeschlossen',               # (5)
                        NAME=>'Commit_SIM',
                        PACKAGE=>$package,
                        METHOD=>$method);

  $method  = "Abort_Simulate";
  $act->define_handler('Laufend'=>'Abgebrochen,                  # (6)
                        NAME=>'Abort_SIM',
                        PACKAGE=>$package,
                        METHOD=>$method);

  $act->define_handler('Unterbrochen'=>'Abgebrochen,             # (7)
                        NAME=>'Abort_SIM',
                        CODE=>undef);
  ...
};
die if $@;
```

Beispiel 3.2.3: Codebeispiel zur Erzeugung des Zustandsautomaten aus Abbildung 3.18

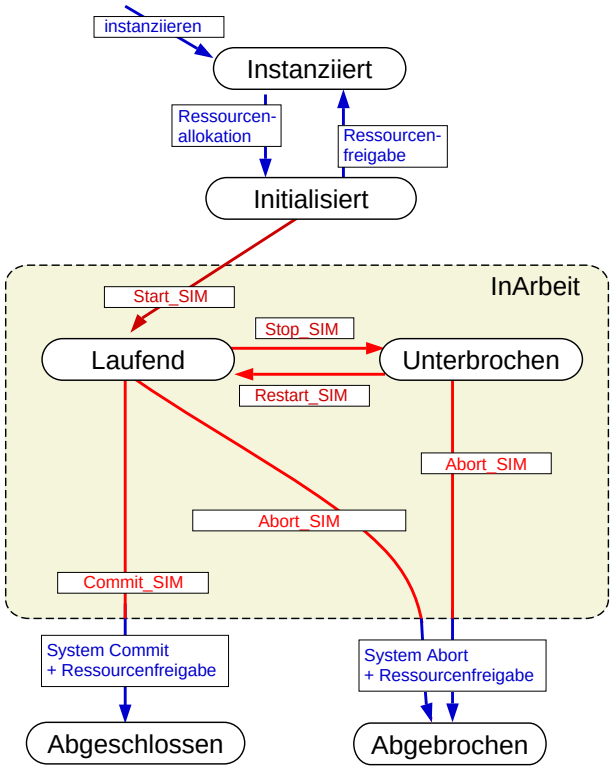


Abbildung 3.18: Beispiel CFX: *Verfeinertes Zusandsübergangsdiagramm*

3.2.4 Bewertung und Einsatzmöglichkeiten

Nachdem die Funktionalität der $\mathcal{W}*FLOW$–Aktivitäten in den vorherigen Abschnitten vorgestellt worden ist, sollen in diesem Abschnitt eine Reihe von Punkten diskutiert werden, die sich mit den verhaltens– und funktionsbezogenen Aspekten von Workflows beschäftigen. Die hier angesprochenen Punkte setzen auf die von $\mathcal{W}*FLOW$ zur Verfügung gestellte Funktionalität auf und zeigen, wie flexibel $\mathcal{W}*FLOW$ zum Aufbau von unterschiedlichen Systemen genutzt werden kann. Hierzu wird jeweils zu Beginn auf die zugrundeliegende Theorie der Thematik, bzw. die unterschiedlichen Realisierungsarten eingegangen und anschließend Lösungswege, basierend auf dem $\mathcal{W}*FLOW$ Baukasten vorgestellt.

3.2.4.1 Modellierung verhaltensbezogener Aspekte

Wie in Abschnitt 3.2.2 bereits festgelegt, werden verhaltensbezogene Aspekte von Workflows mit $\mathcal{W}*FLOW$ durch elementare Mechanismen, basierend auf Bedingungen und Beziehungen von Zuständen, Daten, Metadaten und sonstigen Informationsquellen, realisiert. In den folgenden Abschnitten soll obige allgemeine Aussage konkretisiert werden, und es soll detailliert gezeigt werden, wie sich mittels dieser Mechanismen der Kontrollfluß eines Workflows realisieren läßt, d. h. verschiedene Ausführungsfolgen wie Sequenz, Auslassung, Auswahl, Nebenläufigkeit, Schleifen, etc. modellieren lassen.

Wie ebenfalls bereits beschrieben, gibt es eine Vielzahl von unterschiedlichen Ansätzen zur Modellierung des Kontrollflusses. Im folgenden sollen die verschiedenen Ansätze nach einer von Böhm [BMWCJ98] vorgeschlagenen Klassifikation, basierend auf den unterschiedlichen Arten der Beziehungen zwischen den Zuständen, kurz besprochen werden.

In [BMWCJ98] werden die verschiedenen Modellierungsmechanismen in die drei Klassen *Kon-*

trollflußprimitive, *Kontrollflußkonstrukte* und *Ausführungsanweisungen* unterschieden.
Als einfachste Elemente werden die *Kontrollflußprimitive* vorgestellt. Diese einfachen Ablaufkonstrukte setzen die Zustände von zwei oder mehreren Aktivitäten in Beziehung. Mittels ihnen werden Kausalbeziehungen direkt ausgedrückt. Beispiel hierfür ist der *Kontrollflußkonnektor* des Workflowsystems *FlowMark*. Ein Kontrollflußkonnektor setzt die Zustände startable und done von zwei Aktivitäten so in Beziehung, daß bei Erreichen des Zustandes done der einen Aktivität die zweite Aktivität in den Zustand startable übergeht. Zusätzlich kann noch eine zur Laufzeit auszuwertende Transitionsbedingung mit angegeben werden. Die Interpretation ist hierbei von *FlowMark* fest vorgegeben und startable und done stellen die einzigen sichtbaren Zustände dar, an die ein Kontrollflußkonnektor angebunden werden kann. Abbildung 3.19 zeigt auf der linken Seite ein *FlowMark* Kontrollflußprimitiv, das die beiden Zustände startable (S) und done (D) in Beziehung setzt und auf der rechten Seite ein Kontrollflußprimitiv, das drei Aktivitäten in Beziehung setzt.

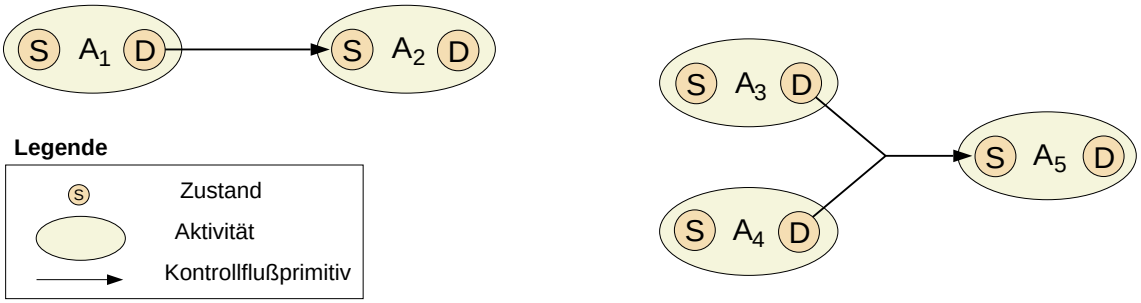


Abbildung 3.19: *Beispiele für Kontrollprimitive*

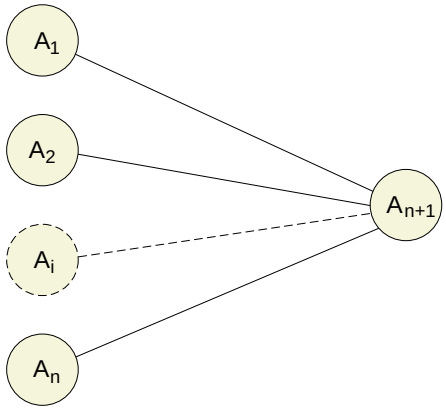
Die Ausdrucksmächtigkeit von Kontrollflußprimitiven hängt sehr stark von den sichtbaren Zuständen einer Aktivität ab. So läßt sich beispielsweise in *FlowMark* keine Parallelität von zwei Aktivitäten erzwingen, da sich das gleichzeitige Ablaufen von zwei Aktivitäten mit den beiden Zuständen startable und done nicht modellieren läßt.

Semantisch höher stehen die *Kontrollflußkonstrukte*. Sie bestehen aus einer Menge von Kontrollflußprimitiven und einer Funktion, die das Verhalten der Primitive festlegt. Beispiele hierfür sind die von der WFMC vorgeschlagenen Elemente AND–Split, OR–Split, OR–Join, etc. [WFM99]. Beispielhaft soll im folgenden das AND–Join *Kontrollflußkonstrukt* betrachtet werden. Ein AND–Join schaltet genau dann weiter, wenn alle n Kontrollflußprimitiven die Beendigung der Durchführung ihrer Aktivität signalisieren. Diese Funktionalität wird durch die Funktion auf der rechten Seite der Abbildung 3.20 beschrieben. Durch den Aufbau eines Kontrollflußkonstruktes aus mehreren Primitiven und einer zusätzlichen Funktion, die das Verhalten definiert, erlaubt der Einsatz von Kontrollflußkonstrukten einen übersichtlicheren Aufbau von Workflowschemata, wie es mit dem Einsatz von Kontrollflußprimitiven möglich wäre.

Als letzte Klasse von Kontrollflußdefinitionen existieren die *Ausführungsanweisungen*. Im Unterschied zu den beiden vorherigen Klassen definieren diese den Kontrollfluß nicht explizit, sondern der auszuführende Workflow wird direkt mittels den Ausführungsanweisungen angegeben. Als Beispiel sei hier die parallele Ausführung von zwei Aktivitäten genannt, die folgendermaßen ausgedrückt werden kann.

```
parallel($activity_1, $activity_2);
```

Hierbei liegt der Fokus eher auf einer deklarativen Beschreibung des Workflows als auf einer




```
$ok = 1;
foreach $i (1..$n)  {
    if ($a[$i]->state() ne "done") {
        $ok = 0;
        last;     # Abbruch d. Schleife
    }
}
if ($ok) {
    $a[$n+1]->activate();
}
```

Abbildung 3.20: *Beispiel für Kontrollflußkonstrukt*

ablauforientierten Durchführungsvorschrift. Ausführungsanweisungen können mittels Kontrollflußprimitiven oder Kontrollflußkonstrukten erstellt werden. Der Workflowschema-Entwickler kommt mit diesen Elementen aber nicht mehr in Berührung, sondern formuliert seine Workflows auf Basis der ihm zur Verfügung stehenden Ausführungsanweisungen. Die Präzision der formulierbaren Workflows bleibt deshalb aber hinter denen von Kontrollflußprimitiven und –konstrukten zurück, allerdings sind sie einfacher anwendbar und die so aufgebauten Workflows sind weitaus übersichtlicher. Ein Vertreter von Workflowsystemen, der Ausführungsanweisungen unterstützt, ist das forschungsorientierte System $\mathcal{MOBILE}$ (siehe Abschnitt 1.2.2).
In [BMWCJ98] wird eine interesante Kombination aus Kontrollflußprimitiven und Ausführungsanweisungen in der Art vorgeschlagen, daß es dem Entwickler von Workflow–Schemata möglich sein sollte, die ihm zur Verfügung stehende Sprache um eigene Ausführungsanweisungen zu erweitern. Diese kann der Schema-Entwickler aus Kontrollflußprimitiven selbst aufbauen und somit semantisch hochstehende Ausführungsanweisungen für seine spezielle Aufgabenstellung entwickeln.

Nach der Übersicht, wie verschiedene Ablaufsteuerungen eines Workflowsystems realisiert werden können, sollen im folgenden die unterschiedlichen Arten von Beziehungen, die zwischen Aktivitäten bestehen können, genauer untersucht werden.
Wie bereits erwähnt, ist im Beispiel des *FlowMark* Kontrollflußkonnektors die Interpretation des Verhaltens vom System her festgelegt. Es entspricht dem Beziehungstyp einer *State/Transition*–Beziehung (S/T) mit der *Modalität* "muß". Dies bedeutet, daß bei Eintritt eines Zustandes S von Aktivität 1 ein Übergang T in Aktivität 2 erfolgen muß.
Eine andere mögliche Interpretation der S/T Beziehung besteht etwa darin, nach Erreichen eines Zustandes S den Übergang T zu erlauben. In diesem Fall wird von einer "kann"–Modalität gesprochen.
Ähnlich den S/T Beziehungstypen verhalten sich die *Transition/Transition*–Beziehungstypen (T/T). Im Unterschied zu S/T Typen erzwingt/ermöglicht das Auftreten eines bestimmten Übergangs einer Aktivität 1 einen Zustandsübergang einer Aktivität 2. Beispielsweise läßt sich hiermit auf einfache Art und Weise die parallele Ausführung von Programmen modellieren. Gemeinsam ist beiden Beziehungstypen, daß die Wirkung in einem (möglichen) Übergang besteht. Es wird hierbei auch von *ablauforientierten* Vorschriften gesprochen.

Daneben können aber auch Beziehungen angegeben werden, die zwischen zwei Aktivitätenzuständen (S/S) bzw. zwischen einem Übergang und einem Zustand (T/S) bestehen. Der Unterschied zu den beiden ersten Typen besteht darin, daß sich bei der Formulierung der Beziehungen

keine Aktionen beschreiben lassen, sondern die Wirkung in der Überprüfung und gegebenenfalls Verhinderung von unmöglichen Zustandskombinationen besteht. Im Gegensatz zu den ablauforientierten Vorschriften der S/T bzw. T/T Beziehungen handelt es sich hier um eine deklarative Vorgabe. In diesem Fall ist die Rolle des Workflowsystems weniger aktiv, vielmehr überläßt es dem Benutzer die Initiative und ist lediglich für die Einhaltung der mittels S/S und T/S definierten Randbedingungen zuständig. Die deklarative Art der Kontrolle ist speziell unter dem Gesichtspunkt wissenschaftlich–technischer Arbeitsabläufe interessant, die, wie in Abschnitt 1.2.3.2 dargelegt, ein weitaus größeres Maß an Flexibilität bieten müssen, als es bei Produktions– oder Administrations–Workflows notwendig ist.

Aufgrund der verschiedenen interessanten Gesichtspunkte, die sich aus der Kombination von Beziehungen zwischen Zuständen und Übergängen untereinander ergeben, ist $\mathcal{W}*FLOW$ so realisiert worden, daß sich alle vier unterschiedlichen Beziehungstypen (S/T, T/T, S/S und T/S) modellieren lassen.

$\mathcal{W}*FLOW$ erlaubt die Realisierung dieser Beziehungstypen unter Zuhilfenahme des auf dem Signal–Slot Mechanismus basierenden Ereignismechanismus. Dazu sendet jede Aktivitäteninstanz bei der Annahme eines neuen Zustandes S ein entsprechendes Ereignissignal[27] aus. Analog dazu wird bei jedem Übergang T verfahren[28]. Jede Beziehung zwischen Zuständen und/oder Übergängen läßt sich nun, basierend auf den Mechanismen aus Abschnitt 2.3.2 und speziell dem Codebeispiel 2.3.1, wie folgt ausdrücken[29]:

S/T Beziehungstyp:

```
$a1->connect('S_'.$source_state, $a2, &$dest_trans());
```

T/T Beziehungstyp:

```
$a1->connect('T_'.$source_trans, $a2, &$dest_trans());
```

In diesen beiden Fällen wird das Auftreten eines Ereignisses (Signalisierung eines Zustandes oder eines Übergangs) einer Aktivität $a1 mit der Ausführung einer Methode $dest_trans der Aktivität $a2 verknüpft. Der Name der auszuführenden Methode entspricht dabei dem Namen des Übergangs, so wie er in Beispiel 3.2.2 mittels des Bezeichners NAME angegeben wurde.

Die beiden andern Beziehungstypen lassen sich wie folgt realisieren:

S/S Beziehungstyp:

```
$a1->connect('S_'.$source_sig, $a2, is_in_state($dest_state) ||
                                    &$handler());
```

T/S Beziehungstyp:

[27] Hierbei handelt es sich um den Prefix S_ gefolgt vom Name des Zustandes.

[28] In diesem Fall wird der PrefixT_, gefolgt vom Name des Übergangs ausgegeben. Die Signalisierung eines Überganges erfolgt standardmäßig vor der Ausführung des Aktionshandlers. Es ist aber auch möglich, andere Signalisierungspunkte, entsprechend den Übergängen aus Abbildung 3.15, anzugeben.

[29] Folgende Annahmen werden bei den betrachteten Codebeispielen gemacht: $a1, $a2 sind $\mathcal{W}*FLOW$–Aktivitäten, $source_state, $source_trans sind Zustände/Übergänge, die von der Aktivität $a1 signalisiert werden, $dest_trans, $dest_state bezeichnen die auf $a2 anzuwendende Methode, bzw. den Zustand in dem sich die Aktivität befinden muß.

```
$a1->connect('T_'.$source_trans, $a2, is_in_state($dest_state) ||
                                      &$handler());
```

In den Fällen, in denen die Beziehungen eine deklarative Vorgabe beschreiben, wird ähnlich wie in den oberen zwei Fällen verfahren. Im Unterschied zu diesen wird beim Eintreffen eines Ereignisses aber kein Übergang in der Aktivität `$a2` ausgelöst, sondern es wird der folgende Ausdruck ausgeführt:

```
is_in_state($dest_state) || &$handler());
```

Hierbei handelt es sich um eine Disjunktion von zwei Perl-Anweisungen. Mittels der Methode is_in_state(...) wird in einem ersten Schritt überprüft, ob sich die Aktivitäteninstanz in dem spezifizierten Zustand befindet. Ist das der Fall, so liefert die Methode den Wert **TRUE** zurück und die Disjunktion evaluiert zu **TRUE**. In dem Fall wertet *Perl* den zweiten Ausdruck nicht aus. Im Falle, daß die Methode **FALSE** zurückliefert, wird der zweite Ausdruck, eine Fehlerbehandlungsroutine, aufgerufen. Deren Aufgabe ist es dann, die Regelverletzung zu behandeln oder auch nur anzuzeigen.

Um zusätzliche, zur Laufzeit auszuwertende Transitionsbedingungen (TB) zu integrieren, können die zu evaluierenden TB's beispielsweise in eine Funktion oder Methode gepackt und diese Funktion mittels Konjunktion (`&&`) dem letzten Parameter des connect(...) Aufrufes vorangestellt werden. Beispielsweise könnte eine Anweisung folgendermaßen lauten:

```
$a1->connect('S_'.$source_state, $a2, tb() && &$dest_trans());
```

In diesem Fall enthält tb() die Transitionsbedingung.

Das Hauptergebniss des Abschnitts läßt sich somit folgendermaßen zusammenfassen:

Ergebnis:

Um einen kompletten Workflow, einschließlich Kontrollfluß zu definieren, wird folgendes benötigt:

- Eine Menge von Aktivitäten, welche die auszuführenden Tätigkeiten spezifizieren.
- Eine Menge der unterschiedlichen Beziehungen zwischen den Zuständen und/oder Übergängen der Aktivitäten, die mittels des Signal–Slot Mechanismus miteinander kommunizieren.

3.2.4.2 Modellierung funktionsbezogener Aspekte

Nachdem im letzten Abschnitt die Realisierung *verhaltensbezogener Aspekte* diskutiert und anhand von Beispielen illustriert wurde, sollen in diesem Abschnitt *funktionsbezogene Aspekte* besprochen werden, die maßgeblich von den Aktivitäten beeinflußt werden. Der *funktionsbezogene Aspekt* befaßt sich, neben der Festlegung der Funktion eines Workflows, mit dem Aufbau von Workflow-Schemata, d. h. der Bereitstellung von Bausteinen zum Aufbau komplexer, hierarchischer Arbeitsabläufe.

Im folgenden soll, um einen Überblick zu geben, eine Reihe unterschiedlicher Ansätze kurz vorgestellt werden; eine ausführliche Betrachtung ist in [BMWS96] zu finden. Die Unterschiede liegen hauptsächlich darin, welche unterschiedlichen Bausteintypen zum Aufbau von Workflowhierar-

chien eingesetzt werden [Bö97]. Abbildung 3.21 beinhaltet die Legende der unterschiedlichen Bausteintypen für die sich anschließenden Abbildungen 3.22 bis 3.25.

Abbildung 3.21: *Legende der Bausteintypen*

Der einfachste Typ von Hierarchie ist durch zwei explizite Bausteintypen definiert, die beide eine fest vorgegebene Funktionalität besitzen. Hierbei handelt es sich um den *Prozeß*, der den Workflow als Ganzes repräsentiert, dem beliebig viele atomare *Aktivitäten* als zweiten Bausteintyp zugeordnet werden, womit eine zweistufige Hierarchie aufgebaut werden kann. Das führt aber schnell zu sehr unübersichtlichen Workflows, da die Aktivitäten alle auf der selben Hierarchiestufe stehen. Dieser Sachverhalt ist in Abbildung 3.22 zu sehen.

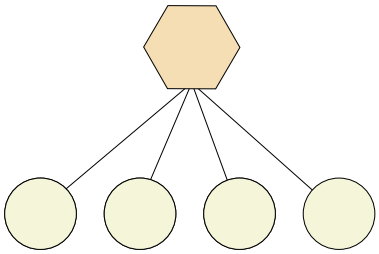


Abbildung 3.22: *Zwei Bausteintypen*

Eine Erweiterung dieses Ansatzes wird z. B. in *FlowPath* [Bul93] realisiert. Hierbei wird neben den beiden obigen Bausteintypen noch der Bausteintyp *Block* eingeführt, innerhalb dem der Bausteintyp *Aktivität* beliebig oft auftauchen kann. Mehrere *Aktivitäten* können somit zu einem *Block* zusammengefaßt werden, was etwas übersichtlichere Workflows ergibt, deren Hierarchie auf drei Stufen beschränkt ist. Abbildung 3.23 zeigt eine derartige Funktionsstruktur.

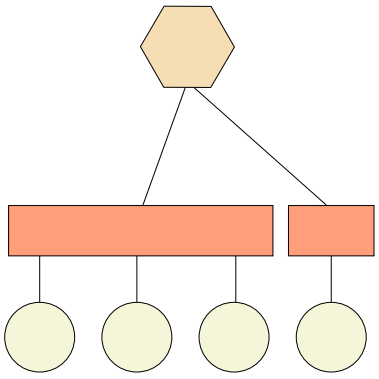


Abbildung 3.23: *Drei Bausteintypen*

Die WFMC favorisiert einen Ansatz, bei dem ebenfalls drei unterschiedliche Bausteintypen vorkommen, diese aber einen rekursiven Aufbau erlauben. Bei den einzelnen Bausteinen handelt es sich um den *Prozeß* als obersten Baustein einer Hierarchie, den *Block* als mittleren Baustein und die *Aktivität* als unterste Stufe in einer Baumhierarchie. Wie bei dem Ansatz von *FlowPath* können hier mehrere *Aktivitäten* im Rahmen eines Blocks zusammengefaßt werden. Im Unterschied zu diesem Ansatz ist es aber auch möglich nicht nur *Aktivitäten* sondern auch

Blöcke durch einen weiteren *Block* zusammenzufassen, was einen rekursiven Aufbau erlaubt. Abbildung 3.24 zeigt eine derartig aufgebaute Hierarchie.

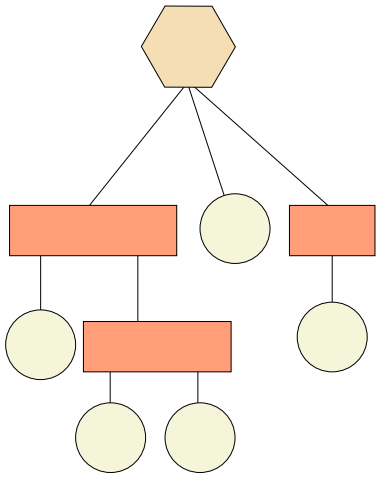


Abbildung 3.24: *Drei Bausteintypen mit rekursivem Aufbau*

Ein weiterer interessanter Ansatz, der im Workflowsystem $\mathcal{MOBILE}$ realisiert wurde, geht von nur einem Bausteintyp aus, der aber entsprechend seiner Einordnung in die Funktionsstruktur unterschiedliche Bedeutung hat. So wird dieser Bausteintyp, sowohl für den Workflow als Ganzes als auch für eine einzelne elementare Aktivitäten eingesetzt. Ein Beispiel ist in Abbildung 3.25 abgebildet.

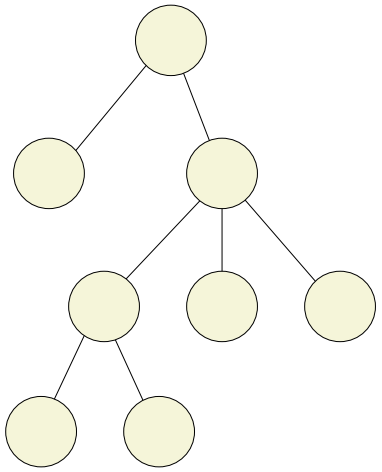


Abbildung 3.25: *Ein Bausteintyp mit situationsbezogener Typisierung*

$\mathcal{W}*FLOW$ führt, im Gegensatz zu den meisten obigen Ansätzen, neben den $\mathcal{W}*FLOW$–Aktivitäten keine weiteren Bausteintypen zum Aufbau von hierarchischen Workflows ein. Der Grund liegt darin, daß diese Realisierungen alle bereits sehr speziell sind und den Entwickler bei der Realisierung eines eigenen Workflowsystems sehr einengen würden. Als weiteres Konzept kennt $\mathcal{W}*FLOW$ lediglich noch die Gruppierung von Aktivitäten innerhalb einer Datenbank, sowie die Referenzierung von Aktivitäten. Es ist dem Entwickler natürlich freigestellt sich eigene Bausteintypen zu entwickeln bzw. aus den bereits vorhandenen *Aktivitäten* neue Bausteintypen abzuleiten. Dies soll an einem einfachen Beispielszenario gezeigt werden. Im folgenden Beispiel wird, allein mit den $\mathcal{W}*FLOW$–Aktivitäten, ein hierarchisches Workflowsystem aufgebaut. Die

vorgestellte Lösung gleicht dem Ansatz von $\mathcal{MOBILE}$, das ebenfalls nur einen Bausteintyp einsetzt.

Wie am Ende des vorherigen Abschnittes über die verhaltensbezogenen Aspekte herausgearbeitet, besteht ein kompletter Workflow aus einer Menge von Aktivitäten sowie einer Menge von Beziehungen zwischen diesen Aktivitäten. Es wurde weiterhin in Abschnitt 3.2.3 gezeigt, wie die Aktivitäten mittels der $\mathcal{W}$*FLOW–API erzeugt und konfiguriert werden können und wie die Beziehungen zwischen ihnen aufgebaut werden.
So kann nun eine Aktivität (Vateraktivität) generiert werden, deren Instanzen beim Übergang vom Zustand Initialisiert nach In-Arbeit einen internen Aktionshandler aufrufen. Im Rahmen des Aktionshandlers wird mittels der $\mathcal{W}$*FLOW–API eine Reihe von weiteren Aktivitäteninstanzen sowie der zugehörige Kontrollfluß erzeugt. Durch die Aktivierung der Vaterinstanz wird somit ein Subworkflow, bestehend aus einer Menge von Aktivitäteninstanzen, sammt Beziehungen, erzeugt. Die mittels des internen Handlers erzeugten $\mathcal{W}$*FLOW–Aktivitäteninstanzen können als $\mathcal{W}$*FLOW–Attribute in der Vaterinstanz abgelegt werden, umgekehrt enthält jede so erzeugte Aktivitäteninstanz eine Referenz auf ihre Vaterinstanz. Das Verhalten der Vateraktivität kann beispielsweise durch entsprechende Vorbedingungen beim Übergang in einen der Finalzustände Abgeschlossen oder Abgebrochen definiert werden. Die Allokation der benötigten Ressourcen wird durch die Vateraktivität realisiert, welche die Ressourcen dann ihren Unteraktivitäten zur Verfügung stellen kann. Da es sich bei der Vateraktivität wiederum um eine $\mathcal{W}$*FLOW–Aktivität handelt, können Workflowhierarchien beliebiger Schachtelungstiefe aufgebaut werden. Weitere interessante Aspekte im Rahmen von hierarchischen Aktivitäten ergeben sich durch unterschiedliche Gültigkeitsbereiche von $\mathcal{W}$*FLOW–Attributen und deren Zugriffsmethoden, (aktivitätenintern, sichtbar in Subaktivitäten, öffentlich sichtbar, etc.). Abbildung 3.26 veranschaulicht das hier geschilderte Szenario unter dem Aspekt der Integration von internen Handlern und Bedingungen.

Ein analoger Ablauf wird bei $\mathcal{MOBILE}$, entsprechend Abbildung 3.27, mit Ausführungsanweisungen modelliert. Die Ablaufsteuerung (verhaltensbezogene Aspekte), die in $\mathcal{MOBILE}$ in den oberen drei Anweisungen mittels Petri–Netzen festgelegt ist, wird in $\mathcal{W}$*FLOW durch die Verwendung von internen Handler und Vor-/Nachbedingungen modelliert.
Es soll an dieser Stelle nochmals betont werden, daß es sich bei $\mathcal{W}$*FLOW und $\mathcal{MOBILE}$ um Systeme mit unterschiedlichen Zielrichtungen handelt. Während es sich bei $\mathcal{MOBILE}$ um ein komplettes Workflowsystem handelt, das z. B. den Kontrollfluß mittels Petri–Netzen definiert, handelt es sich bei $\mathcal{W}$*FLOW um einen Baukasten, dessen Schnittstelle unterhalb der von verfahrensspezifischen Elementen, wie etwa Petri–Netzen, liegt. Obige Ausführungen zeigen jedoch, daß es bereits mittels den von $\mathcal{W}$*FLOW zur Verfügung gestellten Komponenten möglich ist, komplexe hierarchische Workflows aufzubauen.

3.3 Zusammenfassung

Im vorliegenden Kapitel wurden die beiden Basiskomponenten des $\mathcal{W}$*FLOW–Baukastens, die *Container* und die *Aktivitäten* vorgestellt.

Abschnitt 3.1 befaßte sich mit der Konzeption und Realisierung einer Datenhaltungskomponente. Ausgehend von der Vorstellung der speziellen Anforderungen, wie sie von den beiden Randbedingungen, den Einsatz im Rahmen eines Workflowsystems und der Eignung für die Verwaltung wissenschaftlicher Datenbestände, vorgegeben sind, wurden eine Reihe von Ansätzen, wie sie in der Fachliteratur zu finden sind, gesichtet und auf ihre Eignung hin untersucht. Ein Kernpunkt der anschließend konzipierten und realisierten *Containerkomponente* ist ihre *Middle–Tier*

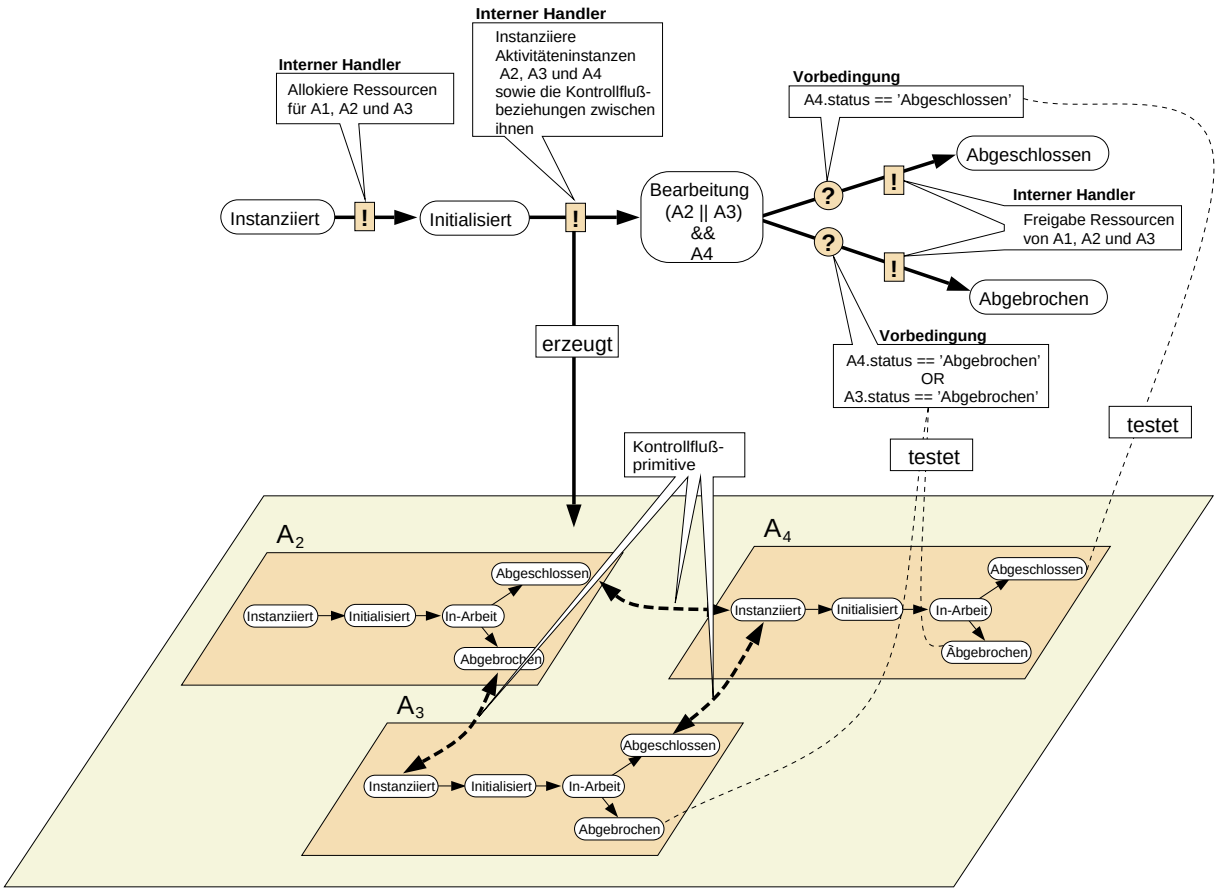


Abbildung 3.26: *Hierarchische Organisation von W∗FLOW–Aktivitäten*

Eigenschaft, die eine einheitliche (Zugriffs–) Schnittstelle zu den unterschiedlichen Datentypen/–quellen bereitstellt, unabhängig vom Format und/oder Speicherort. Ein weiterer wichtiger Punkt beinhaltet die Möglichkeit, die Daten auf "semantisch hohen Niveau" zu speichern. Dies wird durch die Einführung von Metainformationen, welche die Daten beschreiben, und die Möglichkeit der Strukturierung der Datensätze, erreicht. Hierbei wird besonders auf flexible Mechanismen geachtet, die keine starre Strukturierung der Daten vorsehen, sondern eine dynamische Restrukturierung auch zur Laufzeit ermöglichen.
Die notwendige Datenbankfunktionalität wie Transaktionsverwaltung, Suchanfragen und Verteilung im Netz wird unter Zuhilfenahme der unterhalb der *Container* liegenden *ObjectStore* Datenbank realisiert.

Anschließend wurden in Abschnitt 3.2 die eigentlichen Funktionsträger eines Workflows, die Aktivitäten, vorgestellt. Ausgehend von einer Untersuchung des Begriffs der Aktivität anhand der Literatur zeigte sich, daß der Begriff keine einheitliche Definition besitzt, sondern ein weites Spektrum an möglichen Realisierungen aufweist. Eine sich anschließende Untersuchung von Workflows aus dem wissenschaftlich–technischen Bereich in Bezug auf ihren strukturellen Aufbau zeigte, daß eine starke Vernetzung zwischen einzelnen Arbeitsschritten, die auf denselben Daten arbeiten, auftritt, während ansonsten einfache Beziehungen auf Daten– und Kontrollfluß-Ebene vorherrschen. Diese Erkenntnisse führten zur Definition eines erweiterten Aktivitätenmodells, das als Hauptcharakteristik eine Unterteilung in die drei Phasen Ressourcenallokation, Ressourcenbearbeitung und Ressourcenfreigabe aufweist. Eine Abbildung der drei Phasen auf

```
CONTROL_FLOW_CONSTRUCT
      alternate(WORKFLOW: A, B)
        /* Petri Net Specification */
END_CONTROL_FLOW_CONSTRUCT

CONTROL_FLOW_CONSTRUCT
      sequence (WORKFLOW: A, B)
        /* Petri Net Specification */
END_CONTROL_FLOW_CONSTRUCT

CONTROL_FLOW_CONSTRUCT
      ifthen (FUNCTION: condition; WORKFLOW: A, B)
        /* Petri Net Specification */
END_CONTROL_FLOW_CONSTRUCT
```

```
WORKFLOW_TYPE WF_1 (...)
  SUBWORKFLOWS
      A2: WF_2;
      A3: WF_3;
      A4: WF_4;
  END_SUBWORKFLOWS
  ...
  CONTROL_FLOW
      sequence (alternate(A2,A3),
                A4)
  END_CONTROL_FLOW
END_WORKFLOW_TYPE
```

Abbildung 3.27: *Workflow Schema in $\mathcal{MOBILE}$*

die Zustände eines frei definierbaren Zustandsautomaten erlauben die einfache Definition zusammenhängender Arbeitsschritte. Die auszuführenden Arbeitsschritte sind hierbei an die Zustandsübergänge des Automaten geknüpft und können sowohl innerhalb der *Workflow-Engine* als auch extern ablaufen. Ein flexibler Mechanismus zur Formulierung von Bedingungen, sowohl auf Basis von Workflow– als auch Applikationsdaten und externen Ereignissen, erlaubt die Definition komplexer Workflow-Semantiken. Die $\mathcal{W}*FLOW$–Aktivität stellt somit einen allgemeinen Schutzmechanismus zur Klammerung von Arbeitsschritten und Kapselung der benötigten Daten bereit und ist die Basis eines Transaktionsmechanismus. Sie erlaubt die Definition komplexer, aus internen und externen Arbeitsschritten bestehenden Arbeitsabläufe und steuert und sichert den kooperierenden/konkurrierenden Ablauf von Operationen verschiedener Benutzer auf den Daten.

Kapitel 4

Toolservices

4.1 Einführung

Aufgabe der im vorliegenden Kapitel vorgestellten Komponenten ist die Einbindung *Workflow externer Anwendungen* in den Workflowablauf. Diese, dem operationsbezogenen Aspekt zuzurechnende Aufgabe, zählt zu den kritischsten Erfolgsfaktoren im Workflow Bereich [SJHB96], stellen doch die externen Anwendungen, auch nach Einführung eines Workflow-Management-Systems, die wichtigsten Funktionsträger dar [Bö98, OV96].

Umso verwunderlicher ist es, daß dieser Aspekt bei heutigen kommerziellen Systemen nur rudimentäre Unterstützung findet, wie Untersuchungen in [JET97, TG97] zeigen. So zeigten auch Erfahrungsberichte aus unterschiedlichen Branchen[1], die im Rahmen einer Konferenz über workflowbasiertes Prozeßmanagement Anfang 1998 vorgestellt wurden, daß die Integration von vorhandener Software das Hauptproblem bei der Einführung eines WFMS darstellt [Eur98].

Ziel ist die Integration von Anwendungen in ein Workflowsystem. Der Begriff "Integration" läßt aber eine Reihe von Möglichkeiten offen, wie die Anwendungen an das System angebunden werden können. Sie hängen sowohl von der anzubindenden Anwendung als auch vom gewünschten Integrationsgrad ab. Im einzelnen kann zwischen folgenden Anbindungsarten unterschieden werden [VK96]:

1. *Whitebox*–Integration: Hierbei handelt es sich um die engste Ankopplung zwischen Anwendung und Workflowsystem. Die Anwendung nutzt zu diesem Zweck eine vom Workflowsystem zur Verfügung gestellte API. Soche Anwendungen werden speziell für den Einsatz im Rahmen eines Workflowsystems entwickelt bzw. auf Quellcode-Ebene angepaßt. Es wird hierbei auch von *Workflow konformen* Anwendungen gesprochen.

2. *Greybox*–Integration: Das zu integrierende System besitzt eine API, die vom Workflowsystem genutzt werden kann. Ein Beispiel hierfür ist die Nutzung von DDE[2]-Mechanismen zum Datenaustausch zwischen Windows Applikationen.

3. *Blackbox*–Integration: Das anzubindende Werkzeug liegt als ausführbare Datei vor, die nichts von der Existenz des Workflowsystems weiß und mittels Betriebssystemaufruf gestartet wird.

[1] Telekommunikation, Maschinenbau, Energiewirtschaft, Kreditinstitute, Transport, etc.

[2] **D**ynamic **D**ata **E**xchange (siehe Glossar Seite 180)

Bevor in Abschnitt 4.3 auf die Anforderungen an die zu realisierenden Komponenten eingegangen wird, werden im folgenden eine Reihe von Realisierungen bestehender Workflowsysteme sowie Lösungsvorschläge aus der Literatur vorgestellt.

4.2 Existierende Realisierungen und Lösungsansätze

Zu Beginn werden zwei kommerzielle Systeme, die eine weite Verbreitung gefunden haben, erläutert.

Das System *WorkParty* stammt von Siemens Nixdorf und wird bevorzugt im Geschäfts– und Büroumfeld eingesetzt. *WorkParty* unterstützt sowohl die *Whitebox*– als auch *Blackbox*–Integration. Die *Whitebox*–Integration erfordert die Einbindung einer Windows Bibliothek in die zu realisierende Anwendung. Die so aufgebauten Programme können dann mittels der zur Verfügung stehenden API auf Eingabeparameter und -daten zugreifen und eventuell erzeugte Ergebnisse zurückschreiben. Bei der *Blackbox*–Integration ist die Übergabe von Parametern nicht möglich, was den Nutzen der Einbindung stark reduziert. Es findet eine 1:1 Abbildung zwischen einem Arbeitsschritt[3] und der auszuführenden Anwendung statt, so daß dem Anwender keine Flexibilität bei der Programmauswahl zugestanden wird.

FlowMark von IBM erfordert, daß die im Rahmen eines Workflows eingesetzten Programme zuvor registriert werden. Bei der Registrierung werden Informationen über das auszuführende Programm angegeben, so z. B. auf welchem Rechner es ablaufen soll, wie es aufgerufen wird und welche Parameter übergeben werden müssen. Durch die Registrierung ist es möglich, bei der Definition von Workflows von den Aufrufspezifika zu abstrahieren und einen symbolischen Namen, der ebenfalls bei der Registration mit angegeben wird, zu verwenden. Neben der *Blackbox*– wird auch eine *Whitebox*– und *Greybox*–Integration unterstützt. Wie bei *WorkParty* wird aber auch lediglich eine 1:1 Abbildung zwischen Arbeitsschritt[4] und Programm unterstützt.

Beide Systeme weisen durch die unflexible Zuordnung von Arbeitsschritt zu Programm erhebliche Schwächen beim Einsatz im heterogenen Umfeld auf, da die 1:1 Zuordnung von Arbeitsschritt zu Programm auch zu einer festen Zuordnung zwischen Arbeitsschritt und Zielrechner führt, was insbesondere bei der entfernten Ausführung eines Programms zwischen nicht-kompatiblen Rechnersystemen zu unlösbaren Problemen führen kann.

Die Workflow Management Coalition hat die Problematik des Workflow Einsatzes in heterogenen Umgebungen erkannt [WFM98c] und aus diesem Grund in der neuesten Version der Schnittstellenbeschreibung [WFM98e] für den Aufruf externer Anwendungen den Einsatz eines sogenannten *ToolAgent* vorgeschlagen, der als Bindeglied zwischen Workflowsystem und externer Anwendung fungiert. Aufgabe des *ToolAgent* ist es, zum einen eine einheitliche Schnittstelle zum Workflowsystem zur Verfügung zu stellen und andererseits, auf der Seite zur Anwendung hin, verschiedene Kommunikationsmechanismen (Betriebsystemschnittstellen, DDE, CORBA, OLE, ...) bereitzustellen. Die WFMC bezeichnet den *ToolAgent* als eine Art Software Treiber, vergleichbar einem ODBC Treiber [WFM98e]. Weiterhin besteht bei der Definition von Workflowschemata seit neuem die Möglichkeit, symbolische Namen für einzelne Anwendungen zu vergeben (siehe auch *FlowMark*). Dieser kann dann zur Laufzeit von einem zugehörigen ToolAgent in den konkreten Aufrufstring umgesetzt werden. Hiermit ist erstmals eine $1 : n$ Beziehung zwischen Arbeitsschritt und aufzurufender Anwendung möglich, was die Flexibilität und Einsatzbreite der Systeme erhöht. Abbildung 4.1 zeigt den Aufruf von Anwendungen von einem Workflowsystem

[3] In *WorkParty* wird der Ausdruck *Task* verwendet.
[4] IBM verwendet hierfür den Begriff *Programmaktivität*.

sowohl mit als auch ohne *ToolAgent*.

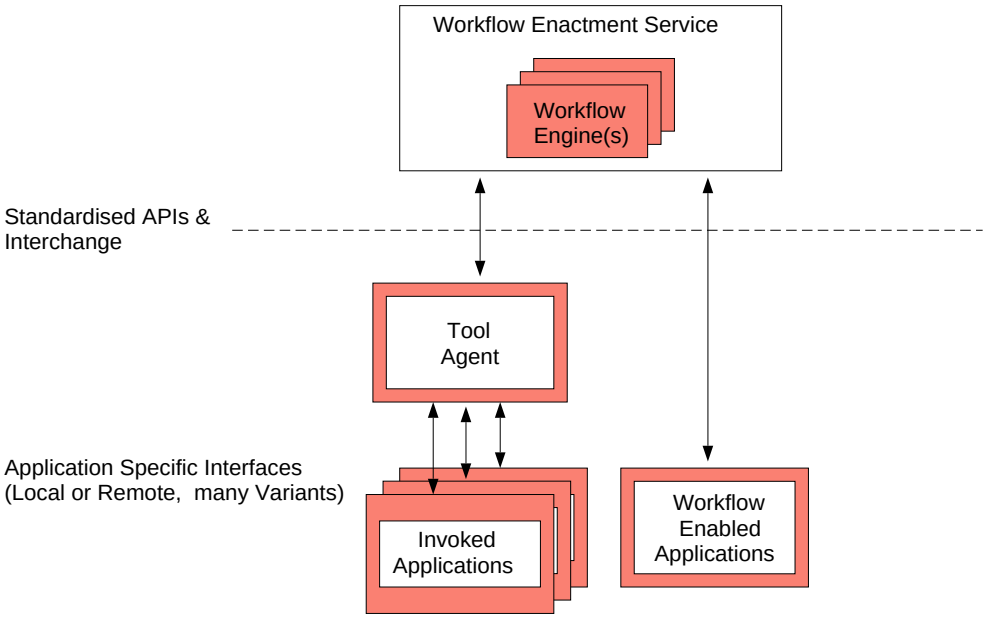


Abbildung 4.1: *Einsatz eines* ToolAgent *im Rahmen des Applikationsaufrufs [WFM98e].*

Die Dokumentation der WFMC macht hingegen keinerlei Aussagen über folgende Punkte:

- Wie erfolgt die Auswahl eines Programms?
- Wie geschieht die Zuordnung von *Workflow-Engine* und *ToolAgent* sowie *ToolAgent* und Anwendung?
- Wie erfolgt eine Umleitung von Ausgaben bei entfernt laufenden Programmen?[5]

Zum Abschluß wird nun noch, als Vertreter aus dem Forschungsbereich, der operationsbezogene Aspekt des Workflowsystems $\mathcal{MOBILE}$ aus Abschnitt 1.2.2 vorgestellt. $\mathcal{MOBILE}$ definiert als Abstraktionsobjekt zur konkreten Anwendung die *Workflow Applikation*. Eine *Workflow Applikation* besteht aus einer normalen Anwendung und einem Wrapper darum.

$$Applikation + Wrapper = Workflow\ Applikation$$

Der Wrapper realisiert, analog zum *ToolAgent* der WFMC, eine einheitliche Schnittstelle zum Workflowsystem und verbirgt die Eigenheiten der Anwendung, wie genaue Aufrufkonvention, Betriebssystembesonderheiten, Parameterübergabe, Fehlersignalisierung, etc.

Einer *Workflow Applikation* können mehrere Anwendungen zugeordnet sein. So kann der *Workflow Anwendung* EDITOR beispielsweise die populären *UNIX*-Editoren *EMACS* [Gos84, Sta87] und *VI* [CL92] sowie der Windows Editor *NOTEPAD* [HW95] zugeordnet sein. Diesen Anwendungen sind wiederum sogenannte Anwendungsinstanzen auf den einzelnen Rechnern zugeordnet, welche dann den konkreten Aufruf beinhalten. Wird im Rahmen eines Workflowschemas

[5] In diesem Punkt gibt es momentan Anstrengungen das Problem herstellerübergreifend zu lösen. Das *Datorr* (**D**esktop **a**ccess **to** **r**emote **r**esources) Projekt [Dat99] beschäftigt sich mit der Frage des Zugriffs auf entfernte Ressourcen wie Supercomputer, Netzwerke, intelligente Instrumente und Daten.

die *Workflow Applikation* EDITOR gestartet, so wird bei der Abarbeitung des entsprechenden Arbeitsschrittes untersucht, ob Anwendungsinstanzen für die Anwendungen *EMACS*, *VI* und *NOTEPAD* auf dem zur Ausführung spezifizierten Rechner verfügbar sind. Ist dies der Fall, so können all die gefundenen Anwendungsinstanzen für eine Ausführung herangezogen werden. Eine konkrete Abbildungsvorschrift wird in $\mathcal{MOBILE}$ jedoch nicht gegeben. Möglichkeiten sind hier eine Auswahl nach Rechnerauslastung oder eine manuelle Benutzerauswahl.
Das vorgestellte Konzept ist, im Vergleich zu den Mechanismen die von den beiden kommerziellen Systemen zur Verfügung gestellt werden, weitaus flexibler und speziell auf den Einsatz in heterogenen Umgebungen ausgerichtet. Zudem unterstützt es die Forderung nach Flexibilität bei der Auswahl durch den Anwender.

Nach diesem Überblick über existierende Mechanismen zur Anbindung externer Anwendungen sollen im folgenden die Anforderungen an die $\mathcal{W}*FLOW$–Komponenten zur Toolintegration vorgestellt werden.

4.3 Definition der Anforderungen

Eine der Aufgaben des operationsbezogenen Aspektes ist, eine Beziehung zwischen einem Arbeitsschritt, so wie er im Rahmen eines Workflowschemas definiert wurde, und einem ausführbaren Programm herzustellen. Im einfachsten Fall kann das durch eine direkte Eintragung des entsprechenden Programms für den Arbeitsschritt realisiert werden. Der Ansatz hat jedoch eine Reihe von Nachteilen:

1. Es muß bereits bei der Definition des Workflows die genaue Aufrufsyntax für das zu startende Programm bekannt sein. Diese Information ist zum Definitionszeitpunkt eines Workflows aber irrelevant, da die Beschreibung des Arbeitsablaufs auf einer semantisch höheren Ebene erfolgt als die Formulierung von Aufrufsequenzen. Weiterhin handelt es sich bei den Personen, die die Arbeitsabläufe definieren, zumeist nicht um Computerexperten. Sie verfügen im allgemeinen nicht über die notwendigen Detailkenntnisse der einzubindenden Softwarepakete.

2. Der Ansatz scheitert, wenn nicht auf allen eingesetzten Rechnern die Programme an derselben Stelle liegen, bzw. mittels Pfadinformationen lokalisiert werden können. Kann das bei kleinen Workflowsystemen, die innerhalb einer homogenen Hard– und Software-Umgebung ablaufen, eventuell noch gefordert werden, funktioniert die Vorgehensweise bei unterschiedlichen Hard– und Software-Umgebungen nicht. In diesem Fall muß schon zum Definitionszeitpunkt eines Workflows der Rechner spezifiziert werden, auf dem eine Anwendung später ablaufen soll, bzw. es müssen innerhalb des Arbeitsschrittes für alle in Frage kommenden Rechner die jeweiligen Aufrufsequenzen angegeben werden.

3. Die Festlegung auf ein bestimmtes Programm bietet keinerlei Flexibilität hinsichtlich Alternativen. Durch die Festschreibung eines bestimmten Programms sinkt aber die Akzeptanz eines Workflowsystems in den Augen derjenigen Benutzer, die nun gezwungen werden, für eine bestimmte Aufgabe ein neues Programm zu benutzen.

4. Beim Wechsel eines Programms müssen in allen betroffenen Workflows die "hardkodierten" Einträge der Arbeitsschritte an das neue Programm angepaßt werden.

Um diese nicht zu tolerierenden Einschränkungen zu umgehen, muß, wie auch in den Ansätzen der WFMC und $\mathcal{MOBILE}$ vorgestellt, von der konkreten Aufrufsyntax abstrahiert werden [SJHB96, HS97].

Anforderung 1:

Um eine Reihe von nicht tolerierbaren Einschränkungen bei der Anbindung externer Anwendungen zu vermeiden, muß eine Trennung der Beschreibung einer Toolaktion im Rahmen der Workflowdefinition und der konkreten Programmaufrufsyntax erfolgen.

Ziel der Abstraktion ist, im Rahmen der Workflow-Schema-Definition Arbeitsschritte beschreiben zu können, ohne auf rechnerspezifische Besonderheiten eingehen zu müssen und dem Benutzer weiterhin den Einsatz der gewohnten Programme zu ermöglichen.

Damit der Benutzer die ihm übertragenen Aufgaben von seinem Arbeitsplatz aus erfüllen kann, entsteht weiterhin die Anforderung, daß sowohl ein lokaler als auch entfernter Programmaufruf im heterogenen Netz unterstützt wird. Die Gründe, welche für die Ausführung auf einem entfernten Rechner sprechen, sind dabei folgende:

- Ein Programm ist an ein bestimmtes Betriebssystem gebunden.
- Die Programmlizenz ist rechnergebunden.
- Das Programm benötigt spezielle Zusatzhardware.
- Der Clientrechner ist nicht leistungsfähig genug.
- Eine gleichmäßige Lastverteilung auf mehrere Rechner ist nötig.
- Das Programm soll dort laufen, wo sich die Daten befinden. Es kann sinnvoll sein, das Programm auf dem Rechner zu starten, auf welchem die benötigten Daten vorliegen, da so ein möglicherweise kostspieliger Datentransport zum Clientrechner unterbleiben kann.

Anforderung 2:

Eine Toolstarterkomponente für das wissenschaftlich–technische Umfeld muß mit heterogenen Rechnernetzen umgehen können und den Start eines Tools lokal und entfernt unterstützen.

Um die Programme starten zu können, müssen diese mit den notwendigen Parametern und Eingabedaten versorgt werden. Daneben muß auch der Transport von Ergebnissen und workflowrelevanten Informationen, wie z. B. Meldungen über Erfolg oder Mißerfolg eines Programmaufrufs, realisiert werden.

Anforderung 3:

Eine Toolstarter-Komponente muß den automatischen Transport von Eingabeparametern und Parametern sowie Ergebnissen unterstützen.

Neben obigen Anforderungen findet sich in der Literatur (z. B.: [SJHB96, GHS95]) oft noch die Forderung nach Unterstützung von entfernten (transaktionalen) Prozeduraufrufen (RPC) sowie CORBA Anbindung. Die Unterstützung von RPC–Mechanismen ist in $\mathcal{W}*FLOW$ durch die Möglichkeit realisiert, für die Arbeitsschritte statt einer abstrakten Methodenbeschreibung auch

Methoden an die *Aktionshandler* zu binden (siehe Abschnitt 3.2.3). Durch die netzwerktransparente API wird hier die entfernte Ausführung von Prozeduren oder Methoden mit abgedeckt. Verteilte Transaktionen und eine Anbindung an CORBA (siehe dazu auch Abschnitt 2.2.2.3) sind noch nicht realisiert. Eine ausführliche Abhandlung von transaktionalen Mechanismen und CORBA Unterstützung findet sich in [Sch97].

Abschnitt 4.1 nennt drei verschiedene Arten der Programmintegration. Als weitere Forderung für die *Toolservices* ergibt sich somit:

Anforderung 4:

Die *Toolservice* Komponenten müssen, neben der gebräuchlichen *Blackbox*–Integration, auch eine *Whitebox*– und *Greybox*–Integration unterstützen.

Zusätzlich wird die Forderung nach einer formalisierbaren Auflösung der konkreten Anwendung zur Laufzeit gestellt. Bei der Sichtung der Literatur zu diesem Themenkomplex konnten hierzu keine geeigneten Ansätze gefunden werden und auch beim Ansatz von $\mathcal{MOBILE}$ erfolgt nur eine semiformale Auflösung [JB96]. In [Bö98] wird die Auswahlproblematik ebenfalls angesprochen, konkrete Lösungsmöglichkeiten aber nicht vorgestellt, sondern nur die Notwendigkeit formalisierbarer Abbildungsregeln und die Abstraktion von konkreten Anwendungen als Grundvoraussetzung genannt. Die Forderung nach Auflösung der abstrakten Beschreibung hängt eng mit der nach Flexibilität zusammen. So ist es für einen Anwender zwar erstrebenswert, Auswahlmöglichkeiten bei den Programmen zu haben, es ist für ihn aber lästig, die Auswahl jedesmal wieder durchzuführen. Ziel der Anforderung ist also, einen formalisierbaren Abbildungsmechanismus zu finden, der auch den Benutzer mit einbezieht.

Anforderung 5:

Aufbauend auf dem Konzept der abstrakten Toolbeschreibung soll ein voll formalisierbarer, flexibler Abbildungsmechanismus zwischen abstrakter Beschreibung und konkretem Programmaufruf realisiert werden.

4.4 Konzeption der neuen Lösung

In diesem Abschnitt wird eine Konzeption vorgestellt, die bei der Beschreibung der auszuführenden Arbeitsschritte von rechnerspezifischen Details abstrahiert und eine Beschreibung der auszuführenden Programme auf einer höheren Ebene erlaubt. Aufbauend auf diesem Konzept wird ein Service realisiert, welcher es in flexibler Weise gestattet, den auf höherem Level definierten Aufgaben, in Abhängigkeit verschiedener Faktoren, unterschiedliche Programme zuzuordnen.

Die von den Toolservices bereitgestellten Dienste sollen in Anlehnung an Abschnitt 2.2.1 derart konzipiert und realisiert werden, daß sie eine eigenständige Einheit bilden und nicht nur im Rahmen der $\mathcal{W}*FLOW$–Komponenten eingesetzt werden können, sondern einen allgemeinen Mechanismus zur Definition und Ausführung von bestimmten Aufgaben auf verteilten, heterogenen Rechnerarchitekturen darstellen.

Dazu wird zuerst eine abstrakte Beschreibung für konkrete Applikationen und Kommandos eingeführt. Anschließend wird das Gegenstück der abstrakten Beschreibung, die *Laufzeitbeschreibung*, vorgestellt (Abschnitt 4.4.2) und schließlich in Abschnitt 4.4.3 die Verbindung zwischen

diesen beiden Konzepten beschrieben, das heißt ein formalisierbarer Abbildungsmechanismus zwischen abstrakter Beschreibung und Laufzeitbeschreibung. Der letzte Abschnitt befaßt sich anschließend mit der Umsetzung der vorgestellten Konzepte in die Softwarekomponenten *Toolserver* und *Toolstarter*.

4.4.1 Abstrakte Beschreibung

Analog zur *Workflow Applikation* in $\mathcal{MOBILE}$ und dem *ToolAgent* der WFMC dient in $\mathcal{W}*FLOW$ die abstrakte Beschreibung einer Toolaktion zur Bereitstellung einer Schnittstelle zum Workflowsystem und damit der Kapselung der technischen Details wie Aufrufkonventionen, Parameterübergabe, Betriebssystembesonderheiten, notwendige Hard- und Softwarevoraussetzungen, etc. An dieser Schnittstelle wird, unabhängig von der späteren konkreten Anwendung, eine einheitliche Funktionalität zur Verfügung gestellt. Die Schnittstelle ist primär für das Starten der konkreten Anwendungen verantwortlich. Es können hier aber auch noch, in Abhängigkeit von der konkreten Anwendung und des Betriebssystems, weitere Operationen, wie z. B. das Beenden einer Anwendung, eine Neuinitialisierung, die Abfrage von aktuellen Statusinformationen, oder ein Test auf Verfügbarkeit durchgeführt werden. Wie die Funktionalität erbracht wird, ist an der Schnittstelle nicht von Bedeutung.

Bei der Modellierung der Schnittstelle zum Workflowsystem sind folgende, bereits oben angesprochene, Punkte von Bedeutung:

- Eine abstrakte Beschreibung spezifiziert einen, über ein workflow-externes Werkzeug durchgeführten Arbeitsschritt.
- Solche Arbeitsschritte benötigen für gewöhnlich Eingabedaten und erzeugen gegebenenfalls Ergebnisdaten.
- Je nach Programm und Betriebssystem gehen in den Start einer Anwendung noch weitere Parameter ein, mittels denen das Programm beeinflußt werden kann. Für die abstrakte Beschreibung sind nur solche Parameter des Tools von Interesse, die logisch zur Beschreibung des Arbeitsschritts benötigt werden. Parameter, die z. B. von der Aufrufumgebung des Betriebssystems, Hardwareplattform etc. abhängen, sind hier nicht von Interesse.

Aufbauend auf diesen Sachverhalten werden die folgenden neuen Definitionen (4.1 bis 4.5) bezüglich der Konzeption der *Toolservice* Komponenten vorgeschlagen.

> **Definition 4.1 (Signatur)** *Ein abstrakter Toolaufruf wird eindeutig durch seine Signatur beschrieben. Sie besteht aus einem Paketnamen, Methodennamen und einer Typspezifikation der Argumente des abstrakten Toolaufrufs.*

Der Begriff der *Signatur* stammt aus dem Bereich der Programmiersprachen, z. B. [Bre96]. Sie bezeichnet dort die Identifikation einer Funktion oder Methode anhand ihres Namens und der Reihenfolge und Typen der Argumente. Durch die Einbeziehung der Argumenttypen in den Begriff der Signatur ist es möglich, Methoden zu überladen, d. h. unterschiedliche Realisierungen für denselben Methodennamen einzuführen. Die Unterscheidung erfolgt in diesem Falle durch die unterscheidbaren Argumente und eine Konfliktregel. So kann es beispielsweise eine Methode edit mit einer Eingabedatei vom Typ ASCII geben und eine andere Methode edit mit einer Eingabedatei vom Typ jpeg. Hierbei handelt es sich um unterschiedliche Arbeitsschritte, denen später bei der Zuordnung von konkreten Laufzeitbeschreibungen unterschiedliche Werkzeuge wie etwa ein ASCII-Editor und ein Grafikprogramm zugeordnet werden können.

Die Trennung in Paket– und Methodenname erfolgt aus Gründen der organisatorischen Verwaltung. Hierbei dient der Paketname zur Gruppierung semantisch zusammengehörender Arbeitsschritte. Innerhalb eines Pakets wird der eigentliche Arbeitsschritt durch den Methodennamen und die Typspezifikation der Argumente spezifiziert.

Der Begriff der *abstrakten Methodenbeschreibung*, so wie er in Abschnitt 4.3 eingeführt wurde, entspricht somit genau der in Definition 4.1 eingeführten Signatur.

Im folgenden wird die Typspezifikation der Argumente innerhalb einer Signaturangabe näher beschrieben:

> **Definition 4.2 (Argument-Typen)** *Die Typ-Spezifikation der Argumente einer Signatur ist eine Folge von Typspezifikationen einzelner Argumente. Eine einzelne Typspezifikation hat eine von zwei Ausprägungen:*
>
> **FILE:** *Eine Typspezifikation der Art* FILE *bezeichnet einen Argumentparameter, der in seiner konkreten Ausprägung eine Referenz auf einen Datenstrom (Bytestream) beinhaltet. Der Inhaltstyp wird innerhalb der Signatur im MIME-Format angegeben.*
>
> **PARA:** *Argumente vom Typ Parameter stehen für einfache Werte, die direkt auf der Kommandozeile an ein Tool übergeben werden. Der Datentyp des Parameter wird in der Signatur durch die Angabe von* INTEGER, STRING *oder* DOUBLE *spezifiziert. Optional kann bei den Parametern noch eine Einschränkung des möglichen Wertebereichs angegeben werden. Dies kann z. B. mittels regulärer Ausdrücke [Fri97, Rez92] oder bei ordinalen Werten durch Intervalle erfolgen.*

Argumente vom Typ FILE stehen also für Ein-/Ausgabedaten von Tools. Sie werden als Bytestream entweder über eine temporäre Datei oder möglicherweise über Standardeingabe bzw. Standardausgabe übergeben. Parameter sind dagegen Werte, die direkt über die Aufrufzeilen an ein Programm übergeben werden.

Damit nicht nur Methoden mit einer festgelegten Anzahl von Parametern definiert werden können, ist es möglich, bei der Definition von Methoden sogenannte *Iteratoren* anzugeben. Unter einem Iterator werden sich wiederholende Eingabeparameter (–gruppen) verstanden. Somit ist es möglich Methoden zu definieren, welche eine unbestimmte[6] Anzahl von Parametern aufweisen.

> **Definition 4.3 (Iterator)** *Unter einem* **Iterator** *wird eine Gruppe von ein oder mehreren Argumenten mit ihren Typen, die wiederholt auftreten können, verstanden.*
>
> *Ein Interator wird durch ein rundes Klammerpaar gekennzeichnet, und kann optional mit der minimalen und/oder maximalen Anzahl von Wiederholungen in geschweiften Klammern genauer spezifiziert werden.*
>
> *Syntax:*[7] `(Argument:Typ [Argument:Typ [...]]) [{[min], [max]}]`

Beispiel 4.4.1 zeigt zwei Signaturen, die zur Verdeutlichung des Sachverhaltes näher besprochen werden.

[6] Es kann aber auch zusätzlich mit angegeben werden, wie oft ein Iterator minimal bzw. maximal auftreten muß bzw. darf.

[7] optionale Elemente stehen in eckigen Klammern.

```
CONVERTER::pod2x PARA:STRING:[html|ps|txt] \                    # (1)
                 FILE:application/x-perl

IMAGEPROC::COMPARE (PARA:STRING FILE:image/*) {2,}              # (2)
```

Beispiel 4.4.1: Beispiele für Signaturen

Bei der ersten Signatur handelt es sich um den Aufruf eines Konverters, der das *Perl* POD-Format[8] in ein anderes Format umwandelt. Mit dem ersten Parameter wird das gewünschte Ausgabeformat spezifiziert. Hierbei ist der Wertebereich des Arguments mittels eines in eckigen Klammern stehenden regulären Ausdrucks auf die drei Werte html, ps und txt eingeschränkt. Der zweite Fileparameter gibt die zu konvertierende *Perl* Quelldatei vom Typ application/x-perl an.

Die zweite Signatur beinhaltet einen *Iterator*. Bei dieser abstrakten Beschreibung werden paarweise Bilddateien, die auch unterschiedliche Grafikformate aufweisen können, verglichen. Da mindestens zwei Dateien notwendig sind, um sie vergleichen zu können, ist die Minimalkardinalität 2. Neben der eigentlichen Bilddatei (Argument FILE:image/*) ist wegen der unterschiedlichen in Frage kommenden Bilddateiformate noch anzugeben, um welchen Typ es sich bei der jeweiligen Bilddatei handelt (Argument PARA:STRING).

Liegt ein konkreter Aufruf einer Methode mit Parametern vor, so erfolgt die Auswahl der passenden Signatur nach folgenden Kriterien:

1. **Name der Methode und des Pakets:** Der Name der Methode und des Pakets müssen beim Aufruf mit einer Definition übereinstimmen.

2. **Typ der Parameter:** Die Argumente besitzen eine bestimmte von der Methode vorgegebene Reihenfolge, in welcher sie der Methode übergeben werden müssen. Die Reihenfolge der verschiedenen Typen ist ein Kriterium für die Auswahl der entsprechenden Signatur. Qualifizieren sich mehrere Signaturen, dann wird die Signatur ausgewählt, die eine exaktere Typbeschreibung aufweist (image/* vs. image/gif).

3. **Anzahl der Argumente:** Die Anzahl der Argumente muß mit der in der Signatur definierten Anzahl übereinstimmen.

4. **Variable Parameteranzahl (Konfliktregel):** Existiert keine Signatur mit einer fixen Anzahl von Argumenten, die auf den konkreten Methodenaufruf paßt, so werden Signaturen mit variabler Anzahl von Argumenten untersucht, ob sie sich auf den konkreten Methodenaufruf abbilden lassen. Passen mehrere Signaturen mit variablen Argumenten, so wird die Signatur gewählt, bei der die meisten fixen Argumente übereinstimmen.

5. **Zusätzliche Beschränkungen auf den Parametern:** Falls bei der Beschreibung einer Signatur eine Beschränkung des Wertebereiches eines Parameters angegeben wurde, so kann diese bei der Auswahl evaluiert werden. Hierbei sind prinzipiell zwei Interpretationen möglich:

[8] **P**lain **O**ld **D**ocument

- Die Beschränkung wird zur Auswahl der passenden Methode herangezogen. In dem Fall ist z. B. eine einfache Fallunterscheidung auf Basis von Eingabeparametern bei der Auswahl der passenden Methode möglich.
- Die Beschränkung wird nicht zur Auswahl der passenden Methode herangezogen, sondern dient lediglich der Überprüfung der Eingabeparameterwerte zur Laufzeit. In diesem Fall wird eine Methode dann ausgewählt, wenn ihre Parametertypen mit denen des Aufrufs übereinstimmen. Falls ein Parameterwert nicht mit einer Beschränkung übereinstimmt, wird eine Fehlermeldung ausgegeben.

Die erste Alternative stellt die flexiblere Lösung dar, während die zweite Möglichkeit einfacher zu realisieren ist und gleichzeitig bessere Möglichkeiten zur Fehlerdetektion (falsche Eingabewerte) bietet.

Die Fähigkeit eines Programms, erst zur Laufzeit zu entscheiden, welche Realisierung einer Methode eingesetzt werden soll, wird in der Literatur als *dynamisches Binden* oder *Bindung zur Laufzeit* [Mey88, Bre96] bezeichnet.

4.4.2 Laufzeitbeschreibung

Das Gegenstück der abstrakten Beschreibung, wie sie im vorherigen Abschnitt mittels Paketen, Methoden und Typspezifikationen von Argumenten definiert wurde, stellt die Laufzeitbeschreibung dar.

Die Laufzeitbeschreibung ist die konkrete Beschreibung *was*, *wo* unter *welchen Randbedingungen* ausgeführt werden soll. Im folgenden Abschnitt wird auf die einzelnen Komponenten, aus denen eine vollständige Laufzeitbeschreibung besteht, näher eingegangen.

Wichtigstes Element der Laufzeitbeschreibung ist das Ausführungskommando. Dabei wird zwischen zwei Ausprägungen unterschieden:

Programm: Es handelt es sich um ein konkretes, auf dem betreffenden Rechner ausführbares Programm. Dem Programm können in der Aufrufzeile eine beliebige Anzahl von FILE- oder PARA-Argumenten mit übergeben werden.

Skript: Bei dieser etwas komplexeren Variante wird nicht nur der Aufrufstring eines Programms übergeben, sondern es wird der Inhalt eines kompletten Skripts übertragen, das dann auf dem Zielrechner ausgeführt werden soll.

Insbesondere die zweite Variante stellt eine sehr flexible Möglichkeit bei der Formulierung von komplexen Arbeitsschritten dar. So kann es gerade im Bereich Softwareinstallation sehr nützlich sein, komplexe Skripte zu erstellen, die komplette Installationen von Anwendungen oder gar komplette Rechnerinstallationen vornehmen. Dies kann gegebenenfalls geschehen, indem ein Programm im Quelltext übertragen wird, dort übersetzt und anschließend installiert wird. Hiermit läßt sich eine zentrale Rechnerverwaltung bzw. Softwareverteilung und Administration realisieren.

Neben der Angabe des Programms oder Skripts ist die Information über die Art der Applikation von Interesse. Die Angabe wird von der *Toolstarter*-Komponente benötigt, um das Programm richtig ausführen zu können. So spielt es etwa eine Rolle, ob das Programm eine eigene Fensterumgebung bereitstellt, oder ob der Applikation vom *Toolstarter* ein Terminalfenster zum Ablauf zur Verfügung gestellt werden muß.

Um ein Programm richtig einsetzen zu können, ist es oft zusätzlich notwendig, eine Reihe von Umgebungsvariablen zu definieren, welche vom Programm gelesen werden. Diese definieren z. B. wo bestimmte Konfigurationsdateien zu finden bzw. temporäre Ergebnisse abzulegen sind.

Neben der lokalen Ausführung von Programmen ist es heute üblich, Programme auf entfernten Rechnern ablaufen zu lassen. In der *UNIX*–Welt ist es durch Programme wie rlogin, rsh und telnet möglich, Applikationen auf fremden Rechnern zu starten und zu steuern. Mittels der *UNIX* Fensterumgebung *X* [SG92] ist es möglich, Programme auf einem Rechner laufen zu lassen und die (graphische) Ein– /Ausgabe auf einen anderen Rechner umzuleiten.

Die kompletten Informationen bestehend aus Kommando oder Skript, Environment, Zielrechner und Typ der Applikation, welche notwendig sind, um eine Applikation erfolgreich in einer speziellen Umgebung zu starten, werden als *Laufzeitbeschreibung* eines abstrakten Toolaufrufs bezeichnet.

Definition 4.4 (Laufzeitbeschreibung) *Die Laufzeitbeschreibung eines Toolaufrufs umfaßt:*

1. *Das Aufrufkommando mit Argumenten. Hierbei handelt es sich um den Aufrufstring, bei dem die Argumente, so wie sie in der abstrakten Beschreibung definiert wurden, durch* `$<x>` *Platzhalter referenziert werden.*
2. *Handelt es sich um ein Skript, so gehört neben dem Aufrufkommando der komplette Skriptcode zur Laufzeitbeschreibung.*
3. *Der Typ der Applikation:* INTERAKTIV, WINDOW, ...
4. *Der Zielrechner, auf dem das Programm oder Skript ablaufen soll.*
5. *Notwendige Umgebungsvariablen.*
6. *Weitere Informationen zum auszuführenden Tool bzw. eine Referenzierung auf weitere Informationen.*

Wie in den Anforderungen festgelegt, sollen die $\mathcal{W}$*FLOW–Toolservices*, neben der *Blackbox*–Integration, auch eine *Greybox*– und *Whitebox*–Integration von Programmen ermöglichen. Da es sich bei $\mathcal{W}$*FLOW* um einen Baukasten handelt, der mit einer netzwerktransparenten Laufzeit–API ausgestattet ist, ist der Fall der *Whitebox*–Integration bereits abgedeckt (siehe dazu auch Seite 86 – Realisierungsschnittstelle), da hierzu lediglich ein Programm, das die API nutzt, realisiert werden muß. Als einziger Punkt ist hierbei die eigentliche API um eine Klasse zu erweitern, die von einem Programm aus den Zugriff auf die $\mathcal{W}$*FLOW*–Engine erlaubt, d. h. die Verbindung zwischen Programm und $\mathcal{W}$*FLOW*–Engine herstellt.
Die *Greybox*–Integration kann analog mittels eines Wrapperprogramms, das sowohl die $\mathcal{W}$*FLOW*–API, als auch die API der anzubindenden Anwendung nutzt, abgehandelt werden. Das Codefragment in Beispiel 4.4.2 zeigt ein Programm oder Skript, das auf Basis der $\mathcal{W}$*FLOW*–API realisiert wurde und mittels der Klasse WildFlow::Task Zugriff auf die $\mathcal{W}$*FLOW*–Engine erhält.

Anweisung (1) stellt die eigentliche Verbindung zur $\mathcal{W}$*FLOW*–Engine her. Dies ist so realisiert, daß beim Start einer Anwendung mittels des Toolstarters eine Reihe von Umgebungsvariablen gesetzt werden können. Der Aufruf der Methode `WildFlow::Task->new(@ARGV)` untersucht die Umgebung, innerhalb der das Programm gestartet wurde, nach bestimmten von der $\mathcal{W}$*FLOW*–Engine gesetzten Umgebungsvariablen und nutzt die darin vorhandenen Informationen, um sich mit der $\mathcal{W}$*FLOW*–Engine zu verbinden. Die zurückgelieferte Instanzenvariable $task erlaubt

```
#!/usr/bin/perl -w

use WildFlow::API;

my $task = WildFlow::Task->new(@ARGV);                                  # (1)

begin 'read, sub {                                                      # (2)

  my $container = $task->get_container('Modelle');                      # (3)
  my @objects = $container->object_list();                              # (4)

  foreach my $object (@objects) {                                       # (5)
      print $object->NAME()."\n";
  }
};
die "$@\n" if $@;
```

Beispiel 4.4.2: *Whitebox*–Integration eines Programms oder Skripts

anschließend den Zugriff auf die Aktivitäteninstanz. In diesem Beispielprogramm wird auf den Container Modelle zugegriffen (3) und die Namen aller darin enthaltenen Containerobjekte (4) auf die Standardausgabe geschrieben (5).
Benutzt dagegen das Skript z. B. noch eine der zahlreichen Module aus den CPAN Archiven, etwa um Zugriff auf eine *ORACLE* Datenbank zu erlangen, so liegt eine *Greybox*–Integration vor, da in diesem Fall von $\mathcal{W}*FLOW$, mittels einer externen API[9], der Zugriff auf ein externes System realisiert wird.

4.4.3 Zuordnung von Signaturen zu Laufzeitbeschreibungen

Nachdem in den beiden vorausgehenden Abschnitten sowohl die abstrakte Beschreibung eines Arbeitsschrittes konzipiert als auch das Gegenstück, die konkrete Laufzeitbeschreibung dazu, spezifiziert wurde, wird nun die letzte der Anforderungen aus Abschnitt 4.3 besprochen: der formalisierbare Abbildungsmechanismus zwischen beiden.

Jeder Signatur kann eine Vielzahl von konkreten Laufzeitbeschreibungen zugeordnet werden. Die Zuordnung einer Signatur zu einer Laufzeitbeschreibung erfolgt in Abhängigkeit des Rechners, auf dem das Programm ablaufen soll. Diese feingranulare Unterscheidung wird z. B. vom System $\mathcal{MOBILE}$ gemacht, dessen Toolaufrufschnittstelle in Abschnitt 4.2 vorgestellt wurde. In der Realität führt das jedoch zu einem unnötigen Aufwand bei der Spezifikation der einzelnen Kommandos, da sich in vielen Fällen eine große Anzahl von konkreten Laufzeitbeschreibungen gleichen. Die Gültigkeitsgrenze einer Laufzeitbeschreibung endet oft nicht am Gehäuse des betreffenden Rechners, sondern umfaßt einen größeren Wirkungsbereich. Mögliche Schranken des Gültigkeitsbereichs werden beispielsweise von den folgenden Kriterien bestimmt:

Die Auswahl einer bestimmten Laufzeitbeschreibung kann in Abhängigkeit von

- Rechnerarchitektur
- Betriebssystem

[9] z. B. das Modul DBD::ODBC

- Konkreter Rechner

erfolgen.

Durch die prädikatenlogische Kombination obiger Kriterien läßt sich eine weitaus flexiblere Zuordnung von Laufzeitbeschreibungen zu abstrakten Beschreibungen erreichen. Indem nun diese Selektionskriterien um die folgenden beiden Kriterien erweitert werden, lassen sich auch benutzer– oder arbeitsgruppenbezogene Einträge vornehmen.

Die Auswahl einer bestimmten Laufzeitbeschreibung kann weiterhin in Abhängigkeit von

- der Gruppenzugehörigkeit eines Benutzers
- dem Benutzer selbst

getroffen werden.

Obige Kriterien können um weitere ergänzt werden. So kann es beispielsweise sinnvoll sein, zusätzlich noch ein Kriterium einzuführen, das die Rechner aus verschiedenen Abteilungen unterscheidet, da sie von verschiedenen Administratoren gewartet werden und deshalb unterschiedlich konfiguriert sein können. Weiterhin ist auch eine Hierarchisierung, wie sie in Abbildung 4.2 am Beispiel der Betriebssysteme gezeigt wird, oder ein *Pattern Matching* Mechanismus sinnvoll. Bedingung ist lediglich, daß die Kriterien zur Laufzeit ermittelt werden können.

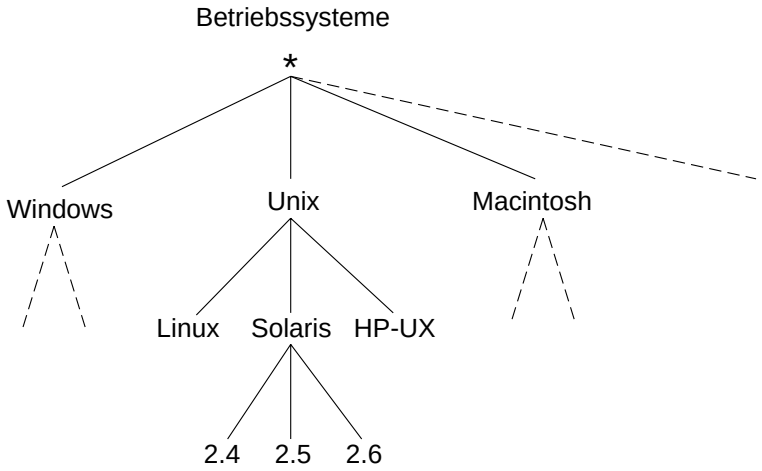


Abbildung 4.2: *Hierarchische Strukturierung eines Kriteriums*

Diese Randbedingungen werden im folgenden als *Laufzeitinformationen* bezeichnet. Bei jeder Anforderung von Seiten eines Klienten, ein Programm zu starten, werden die Informationen zusammen mit der Angabe von Paket, Methode und Parameter an den *Toolserver* übertragen.

> **Definition 4.5 (Laufzeitinformation (Qualifier))** *Unter Laufzeitinformationen werden solche Informationen verstanden, die zur Laufzeit extrahiert, und als Qualifier für eine bestimmte Laufzeitumgebung genutzt werden können. Die Abbildung von einer abstrakten Methodenbeschreibung hin zu einer Laufzeitbeschreibung ist somit eine Funktion in Abhängigkeit der Laufzeitinformationen.*

Die Abbildungsvorschrift zwischen abstrakter Methodendefinition und konkreter Laufzeitbeschreibung ist in Abbildung 4.3 graphisch dargestellt.

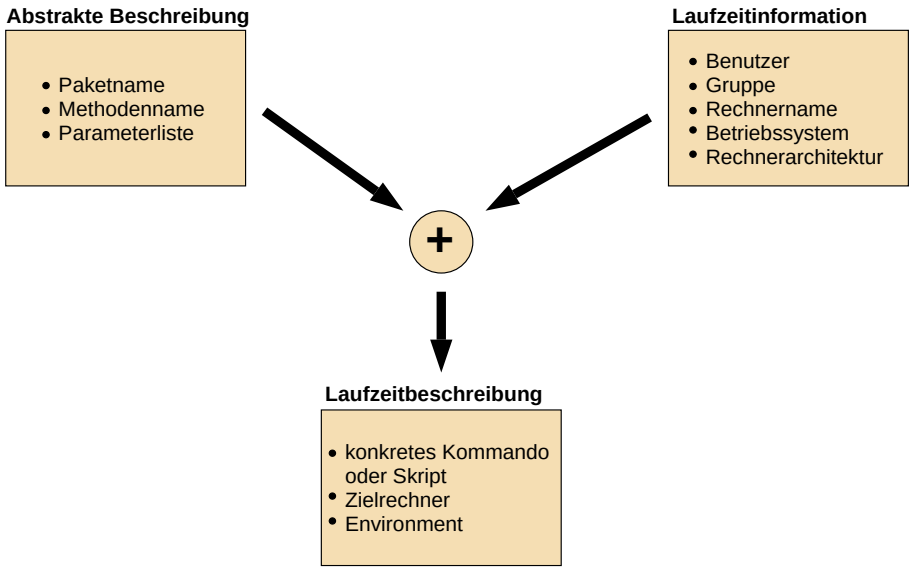


Abbildung 4.3: *Abbildungsvorschrift des* Toolservers

Die Einträge des *Toolservers* liegen in der Form vor, daß für eine bestimmte abstrakte Methode eines Pakets in Abhängigkeit von den Laufzeitinformationen spezifische Laufzeitbeschreibungen gespeichert sind.

Damit nun nicht für jede erdenkliche Laufzeitbeschreibung ein konkreter Eintrag in der Datenbank vorliegen muß, ist es möglich, durch den Einsatz von *Wildcards* (*) dem System ein Standardverhalten zu geben. So kann z. B. durch den ersten Eintrag in Abbildung 4.4 festgelegt werden, daß die Bearbeitung von ASCII Dokumenten auf SunOS Rechnern mittels des Editors textedit erfolgt (3. Eintrag von oben).

Im allgemeinen werden von einem Systemadministrator in einem ersten Schritt für jede neue Signatur eine Registrierung für die unterschiedlichen Betriebssysteme bzw. Rechnerarchitekturen festgelegt. Die Einträge können dann durch Spezialisierung über Rechnername, Gruppe oder Benutzer von den Anwendern überschrieben werden.

In Abbildung 4.4 ist beispielsweise als Standardeintrag für die Signatur TOOLS:EDIT(text/plain) und das Betriebssystem Windows NT der Editor notepad eingetragen. Eine Konkretisierung des Eintrags für den Benutzer schmidt sieht vor, stattdessen eine NT-Version des Editors emacs auf der lokalen Maschine zu starten. Ein weiterer Eintrag für diese Signatur sieht für den Benutzer sieber vor, auf allen Rechnern mit dem Betriebssystem Linux einen xemacs Prozeß auf dem Rechner miserv2 zu starten und die Ausgabe mittels einer vordefinierten Umgebungsvariablen *WF_DISP* auf sein lokales *X*–Display umzuleiten.

4.5 Realisierung

Nachdem in den vorherigen Abschnitten das Grundkonzept einer abstrakten Beschreibungsmethode auf Basis von Paketen und Methoden erläutert wurde, wird im vorliegenden Abschnitt die Architektur und Funktionalität der zu realisierenden Komponenten vorgestellt. Es handelt sich, wie bereits in der Einleitung des Abschnittes über die *Toolservices* erwähnt, um die Komponenten *Toolserver* und *Toolstarter*, welche sich die Aufgabe teilen, Applikationen in verteilten heterogenen Umgebungen zu starten. Die Aufgaben sind derart verteilt, daß die *Toolserver*

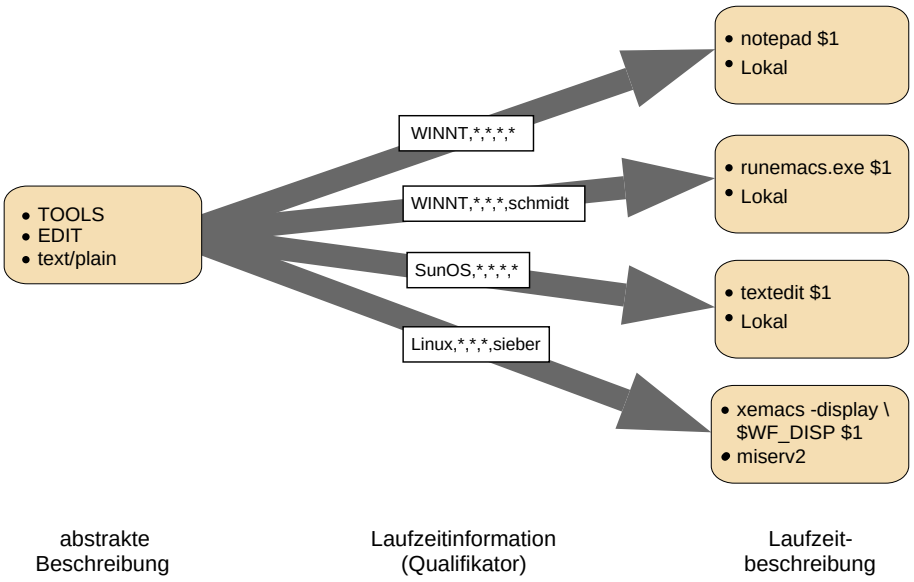


Abbildung 4.4: *Beispieleinträge für die Signatur* TOOLS:EDIT(text/plain)

Komponente die Verwaltung der abstrakten Methodenbeschreibungen und deren Abbildung auf konkrete Laufzeitbeschreibungen vornimmt, während die *Toolstarter* Komponente für die Bereitstellung der Ablaufumgebung, den Start, sowie die Überwachung des Programms verantwortlich ist.

4.5.1 Architektur der Toolservices

Die Architektur der Toolservices basiert auf dem Client–Server Prinzip. Die Persistenz wird von der kommerziellen Datenbank *ObjectStore* der Firma Object Design realisiert. Die Funktionalität des *Toolservers* ist in einem Client Prozeß der *ObjectStore* Datenbank realisiert. Die vorliegende Realisierung erlaubt es, mehrere *Toolserver*-Prozesse verteilt laufen zu lassen. Diese Architektur ist in der Literatur auch unter dem Begriff *Multi–Tier–Architektur*[10] [Car96] bekannt, da die zu erbringenden Dienste hierbei auf mehrere Rechner verteilt werden.

Abbildung 4.5 zeigt den prinzipiellen Aufbau der Toolservices. In der Abbildung befindet sich oben der *ObjectStore* Server, der den eigentlichen *Toolserver*-Prozessen (Mitte der Abbildung) die persistente Speicherung der Daten erlaubt. Das Netzprotokoll zwischen dem *ObjectStore* Server und den *Toolservern* basiert auf einem seitenorientierten Memory Mapping Mechanismus.

Die Kommunikation zwischen den *Toolserver*-Prozessen und den *Toolstartern* (untere Reihe der Abbildung) basiert auf der $\mathcal{W}$*FLOW*–API. Da es sich bei den *Toolservern* um Client–Prozesse eines *ObjectStore* Servers [Obj96a, Obj96b] handelt, d. h. die Daten nicht zentral vom *Toolserver* abgelegt und verwaltet werden, sondern in einer externen *ObjectStore* Datenbank, die sich auf einem beliebigen Rechner befinden kann, besitzt er ein hohes Maß an Skalier– und Verfügbarkeit.

In den folgenden Abschnitten wird besprochen, wie die Schnittstellen der Komponenten aussehen und wie die Kommunikation zwischen *Toolserver* und *Toolstarter* funktioniert.

[10] siehe Glossar Seite 181

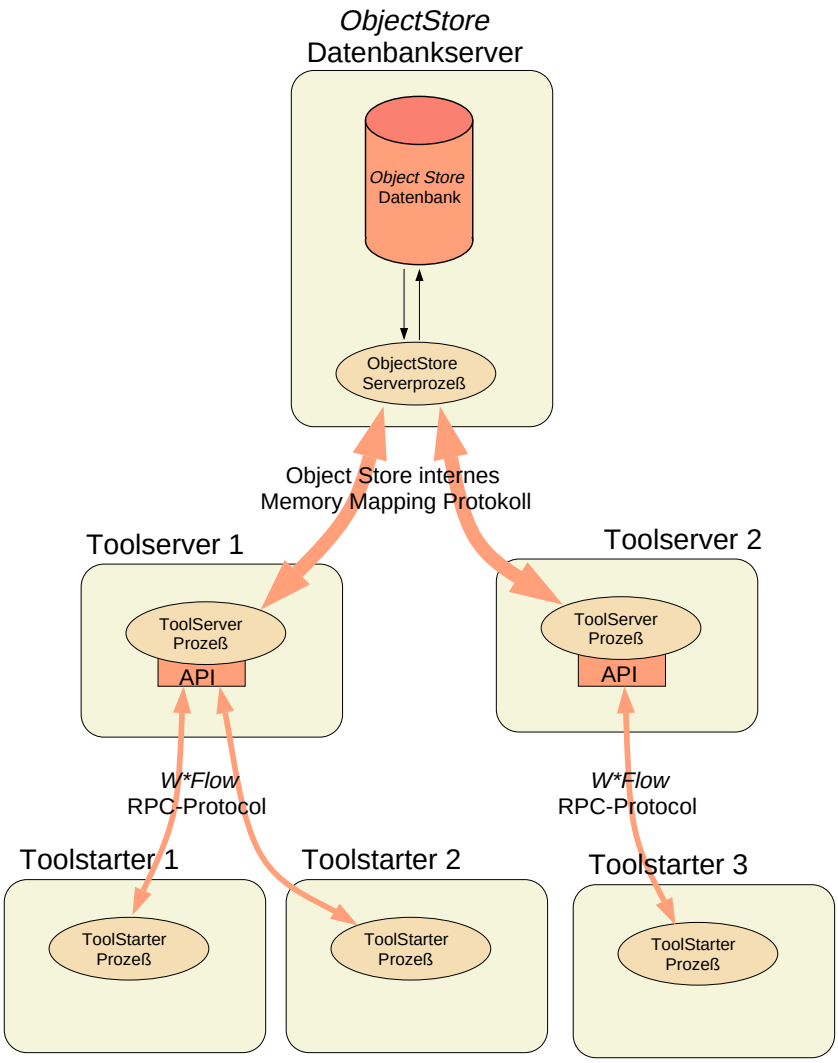


Abbildung 4.5: *W∗FLOW: Architektur der Toolservices*

4.5.2 Toolserver

Aufgabe des *Toolservers* ist die Verwaltung der Abbildungsvorschriften zwischen abstrakter Methodenbeschreibung und konkreter Laufzeitinformation.
Die Abbildung einer abstrakten Methodenbeschreibung hin zu einer konkreten Laufzeitbeschreibung innerhalb des *Toolservers* ist in Abbildung 4.6 repräsentiert. Deutlich sind die drei Bereiche *abstrakte Beschreibung* (links), die als *Qualifikator* dienende *Laufzeitinformation* (mitte) und die eigentliche *Laufzeitbeschreibung* (rechts) zu erkennen.

Die Funktionalität der Schnittstelle zum *Toolserver* kann in zwei Teile untergliedert werden:

1. Administrative Schnittstelle
2. Laufzeit Schnittstelle

Unter der *administrativen* Schnittstelle wird der Teil des *Toolservers* verstanden, der sich mit der Administration der Toolbeschreibungen befaßt. Im Rahmen dieser Teilschnittstelle werden

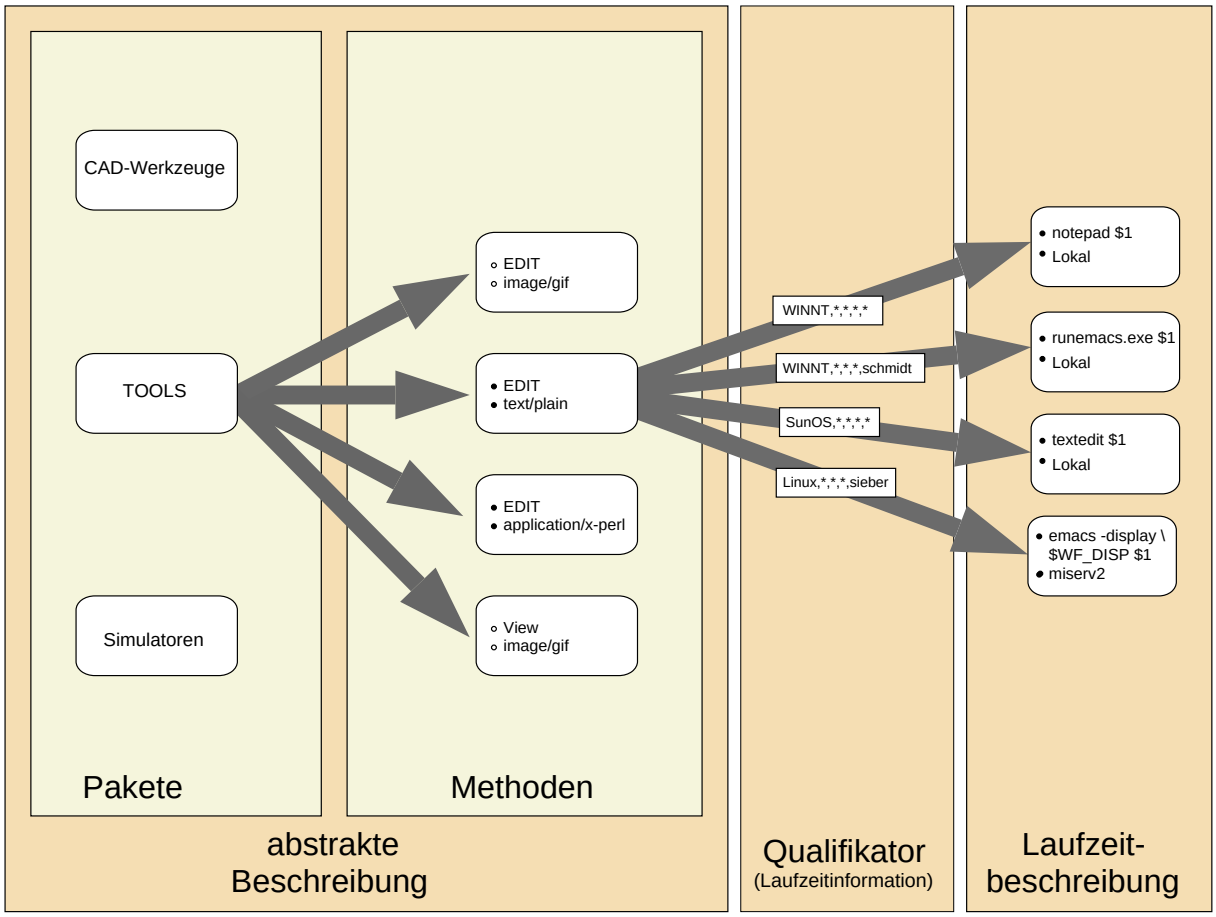


Abbildung 4.6: *W*FLOW: Toolserver Datenstruktur*

Methoden zum Zugriff, Anlegen und Löschen von Paketen, Signaturen und Laufzeitbeschreibungen bereitgestellt. Weiterhin können noch eine Reihe von Zusatzinformationen zu den einzelnen Paketen und Signaturen verwaltet werden.

Im folgenden sollen einzelne Bereiche der administrativen Schnittstelle des Toolservers, so wie sie im Rahmen des PRAXIS Projektes [DGS98a, DGS98b, DGS98c] am IAI realisiert wurden, kurz vorgestellt werden. Die Benutzerschnittstelle baut auf die *W*FLOW*–API auf und ist in Form einer WWW–basierten Oberfläche realisiert.

Als oberstes Strukturierungsmittel gibt es, wie in Definition 4.1 eingeführt, das Paket. Abbildung 4.7 zeigt einen Überblick über alle im *Toolserver* definierten Pakete.

Von dieser Seite kann durch Anklicken des entsprechenden Paketnamens (1) eine weitere Seite geöffnet werden, die einen Überblick über ein einzelnes Paket gibt, so wie in Abbildung 4.8 am Beispiel des Pakets Tools, dargestellt. Das Paket enthält zwei Methoden mit dem Namen EDIT, die sich jedoch in den Parametern unterscheiden. So ist es möglich, ohne genaue Kenntnis des Dateityps, die Methode EDIT zur Bearbeitung einer Reihe von unterschiedlichen Dateien zu spezifizieren, da die Auswahl der entsprechenden Methode erst zur Laufzeit und in Abhängigkeit des Dateityps erfolgt. Das genaue Aussehen einer Methodensignatur kann durch Anklicken des entsprechenden Eintrags in der Spalte Parameter ermittelt werden. Ein Beispiel für eine etwas komlexere Signatur mit Iterator ist in Abbildung 4.13 dargestellt.

Abbildung 4.7: *Liste der Pakete im* PRAXIS *Toolserver*

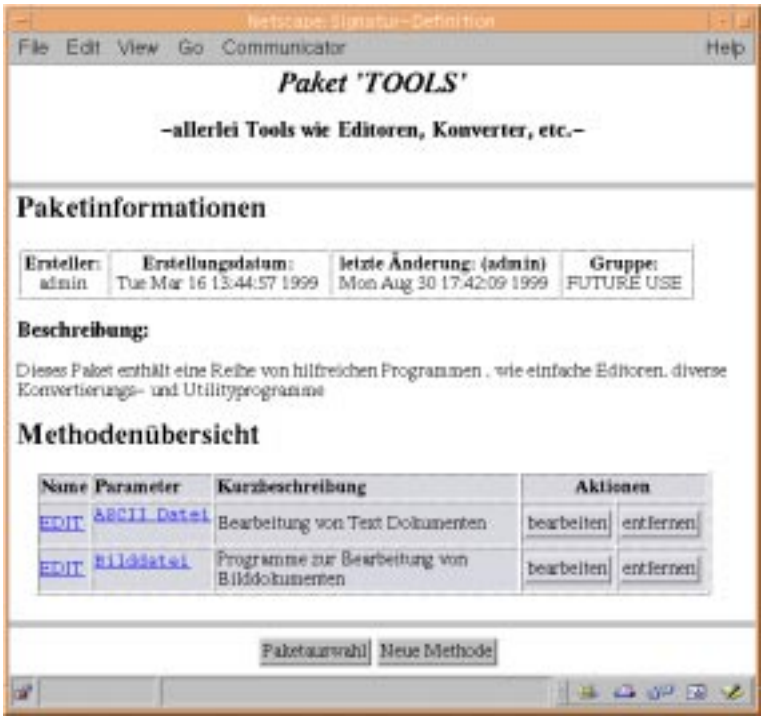


Abbildung 4.8: *Liste der Methoden innerhalb eines* PRAXIS *Toolserver Pakets*

Abbildung 4.9 zeigt ein Formular, das die Schnittstelle zwischen abstrakter Methodenbeschreibung und konkreten Kommandos darstellt. Die Einträge entsprechen denen aus Abbildung 4.6. Dargestellt wird eine Übersicht der verfügbaren Kommandos für die Methode EDIT aus dem TOOLS Paket mit einer ASCII Datei als Parameter. In den ersten fünf Spalten der Tabelle werden die spezifizierten Qualifier angezeigt, wobei das Sternsymbol (*) signalisiert, daß für einen Qualifier keine Angabe gemacht wurde. Die Spalte mit der Bezeichnung **Typ** kennzeichnet, ob es sich bei dem Eintrag um ein **K**ommando oder ein **S**kript handelt.

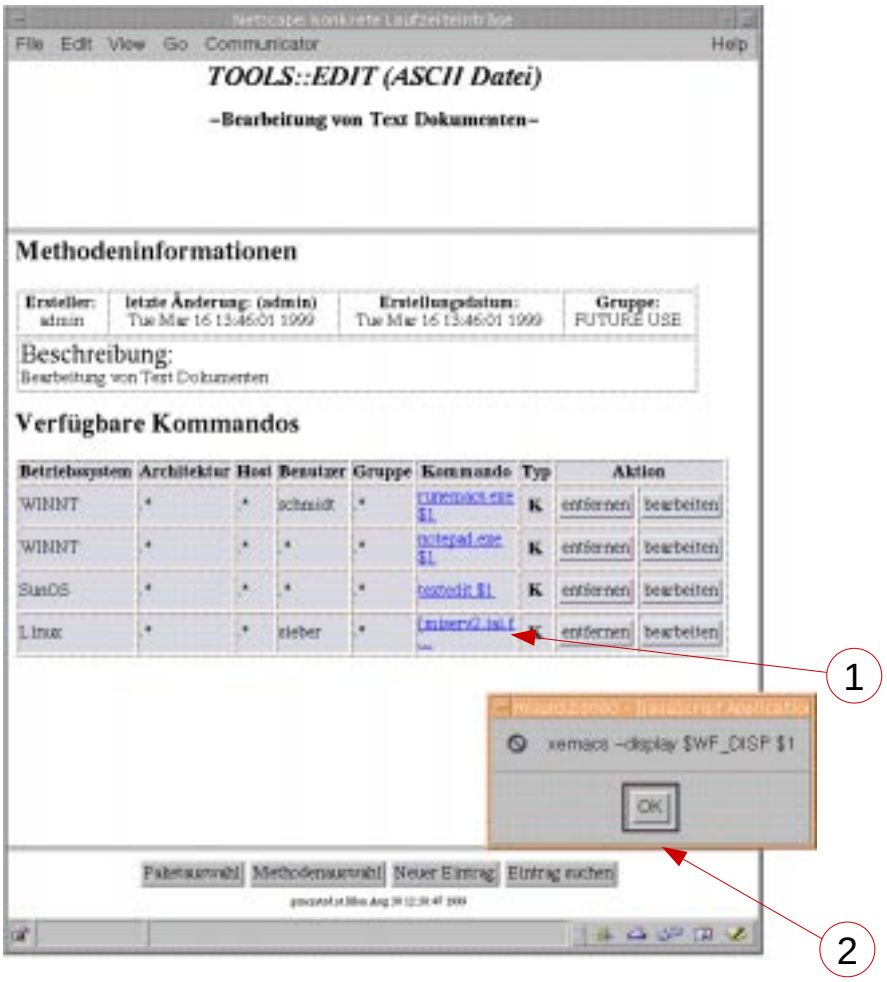


Abbildung 4.9: *Einträge aus Abbildung 4.4 im* PRAXIS *Toolserver*

Durch Anklicken der Einträge in der Spalte Kommando (1) wird ein weiteres Fenster (2) geöffnet, das den vollständigen Aufrufstring bzw. das komplette Skript des jeweiligen Eintrags enthält.

Im unteren Bereich des Formulars existieren Buttons zur Paket- und Methodenauswahl, zum Suchen von konkreten Methoden anhand einer zu spezifizierenden Laufzeitbeschreibung und zur Definition von neuen Laufzeitbeschreibungen.

Abbildung 4.10 zeigt die Definition eines Laufzeiteintrags. Im oberen Bereich (1) werden die Laufzeitinformationen, bestehend aus Betriebssystem, Architektur, Host, Benutzergruppe und Benutzer, spezifiziert, für die der folgende Eintrag Gültigkeit besitzt. In der unteren Hälfte wird das konkrete Kommando oder Skript (2), der Zielrechner (3) sowie optional eine zusätzliche Beschreibung (4) oder ein Verweis darauf gespeichert. In dem konkreten Beispiel wird der Edi-

tor xemacs auf dem Rechner miserv2.iai.fzk.de gestartet und die Ausgabe auf das mittels der Umgebungsvariable $WF_DISP (s. u.) spezifizierte *X*-Display umgeleitet. Neben dem eigentlichen Aufrufstring, bei dem das FILE Argument durch den $1 Platzhalter repräsentiert und zur Laufzeit durch den aktuellen Dateinamen ersetzt wird, erfolgt zusätzlich die Angabe einer URL, die weitere Dokumentation über das verwendete Programm referenziert. Der *Toolstarter* erhält von der *W*FLOW*-Engine noch eine Reihe von Umgebungsvariablen mit übergeben, auf die bei der Definition des Aufrufstrings zurückgegriffen werden kann. So wird in dem Beispiel die *W*FLOW*-Umgebungsvariable $WF_DISP benutzt, um die Ausgabe auf das benutzerlokale Display umzuleiten. Die Liste der momentan von *W*FLOW* unterstützten Umgebungsvariablen ist in Tabelle 4.1 zu sehen.

Tabelle 4.1: *Von W*FLOW definierte Umgebungsvariablen*

Name	Bedeutung
WF_LOCALHOST	Host, auf dem das Kommando initiiert wurde.
WF_HOST	Host, auf dem das Kommando abläuft
WF_OS	Betriebssystem, auf dem das Kommando abläuft
WF_USER	Benutzer, der das Kommando initiiert hat
WF_GROUP	Gruppe, der der Benutzer angehört
WF_DISP	*X*-Display, an dem der Benutzer sitzt (nur bei *X11*-Umgebungen definiert)

Neben der normalen Angabe von fixen Parametern unterstützt der *PRAXIS* Toolserver auch Iteratoren, wie sie in Abschnitt 4.4.1 vorgestellt wurden. Die Definition einer Methode mit variabler Anzahl von Parametern soll im folgenden anhand der Abbildungen 4.11 und 4.12, sowie der Methode COMPARE aus Beispiel 4.4.1 gezeigt werden.

Die erste Abbildung zeigt das Formular, innerhalb dem das Verhalten des Iterator, d. h. die Anzahl der darin auftretenden Felder (1) und die minimale und maximale Anzahl von möglichen Wiederholungen (2) festgelegt werden. Abbildung 4.12 legt anschließend die innere Struktur des Iterators fest, d. h. den Typ der einzelnen Felder (1), den Inhaltstyp (2) und optional eine erklärende Beschreibung (3).

Im konkreten Fall ist der erste Iterator-Parameter vom Typ String und wird zur Spezifikation des Typs des zweiten Parameters benötigt. Der zweite Parameter ist vom Typ File und kann von einem beliebigen Untertyp des MIME-Typs Image sein. Eine Einschränkung für Parameterwerte, wie sie in Abschnitt 4.2 vorgestellt und in Beispiel 4.4.1 zum Einsatz kommt, wird vom *PRAXIS* Toolserver momentan nicht unterstützt. Im oberen Bereich (4) wird dem Anwender der aktuelle Status bei der Definition des Iterators gezeigt (`[...]` Felder). Dies ist besonders bei Modifikationen an bestehenden Signaturen sowie Signaturen mit mehreren bzw. geschachtelten Iteratoren, wie sie vom *PRAXIS* Toolserver unterstützt werden, hilfreich.

Nach Abschluß der Definition einer neuen Methode erscheint diese in der Auflistung aller innerhalb eines Pakets definierten Methoden (Abbildung 4.13). Durch Anklicken des Eintrags (1) in der Spalte Parameter öffnet sich ein weiteres Fenster und zeigt eine semiformale Repräsentation der Signatur der Methode.

Das Gegenstück zur administrativen Schnittstelle bildet die *Laufzeit* Schnittstelle des *Toolservers*. Sie erlaubt die Abfrage von konkreten Laufzeitbeschreibungen für die im Rahmen der

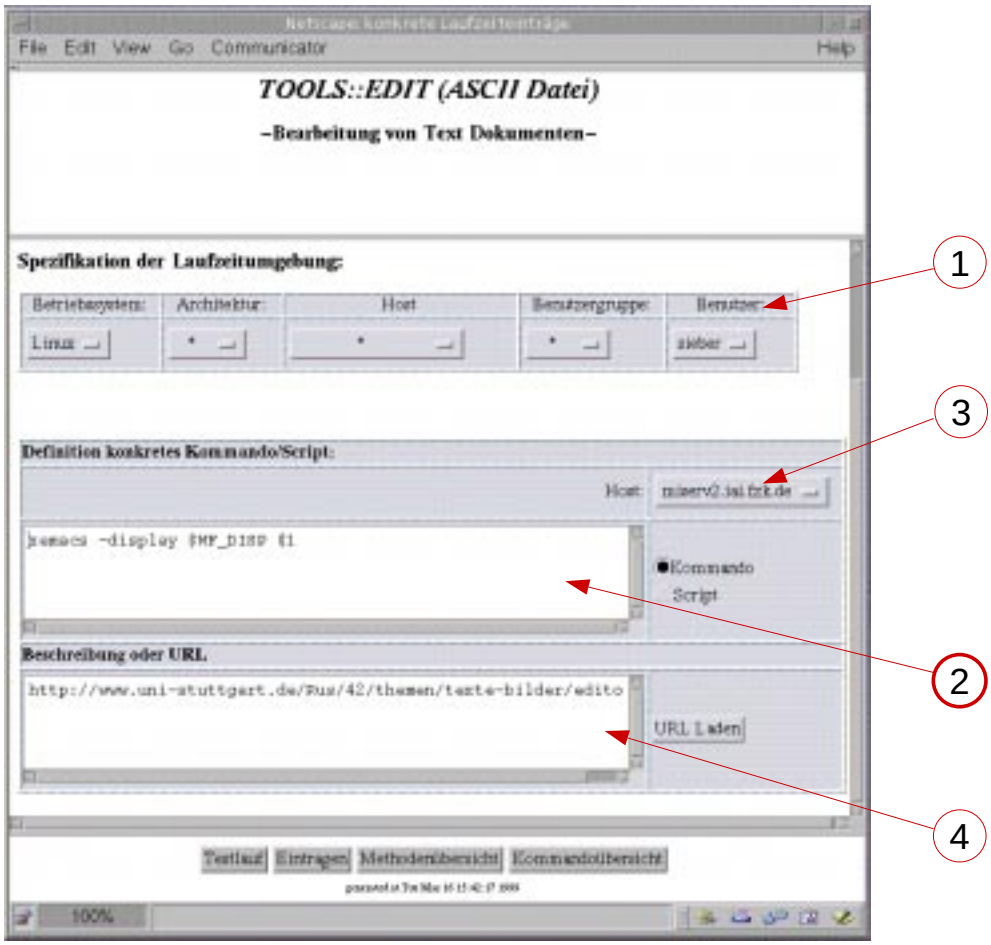


Abbildung 4.10: *Eintrag einer Laufzeitbeschreibung in das WWW basierte* PRAXIS *Toolserver Frontend*

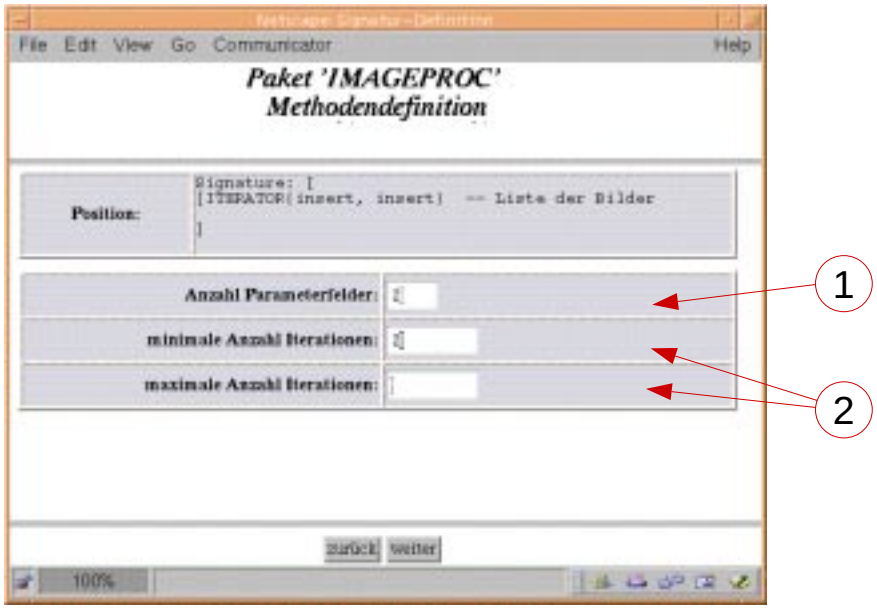


Abbildung 4.11: *Erstellen eines neuen Methodeneintrags im* PRAXIS *Toolserver (1)*

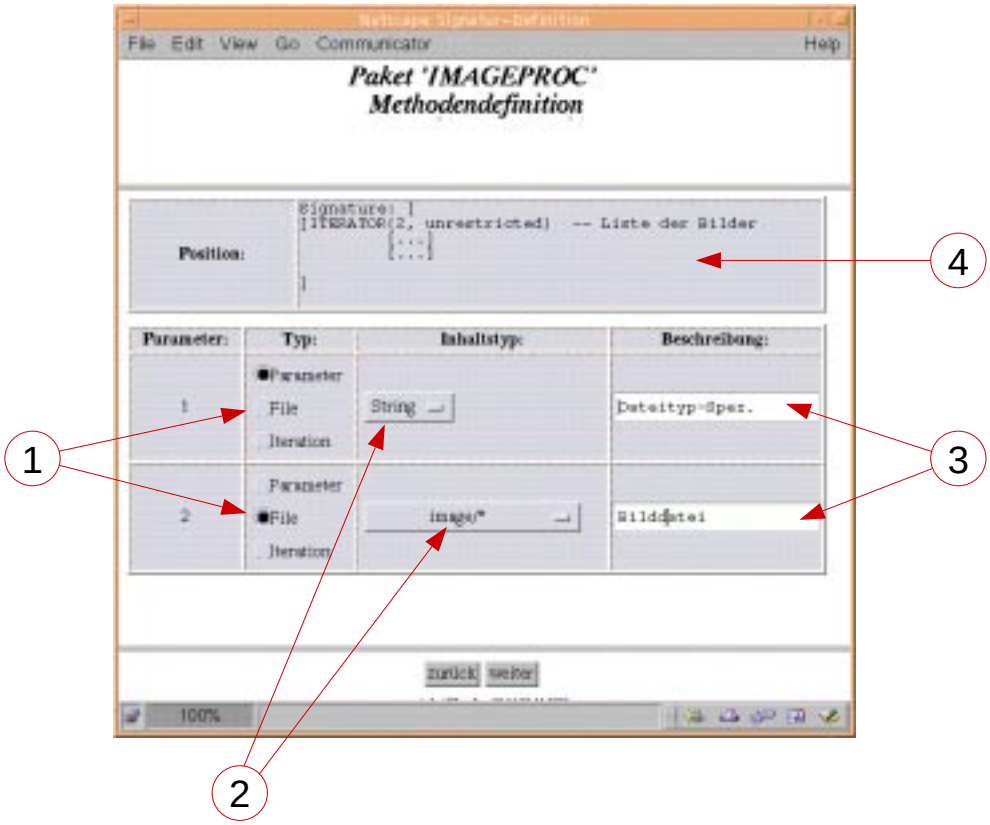


Abbildung 4.12: *Erstellen eines neuen Methodeneintrags im* PRAXIS *Toolserver (2)*

administrativen Schnittstelle definierten Signaturen der Pakete.

Eine Laufzeitbeschreibung (siehe Abschnitt 4.4.2) besteht im einzelnen aus folgenden Angaben:

- Das konkrete Kommando oder Skript.
- Art der Applikation, um die es sich handelt (fensterbasiert, interaktiv, ...)[11].
- Der Rechner, auf dem das Programm oder Skript ausgeführt werden soll.
- Das Environment, das benutzt werden soll[12].

Die *Laufzeit* Schnittstelle wird vom *Toolstarter* genutzt, um die abstrakte Methodenbeschreibung aufzulösen. Im Rahmen des *PRAXIS* Projekts wurde die Laufzeitschnittstelle als CGI–Programm realisiert. Dies bedeutet, daß eine Anfrage an den *Toolserver*, die FORM eines URL besitzt.

Eine Anfrage an die Laufzeitschnittstelle des Toolservers hat das in Beispiel 4.5.1 dargestellte Format:

Dabei müssen die Parameter PACKET, METHOD und SIGN auf jeden Fall definiert sein, während von den drei Parametern OS, ARCH und HOST mindestens einer definiert sein muß. Die Parameter USER und GROUP sind optional.

[11] Im *PRAXIS* Toolserver derzeit nicht implementiert.
[12] derzeit nicht implementiert.

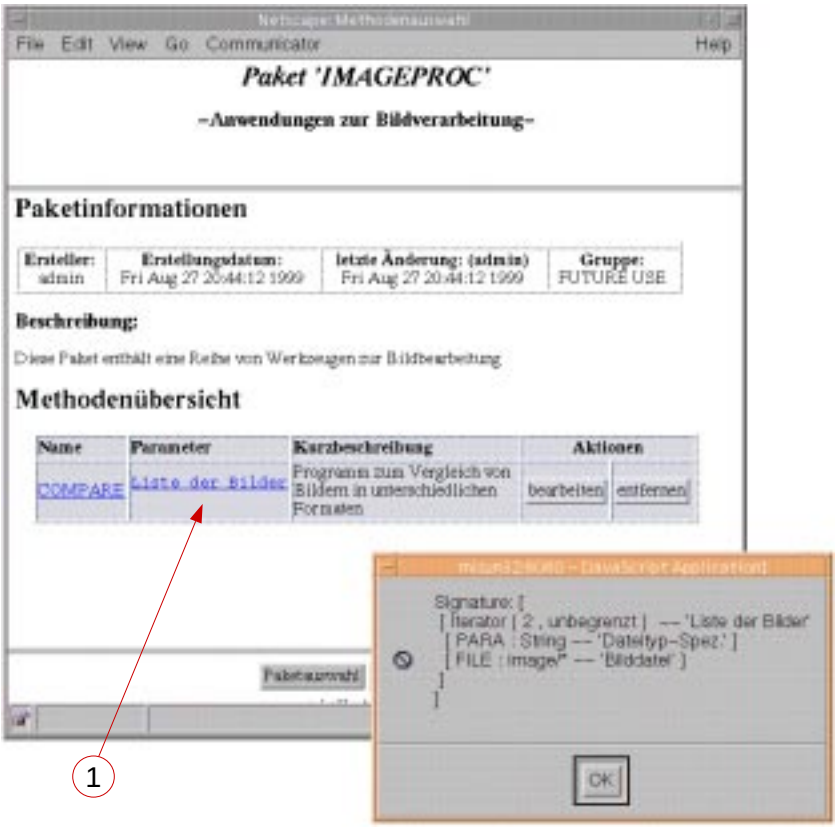


Abbildung 4.13: *Auflistung der in einem Paket definierten Methoden*

```
http://www.iai.fzk.de/praxis/toolserver.cgi?PACKET=<packet>&\
                                           METHOD=<method>&\
                                             SIGN=<sig>&\
                                             USER=<user_name>&\
                                            GROUP=<group>&\
                                               OS=<os>&\
                                             ARCH=<arch>&\
                                             HOST=<host>
```

Beispiel 4.5.1: Format der URL mit Parametern zur Anfrage des *PRAXIS–Toolservers*

Der *Toolserver* erhält seine Anfragen vom *Toolstarter*, auf dessen Realisierungsdetails im folgenden näher eingegangen wird.

4.5.3 Toolstarter

Beim *Toolstarter* handelt es sich um eine Softwarekomponente, welche für den Start der eigentlichen Applikationen verantwortlich ist. Das impliziert auch die Bereitstellung der notwendigen Daten und den Rücktransport von erzielten Ergebnissen.

Ausgangspunkt für den Aufruf einer Applikation durch einen Klienten ist eine Steuerungsdatei, die ein fest definiertes Format hat. In ihr werden unter anderem der Name des Pakets, die Methode und die benötigten Daten festgelegt. Im folgenden sollen Syntax und Semantik der Steuerungsdatei erläutert werden.

Eine Steuerungsdatei besteht aus ASCII Text und kann mit jedem beliebigen Texteditor erstellt werden. Die Einträge bestehen aus einzelnen Zeilen, denen ein Schlüsselwort vorangeht. Momentan gültige Schlüsselworte sind in Tabelle 4.2 aufgezählt.

Tabelle 4.2: *Syntax einer Datei vom Typ application/x-praxis-rpc*

Schlüsselwort	Wert	Beschreibung
REQUESTOR:	⟨url–Adresse⟩	URL, von welcher der Auftrag zur Ausführung eines Programmaufrufs erfolgte.
SERVER:	⟨url–Adresse⟩	*Toolserver*, der zur Auflösung von Paket– und Methodennamen in eine konkrete Laufzeitbeschreibung herangezogen werden soll.
AUTHENTICATION:	⟨string⟩	Dient zur Verifikation, daß es sich um einen autorisierten REQUESTOR handelt (wird in Version 1.00 noch nicht ausgewertet).
OPTIONS:		Wird momentan (Version 1.00) nicht genutzt. Dient zur Erweiterung der Semantik bei späteren Versionen.
ENVIRONMENT:	⟨string⟩ = ⟨string⟩, ⟨string⟩ = ⟨string⟩, ...	Liste von Attribut–Wert Paaren, die Umgebungsvariablen setzen.
VERSION:	⟨string⟩	Dient zur Identifikation unterschiedlicher Versionen dieser Datei. Aktuelle Version ist 1.00.
PACKAGE:	⟨string⟩	Name des Pakets, aus dem die auszuführende Methode stammt.
METHOD:	⟨string⟩	Name der auszuführenden Methode.
FILE:	⟨type⟩:⟨url–Adresse⟩	Dateiargument
PARA:	(STRING:⟨string⟩)| (INTEGER:⟨integer⟩)| (DOUBLE:⟨double⟩)| (URL:⟨url–Adresse⟩)	Argument vom Typ Parameter.

Den Schlüsselworten folgt jeweils der entsprechende Eintrag. Die Reihenfolge der ersten sieben Schlüsselwörter innerhalb der Datei ist beliebig. Sie dürfen jeweils nur einmal vorkommen. Die letzten beiden Schlüsselwörter dürfen mehrmals verwendet werden.

Im Gegensatz zu den oberen Schlüsselwort–Wert Paaren spielen bei den folgenden zwei Einträgen die Reihenfolge, in der sie auftreten, eine Rolle. Sie spezifizieren die Parametertypen und –werte der entsprechenden Methode. Es werden in Analogie zu Abschnitt 4.4.1 mehrere Parametertypen unterschieden.

Bei dem Typ FILE handelt es sich um eine Datei. Nach dem Schlüsselwort folgt der MIME–Typ

der Datei. Er wird zur Ermittlung der passenden Methode herangezogen. Dem Typ der Datei folgt schließlich die Adresse der Datei in Form einer URL. Hierbei kann es sich um ein beliebiges Protokoll (http:, ftp:, file:, etc.) handeln, es muß nur gewährleistet sein, daß der entsprechende Server das Protokoll versteht. Bei einem Parameter vom Typ FILE muß die Datei in einem ersten Schritt auf den lokalen Rechner geladen werden. Die Datei erhält lokal möglicherweise einen neuen Namen, der dann anschließend an die Methode übergeben werden muß.

Bei dem Typ URL handelt es sich um eine URL, die vor Aufruf der Applikation ausgewertet werden muß. Im Gegensatz zu einem Parameter vom Typ FILE wird das Resultat des URL–Requests aber nicht in Form einer Datei zur Verfügung gestellt, sondern das Resultat wird in der Kommandozeile an das zu startende Programm übergeben[13].

Beim Typ PARA handelt es sich um einen Parameter. Im Gegensatz zum Parametertyp FILE muß ein Parameter vom Typ PARA lediglich an die entsprechende Methode weitergeleitet werden. Der zweite Parameter ist optional und spezifiziert den Typ. Erlaubte Typen sind STRING, INTEGER und DOUBLE. Der dritte Eintrag enthält den eigentlichen Wert des Parameters. Hierbei kann es sich sowohl um einen einzelnen numerischen Wert oder String handeln (Bsp: "-140", "PostScript", etc.)[14], oder es kann sich wiederum um eine URL handeln. Im Gegensatz zu einem Parameter vom Typ FILE wird hierbei aber keine lokale Kopie der Daten angelegt, sondern die Werte werden anhand der gegebenen Typinformation interpretiert und der Methode direkt als Parameter übergeben.

Anhand der Reihenfolge des Auftretens dieser beiden letzten Schlüsselworte zusammen mit den übergebenen Informationen über den Typ des Parameters wird die Signatur der Methode ermittelt und die entsprechende Methode ausgewählt.

Ein Beispiel für eine vollständige Steuerungsdatei ist in Beispiel 4.5.2 gegeben:

In dem Beispiel wird eine abstrakte Methode `COMPARE` im Paket `IMAGEPROC` aufgerufen. Hierbei handelt es um die zweite Signatur aus Beispiel 4.4.1, die einen Iterator enthält. Der konkrete Toolaufruf besitzt vier Parameter, die auf den Iterator abgebildet werden. Die Parameter werden entsprechend der Signatur als zwei Tupel, bestehend jeweils aus dem Inhaltstyp der Datei im MIME-Format (Typ PARA) und der eigentlichen Datei (Typ FILE), die eine Bilddatei spezifiziert, interpretiert.

Die Aufgaben des *Toolstarters* lassen sich in folgende Teilbereiche untergliedern:

Kommunikation mit anderen Toolstartern: Stellt der *Toolstarter* fest, daß die Applikation auf einem anderen Rechner gestartet werden soll, so delegiert er die Aufgabe an den entsprechenden Rechner weiter. Dies wird in Form eines RPC Aufrufs durchgeführt. Der lokale *Toolstarter* behält dabei die Kontrolle über den Ablauf, der in dem Fall aber auf einem anderen Rechner stattfindet.

Vorbereiten des Toolaufrufs: Im vorbereitenden Schritt wird sozusagen die "Infrastruktur" hergestellt, die das Programm zum ordnungsgemäßen Ablauf benötigt. Hierzu zählt etwa das Setzen der Umgebungsvariablen.

Bereitstellen der benötigten Daten: Der ausführende *Toolstarter* ist dafür verantwortlich, daß die benötigten Daten zur Verfügung stehen. Der *Toolstarter* verwaltet hierzu einen

[13] Das macht natürlich nur dann einen Sinn, wenn das Ergebnis des URL–Requests aus einem Wort bzw. einer Zeile besteht.

[14] Die Werte müssen in Anführungszeichen gesetzt werden, um sie von einer echten URL unterscheiden zu können.

```
#
# Authentication part:
#
REQUESTOR: http://www.iai.fzk.de/praxis/wildflow/engine2/starter.cgi
SERVER: http://www.iai.fzk.de/praxis/wildflow/toolserver.cgi?id=1Ac4f63
AUTHENTICATION: 874353%56%$&;:_;#**'+3245-3252-5-wfewfaf-ew4w
VERSION: 1.00
#
# Abstract Signatur notation part:
#
PACKET:      IMAGEPROC
METHOD:      COMPARE
ENVIRONMENT: WF_DISP='misun32.iai.fzk.de:0.0', WF_USER='schmidt', \
             WF_GROUP='praxis', WF_LOCALHOST='misun32.iai.fzk.de'
PARA:        STRING : image/jpeg
FILE:        image/jpeg : ftp://ftp.imt.de/data/transfer/images/waf_1.jpeg
PARA:        STRING : image/gif
FILE:        image/gif : ftp://ftp.imt.de/data/images/gif/waf_clean.gif
```

Beispiel 4.5.2: Beispiel für eine Steuerungsdatei

"intelligenten" Cache[15], um jederzeit die benötigten Daten zur Verfügung stellen zu können und lädt Daten bei Bedarf in diesen Cache.

Toolaufruf: Der *Toolstarter* startet das Tool.

Übertragung der erzielten Ergebnisse: Erzielte Ergebnisse müssen nach Beendigung der Applikation an die Datenbank übertragen werden, wo sie anderen Aktivitäten zur Verfügung stehen.

Der *Toolstarter* kann somit in die zwei funktionale Einheiten *Application Launcher* und *Kommunikationsmodul* unterteilt werden:

1. Application Launcher: Dieser Teil ist für das Ausführen von rechnerspezifischen Kommandos wie dem Starten von bestimmten Applikationen oder dem Setzen von Umgebungsvariablen zuständig.
2. Kommunikationsmodul: In dem Modul wird der Kommunikationsmechanismus realisiert, der durch RPC das Starten von Applikationen mittels entfernt laufenden *Toolstartern* erlaubt. Hierzu wendet sich das lokale Kommunikationsmodul an das ensprechende Modul des entfernten Rechner, das dann wiederum den *Application Launcher* (siehe (1)) anweist, die Anweisung(en) auszuführen.

Die einzelnen Komponenten sowie die Kommunikationskanäle sind in Abbildung 4.14 graphisch dargestellt.

Das Zusammenspiel zwischen den einzelnen besprochenen Komponenten *Toolserver* und *Toolstarter* wird im folgenden näher beschrieben. Neben der textuellen Beschreibung ist der Ablauf in Abbildung 4.15 graphisch dargestellt.

[15] siehe Glossar Seite 180

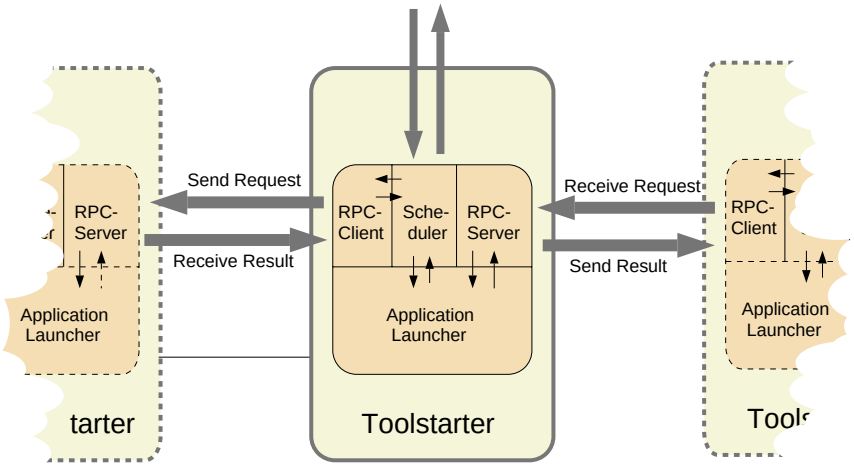


Abbildung 4.14: Toolstarter-*Komponenten und Zusammenspiel*

Ausgangspunkt ist der Moment, an dem der *Toolstarter* die in Abschnitt 4.5.3 beschriebene Steuerungsdatei erhalten hat. (Abbildung 4.15 Punkt (1)).

Nachdem ein *Toolstarter* auf dem Klient eine Steuerungsdatei empfangen hat, muß die darin enthaltene abstrakte Beschreibung zuerst in eine konkrete Laufzeitbeschreibung umgewandelt werden. Dazu wendet sich der Klient an die mit dem Schlüsselwort SERVER angegebene URL. Dabei gibt er folgende Parameter an:

- PAKET=⟨Paketname⟩
- METHOD=⟨Methodenname⟩
- SIGNATUR=⟨Liste der Parametertypen⟩[16]
- ARCHITEKTUR=⟨Rechnerarchitektur⟩
- OPERATINGSYSTEM=⟨Betriebssystemname⟩
- USERNAME=⟨Benutzername⟩
- COMPUTERNAME=⟨Rechnername⟩

Während die ersten drei Angaben unverändert aus der Steuerungsdatei übernommen werden (abstrakte Beschreibung der auszuführenden Applikation), fügt der Klient mit den nächsten vier Angaben spezielle Informationen über seine Laufzeitumgebung und seine Identität hinzu (Abbildung 4.15 Punkt (2)).

Mittels diesen Informationen ist es dem *Toolserver* möglich, die konkrete Laufzeitumgebung aus der Datenbank zu extrahieren. Diese besteht u. a. aus dem konkreten Aufruf, bei dem die Aufrufparameter durch `$<x>` Platzhalter repräsentiert werden.

So kann z. B. der konkrete Aufruf aus Beispiel 4.5.2 folgendermaßen aussehen:

```
/usr/local/bin/image_compare -t $1 $2 -t $3 $4
```

[16] FILE, PARA Einträge entsprechend Beispiel 4.5.2

Das konkrete Kommando oder Skript wird anschließend zusammen mit Angaben über den Typ der Applikation und dem Zielrechner, auf dem die Applikation ablaufen soll, an den *Toolstarter* zurückgeschickt (Abbildung 4.15 Punkt (3)).

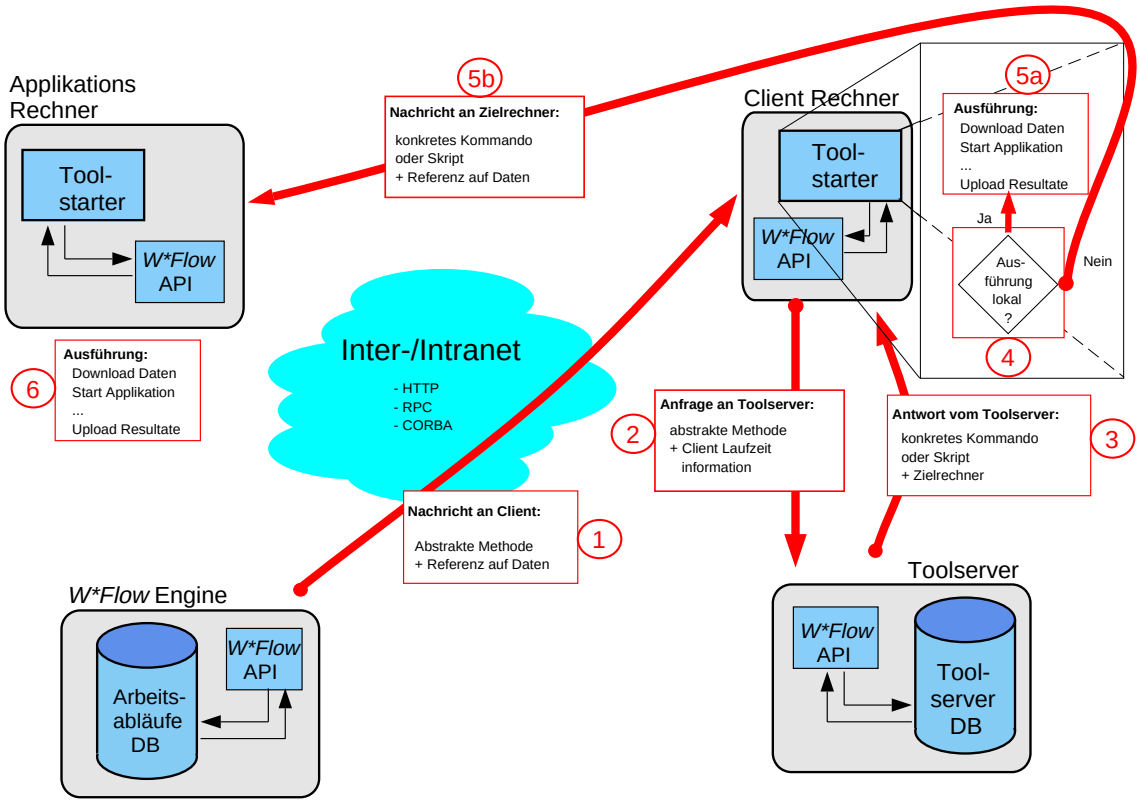


Abbildung 4.15: *Ablauf eines Toolaufrufes*

Der *Toolstarter* ermittelt nun anhand des Rechnernamens, der ihm vom *Toolserver* mitgeteilt wurde, ob die Applikation lokal oder auf einem entfernten Rechner gestartet werden soll (Abbildung 4.15 Punkt (4)).

Soll die Applikation lokal gestartet werden, werden die in der Steuerungsdatei spezifizierten Daten auf den lokalen Rechner geladen und anschließend die Applikation mit den Daten gestartet.

Dazu substituiert der *Toolstarter* die `$<x>`–Platzhalter im Aufrufstring durch die konkreten, lokal heruntergeladenen Dateien. Bei obigem Beispiel könnte die Ersetzung folgendermaßen aussehen:

```
/usr/local/bin/image_compare -t image/jpeg /tmp/waf_1.jpeg \
                             -t image/gif /tmp/waf_clean.gif
```

Dabei sind die Dateinamen */tmp/waf_1.jpeg* und */tmp/waf_clean.gif* vom lokalen System gewählt worden. Im allgemeinen wird das lokale System zwar versuchen, möglichst die gleichen Namen zu vergeben wie auf dem Server, was aber nicht immer garantiert werden kann.[17]

Anschließend kann die Applikation auf dem Rechner gestartet werden (Abbildung 4.15 Punkt (5)) und nach Beendigung können die erzielten Ergebnisse übertragen werden.

[17] Z. B. wenn das Betriebssystem des Klienten bestimmte Dateinamen nicht zuläßt, oder die Dateien von einen CGI–Programm ausgewählt werden.

Wird in Schritt (4) der Abbildung 4.15 ermittelt, daß die Applikation auf einem entfernten Rechner gestartet werden soll, so wendet sich der *Toolstarter* an den auf dem Zielrechner laufenden *Toolstarter*-Daemon und überträgt ihm die Ausführung. Dazu wird dem entfernt laufenden *Toolstarter* die Steuerungsdatei aus Schritt (1) übergeben, wobei jedoch die beiden Einträge `PACKET` und `METHOD` durch das konkrete Kommando oder Skript ersetzt werden (Abbildung 4.15 Punkt (5b)). Anschließend erfolgt der Start der Applikation analog zu Schritt (5a) auf dem lokalen Rechner. Während des Ablaufs der Applikation wird der lokale *Toolstarter* über eventuell auftretende Fehler oder Warnungen informiert.

4.6 Zusammenfassung

Ziel des Kapitels war die Vorstellung eines flexiblen Mechanismus zum Aufruf und der Spezifikation von Programmen in heterogenen Umgebungen. Ausgehend von Ansätzen aus der Literatur wurde eine neue Lösung vorgestellt, die eine Teilung von rechnerspezifischen Informationen und rechnerunabhängigen Informationen realisiert. Das Problem der Zuordnung zwischen den beiden Informationsblöcken wurde durch einen formalisierbaren Ableitungsmechanismus gelöst, der eine vollautomatische Zuordnung von abstrakter Beschreibung, wie sie im Rahmen der Workflowspezifikation vorliegt, zum konkreten Programm durchführt. In dieser Abbildung ist gleichzeitig eine flexible Benutzerkonfiguration enthalten, so daß die Forderung nach Flexibilität des Benutzers ebenfalls erfüllt wird.

Das Konzept wurde anschließend im Rahmen der Entwicklung von $\mathcal{W}*FLOW$ durch die beiden Komponenten *Toolserver* und *Toolstarter* realisiert.

Kapitel 5

Anwendungsbeispiel

Nachdem mit den beiden Komponenten *Toolserver* und *Toolstarter* die Vorstellung der Bausteine des $\mathcal{W}*FLOW$–Baukastens beendet wurde, soll nun, basierend auf einem realen Szenario, die Entwicklung einer konkreten Arbeitsumgebung mittels der $\mathcal{W}*FLOW$–Komponenten gezeigt werden. Das Szenario entstammt dem DEMIS–Projekt[1], dessen Aufgabenstellung die Bereitstellung eines Verbundes von Simulations– und Optimierungswerkzeugen mit dem Ziel Gesamtmodelle im Bereich der Mikrosystemtechnik zu realisieren und zu optimieren [JGL99], ist.

Die hier vorgestellte $\mathcal{W}*FLOW$–Arbeitsumgebung basiert zu einem großen Teil auf der in Abschnitt 1.2.3.1 vorgestellen Simulation mittels *CFX*. Einzelne Aspekte dieser Simulation wurden auch später in der Arbeit, beispielsweise bei der Vorstellung des Container– und Aktivitätenkonzepts wieder aufgegriffen und beleuchtet, so daß auf die dort gewonnen Erkenntnisse in Bezug auf Modellierung der Container und Aktivitäten aufgebaut werden kann. Im vorliegenden Kapitel liegt der Fokus auf dem Zusammenspiel der Komponenten, auf anwendungsspezifischen Erweiterungen, Anbindung an Oberflächen und dem Einsatz des Baukastens im Rahmen einer zu realisierenden Arbeitsumgebung.

Das Kapitel ist wie folgt aufgebaut. Zuerst erfolgt eine Beschreibung des vorliegenden Anwendungs-Szenarios. Anschließend wird eine Methodik zum Aufbau von Entwicklungsumgebungen mittels den $\mathcal{W}*FLOW$–Komponenten vorgestellt, in deren Rahmen wichtige Aspekte zur Arbeit mit dem Baukasten beleuchtet werden. Den Abschluß bildet die Vorstellung einer Reihe von möglichen Erweiterungen der entwickelten Arbeitsumgebung.

5.1 Vorstellung des Szenario

Ein großer Teil des DEMIS–Projektes befaßt sich mit der Modellbildung. In dem hier vorgestellen Fall wird eine Modellbildung von medizinischen Teststreifen durchgeführt, an die sich eine Evaluierung der Modelle anhand exemplarisch hergestellter Teststrukturen anschließt [PSH+99]. Dazu werden Messungen zur Fließgeschwindigkeit von Flüssigkeiten in vorgegebenen mikrofluidischen Strukturen durchgeführt. Der Aufbau der Messvorrichtung ist in Abbildung 5.1 dargestellt.

Bei der Messung wird eine Mikrostruktur unter ein Mikroskop gelegt und auf ihr mittels Pipette eine Flüssigkeit am Kanalanfang aufgebracht. Durch die Kapillarkräfte fließt die Flüssigkeit durch das Kanalröhrchen. Eine Kamera, die an das Mikroskop angeschlossen ist, filmt das

[1] **DE**signoptimierung für **MI**kro**S**ysteme

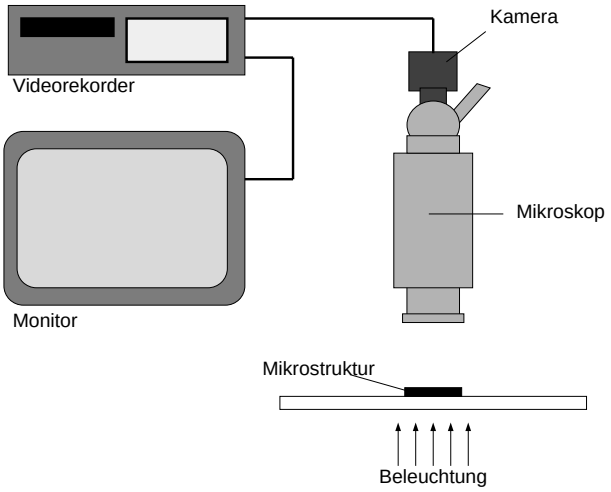


Abbildung 5.1: *Messeinrichtung (aus [PSH+ 99])*

Durchwandern der Flüssigkeit und zeichnet den Vorgang auf einem Videorekorder auf. Mit diesem Versuchsaufbau läßt sich die Durchlaufzeit der Flüssigkeit durch das Röhrchen auf 1/25 sec. genau bestimmen.

Ergänzend zu den realen Messungen sollen geeignete Simulationsmodelle der Teststreifen entwickelt werden. Die zu realisierenden Modelle sollen bei der Entwicklung und Designoptimierung medizinischer Teststreifen das aufwendige und teure *Trial- and Error* Verfahren überflüssig machen und somit den Entwicklungsaufwand stark verringern. Da bei der Simulation von fluidischen Problemen im Mikrobereich bisher nur wenige Erfahrungen vorliegen und auch die meisten Simulationswerkzeuge keine Modellierung von im Mikrobereich wichtigen Phänomenen, wie Kapillarkräfte und dem Verhalten an Phasengrenzen vorsehen, ist ein wichtiger Teil des Projektes die Validierung der Simulationsergebnisse anhand der an den realen Mikrokomponenten vorgenommenen Tests.

Die zu lösenden Aufgaben lassen sich somit auf hoher Abstraktionsebene entsprechend Abbildung 5.2 darstellen. Hierbei handelt es sich erstmal um zwei räumlich getrennt durchzuführende Aufgaben, nämlich der Entwicklung einfacher Teststrukturen und der Durchführung von Messungen durch eine Firma einerseits und der Modellbildung und Simulation andererseits. Diese zwei Aufgaben sind aber miteinander verzahnt, da die entwickelten Mikrostrukturen modellhaft nachzubilden und die realen Messungen der Flußgeschwindigkeit mittels Simulation nachzuvollziehen sind.

Die Herstellung der Mikrostrukturen und die sich anschließenden Versuche zur Fließgeschwindigkeit erfolgen ohne durchgehende Rechnerunterstützung. Es werden lediglich die notwendigen Daten und Parameter der hergestellten Mikrostrukturen und der Versuche mitprotokolliert. Die protokollierten Daten umfassen beispielsweise die Abmessungen der Mikrokanäle, das Material, etc. Weiterhin sind für jede Mikrostruktur, die im Rahmen des Versuchsaufbaus ermittelten Fließgeschwindigkeiten sowie eine Reihe der beim Versuch vorliegenden Randbedingungen, wie Temperatur, etc., von Bedeutung. Diese Daten stellen die Basis für die durchgeführten Modellierungen und Simulationen dar.

Im Gegensatz zu der eher manuellen Vorgehensweise bei der Herstellung der Teststrukturen und den Messungen bei der Firma bietet sich am IAI, aufgrund der durchgehend rechnergestützten Arbeitsschritte, eine Einbettung der einzelnen Werkzeuge in einen Workflow an. So werden im

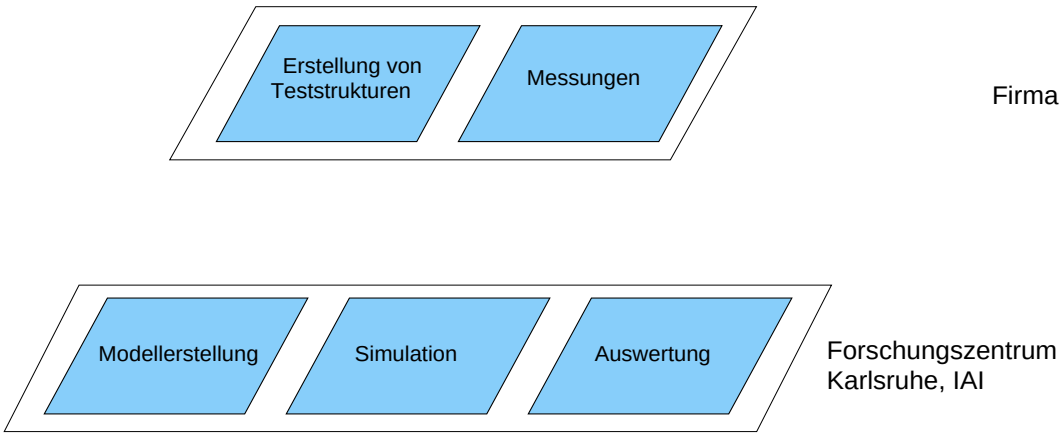


Abbildung 5.2: *Top Level Workflows*

Rahmen der durchgeführten Modellierungen und Simulationen folgende Programme/Werkzeuge eingesetzt:

Text-Editoren: Je nach Anwender und Rechnerplattform kommen die Editoren *emacs*, *vi* und *notepad*, z. B. zur Bearbeitung der Steuerungsdateien, zum Einsatz.

Modellierungstools: Zur Erstellung des Fluidmodells wird das Programm *ANSYS* unter dem Betriebssystem *Solaris* eingesetzt. Weiterhin wird noch das Programm *Meshbuild* (AEA–Technologies) zur Erzeugung der für das Simulationsprogramm notwendigen Gitterzerlegung des Modells benötigt.

Simulation: Als Simulator wird das Programm *cfx-solver* aus dem *CFX* Programmpaket (AEA–Technologies) verwendet. Zusätzlich kommt das Public Domain Programm *HistoScope* zur Überwachung des Simulationslaufes zum Einsatz.

Zur Visualisierung der Simulationsergebnisse kommen die Programme *cfx-graph*, *cfx-visualize* und das Public Domain Programm *FieldView* zum Einsatz.

Präsentation: Die Präsentation und Dokumentation der erzielten Ergebnisse wird mit dem Postscript–Viewer *GhostView*, dem Konvertierungstool *Acrobat Distiller* (Postscript nach PDF Konverter) und dem PDF–Viewer *Acrobat Reader* durchgeführt .

Sonstige: Neben den obigen Programmen werden noch eine Reihe von selbstentwickelten Programmen benötigt, so z. B. das Programm *modgen* zur Konvertierung der mittels *ANSYS* erstellten Modelldatei in das von *Meshbuild* erwartete Dateiformat.

Im konkreten Szenario sind neben dem Entwickler der Arbeitsumgebung drei Personen an der Lösung der Aufgabe beteiligt.
Die erste Person ist hauptsächlich mit der Erstellung von Modellen beschäftigt. Dies umfaßt, neben der eigentlichen Erstellung des Modells, auch die Durchführung von einfachen Simulationen und die Begutachtung der Resultate. Der Schwerpunkt der Aufgabe liegt aber in der Modellbildung.

Die zweite Person befaßt sich schwerpunktmäßig mit der Simulation. Dazu nutzt sie die vom Modellierer erstellten Modelle und führt mit diesen eine Vielzahl unterschiedlicher Simulationen

durch. Zu den Aufgaben gehört die Ermittlung der Simulationsparameter, der optimalen Gitterzerlegung, der Simulatoransteuerung sowie eventuell die Realisierung und Einbindung zusätzlicher Rechenmodule in den Simulator. Daneben werden von der zweiten Person eine Reihe von Visualisierungstools eingesetzt, um die Ergebnisse der Simulationen zu begutachten.

Die dritte Person ist schließlich für die Präsentation der erzielten Ergebnisse, beispielsweise im Rahmen von durchgeführten Workshops, zuständig. Hierzu werden die vorhandenen Simulationsergebnisse ausgewertet, unter verschiedenen Gesichtspunkten visualisiert (Postscript, Videosequenzen, ...) und in unterschiedliche Dokumente (Reports, Folien, Onlinepräsentationen) eingebunden. Es ist auch nicht auszuschließen, daß bestimmte Simulationsläufe mit leicht modifizierten Parametern wiederholt werden.

Das Zusammenspiel und die von den beteiligten Personen eingesetzten Werkzeuge und Plattformen sind in Abbildung 5.3 nochmals graphisch dargestellt.

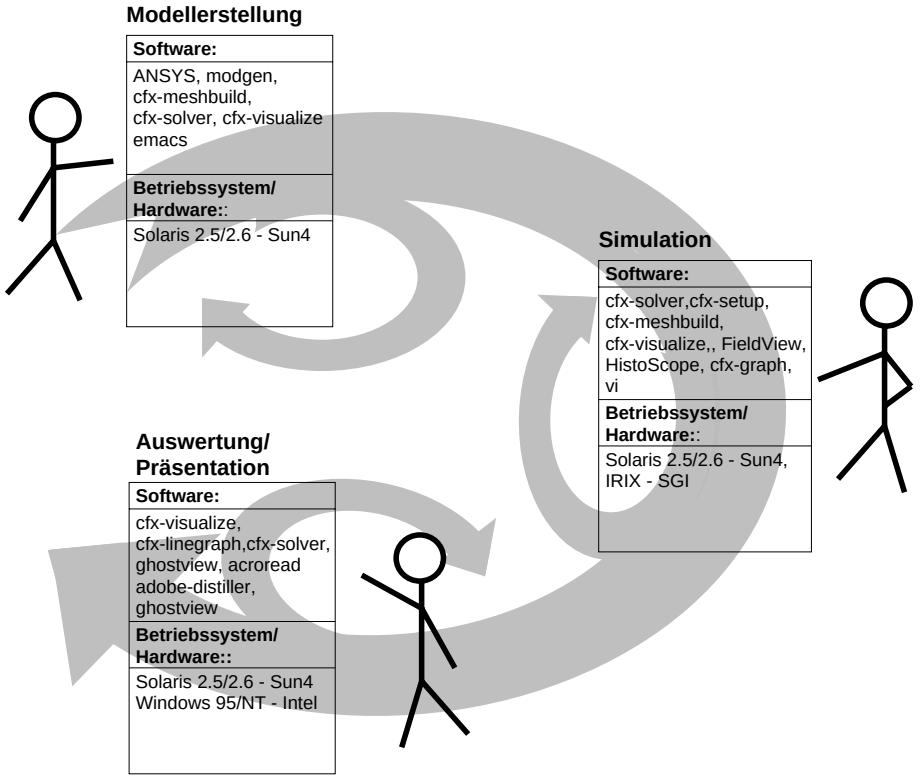


Abbildung 5.3: *Aufgabenverteilung und eingesetzte Werkzeuge/Plattformen*

5.2 Entwurf der Arbeitsumgebung

Nach Vorstellung des Szenarios und der Beschreibung der anfallenden Aufgaben, der zum Einsatz kommenden Werkzeuge und beteiligten Personen wird im folgenden die Arbeitsumgebung zur Integration der Arbeitsschritte entworfen.

5.2.1 Mikrostrukturentwicklung und Messungen

Da bei der Herstellung der Mikrokanäle und der anschließenden Tests keine Rechnerunterstützung notwendig bzw. realisiert ist, genügt es, die anfallenden Daten in einem Container geeigneter Struktur abzulegen. Eine solche Containerstruktur ist in Abbildung 5.4 dargestellt.

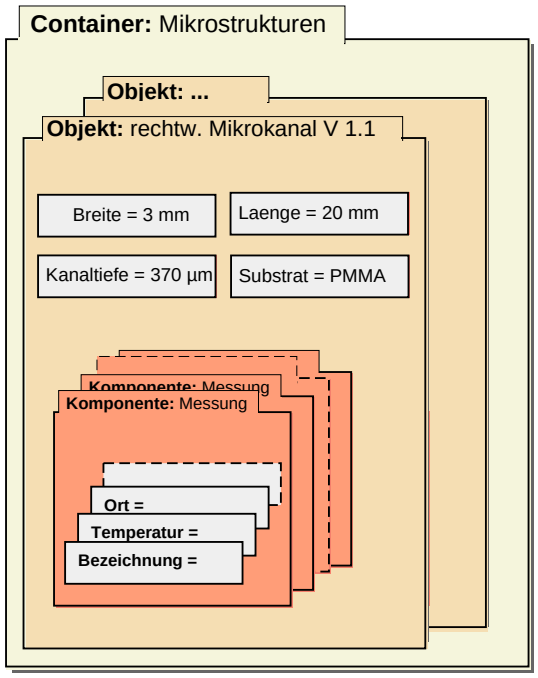


Abbildung 5.4: *Container zur Aufnahme der Daten und Messungen für eine Mikrostruktur*

Zur Eingabe der anfallenden Daten muß dem oder den an der Durchführung der Aufgaben beteiligten Personen ein Benutzerinterface zur Verfügung gestellt werden, in das die anfallenden Parameter und Ergebnisse eingetragen werden können. Aufgrund der räumlichen Verteilung zwischen den beiden Teilaufgaben bietet sich eine WWW basierte Lösung an. Sie hat den Vorteil, daß keine zusätzlichen Programme[2] installiert werden müssen und eine weitestgehende Plattformunabhängigkeit erreicht wird. Dazu ist es lediglich notwendig, eine WWW Oberfläche für den Container zu realisieren, innerhalb der die notwendigen Parameter über die Mikrostruktur und die zugehörigen Messungen eingebracht werden. Abbildung 5.5 zeigt ein Formular zur Eingabe einer neuen Messung für eine bestehende Mikrostruktur.

Das zur Verfügung gestellte Formular erlaubt es, die notwendigen Herstellungsparameter sowie die erzielten Messergebnisse in das Workflowsystem einzubringen. Die Realisierung einer WWW Oberfläche stellt aufgrund der Schnittstellensprache *Perl* keine große Anforderungen an den Entwickler und kann mit geringem Zeitaufwand realisiert werden. Je nach Anbindung der Versuchsanordnung aus Abbildung 5.1 an ein Rechnersystem können dabei auch multimediale Daten, z. B. die bei einer Messung erstellte Videosequenz, mit im Container abgelegt werden. Die Daten stehen anschließend im Container bereit und können vom zu entwickelnden Modellierungs– und Simulations-Workflow genutzt werden.

5.2.2 Modellierung und Simulation

Wie bereits in Abschnitt 1.2.3.1 beschrieben, handelt es sich bei der Modellerstellung um einen iterativen Vorgang, der aus den sich wiederholenden Schritten Modellmodifikation, Simulation und Begutachtung der Simulationsergebnisse besteht. Ein kompletter Zyklus, der auch die Ein-/Ausgabebeziehungen zwischen Daten und Programmen darstellt, ist in Abbildung 5.6 zu sehen.

Die Entwicklung der zugehörigen $\mathcal{W}*FLOW$–Arbeitsumgebung geht in mehreren Schritten vor

[2] außer eventuell ein Standardbrowser, sofern auf dem System noch nicht vorhanden

Abbildung 5.5: *WWW Schnittstelle zum Versuchscontainer*

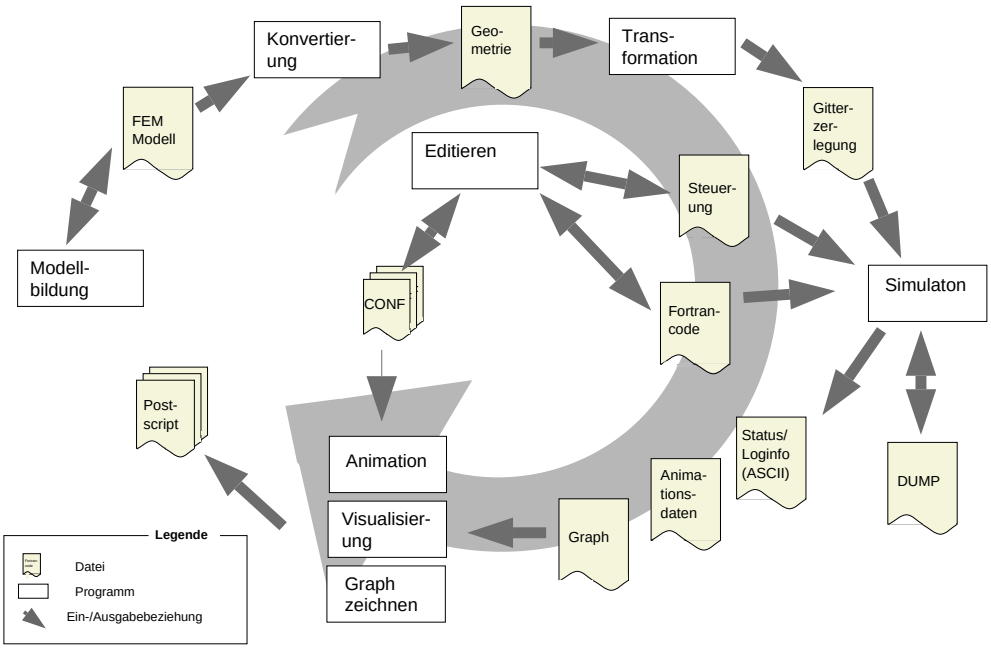


Abbildung 5.6: *Modellierungszyklus*

sich. Zu Beginn erfolgt die Modellierung der Container. Anschließend werden zusammengehörende Einzelschritte zu Aktivitätenobjekten zusammengefaßt und mit den Containern verknüpft. Im nächsten Schritt erfolgt die Festlegung und Realisierung der speziellen Funktionalität der Aktivitätenobjekte, gefolgt von der Konfiguration des *Toolservers*, d. h. die Verknüpfung der bisher abstrakten Aktionen mit konkreten Rechnern und Programmen. Den Abschluß bildet die Festlegung des Zusammenspiels zwischen den Aktivitäten.

5.2.2.1 Modellierung der Container

Im ersten Schritt erfolgt die Realisierung der für die Anwendung notwendigen Container zur Aufnahme/Verwaltung der anfallenden Daten, der Repräsentation von Beziehungen zwischen den Datensätzen und der Hinzunahme von Metainformationen. Die Modellierung der *Modell–* und *Steuerdatencontainer* für das hier vorliegende Szenario wurde bereits beispielhaft in Abschnitt 3.1.3 bei der Vorstellung des Containerkonzepts durchgeführt, eine Modellierung des Ergebniscontainers ist in Abbildung 3.11 dargestellt. Die Modellierung wird von dort übernommen und ist in Abbildung 5.7 nochmals schematisch dargestellt.

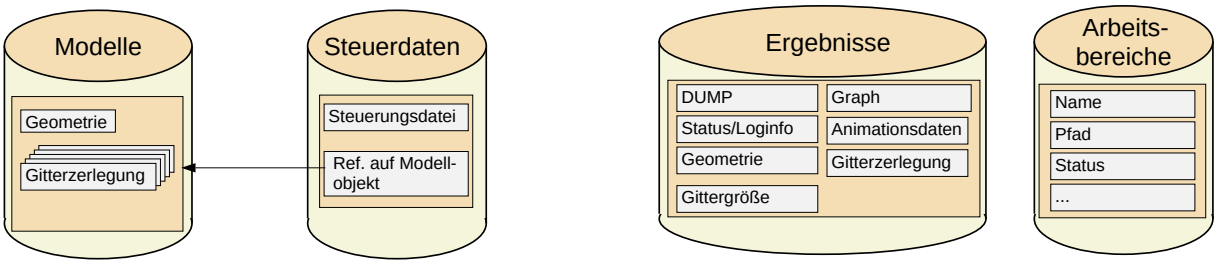


Abbildung 5.7: *Container*

Für das konkrete Szenario wird noch ein zusätzlicher Container eingesetzt, der keine für die Anwendung notwendigen Daten enthält, sondern zur Synchronisation genutzt wird. Der Container enthält eine Reihe von Objekten, welche *CFX* Simulationsumgebungen repräsentieren. Eine *CFX* Simulationsumgebung entspricht einem Unterverzeichnisbaum mit einer bestimmten von *CFX* vorgegebenen Struktur und der den von den *CFX*Werkzeugen benötigten Dateien. Das Verzeichnis stellt das Arbeitsverzeichnis der Simulation dar. Ein Simulationslauf ist stets an eine solche Verzeichnisstruktur gebunden, und es kann auch immer nur eine Simulation innerhalb eines Arbeitsverzeichnisses ablaufen. Sollen mehrere Simulationen gleichzeitig ablaufen, so müssen sie von verschiedenen Verzeichnissen aus gestartet werden. Damit mehrere Simulationen parallel, ohne sich zu beeinflussen, ablaufen können, liefert der Container auf Anfrage eine zur Verfügung stehende Simulationsumgebung mit einem momentan nicht benutzten Arbeitsverzeichnis zurück. Die Objekte im Container Arbeitsbereiche enthalten Informationen über den Verzeichnispfad, den Status der Umgebung (frei, belegt, unbekannt), etc. einer Arbeitsumgebung. Die Funktionsweise der Objekte ist in diesem Fall mit einem Semaphor zu vergleichen, der den Zugang zu beschränkten Ressourcen (den Simulationsumgebungen) regelt.

5.2.2.2 Identifikation der Aktivitäten

Die Modellierung der Aktivitäten umfaßt in einem ersten Schritt die Identifikation der zusammengehörenden Arbeitsschritte. Das zugrundeliegende Konzept zur Gruppierung basiert, wie in Abschnitt 3.2.1 erläutert, auf der Nutzung gemeinsamer Daten.
In dem genannten Abschnitt ist auch eine Gruppierung der einzelnen Aktionen in die Aktivitäten *Modelldatei bearbeiten*, *Steuerungsdatei bearbeiten*, *Simulation* und *Auswertung* beschrieben. Diese Gruppierung ist in Abbildung 5.8 zusammengefaßt noch einmal dargestellt.

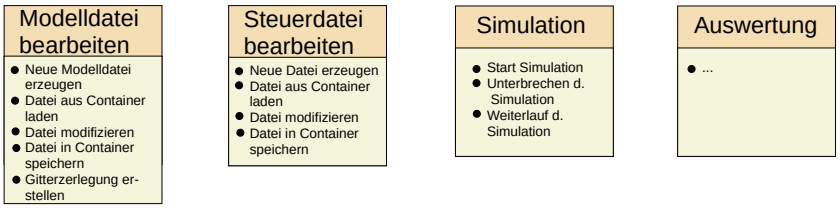


Abbildung 5.8: *Aktivitäten des Anwendungsszenarios*

Die anschließend durchzuführende Zuordnung der Container zu den Aktivitäten ist in Abbildung 5.9 zu sehen.

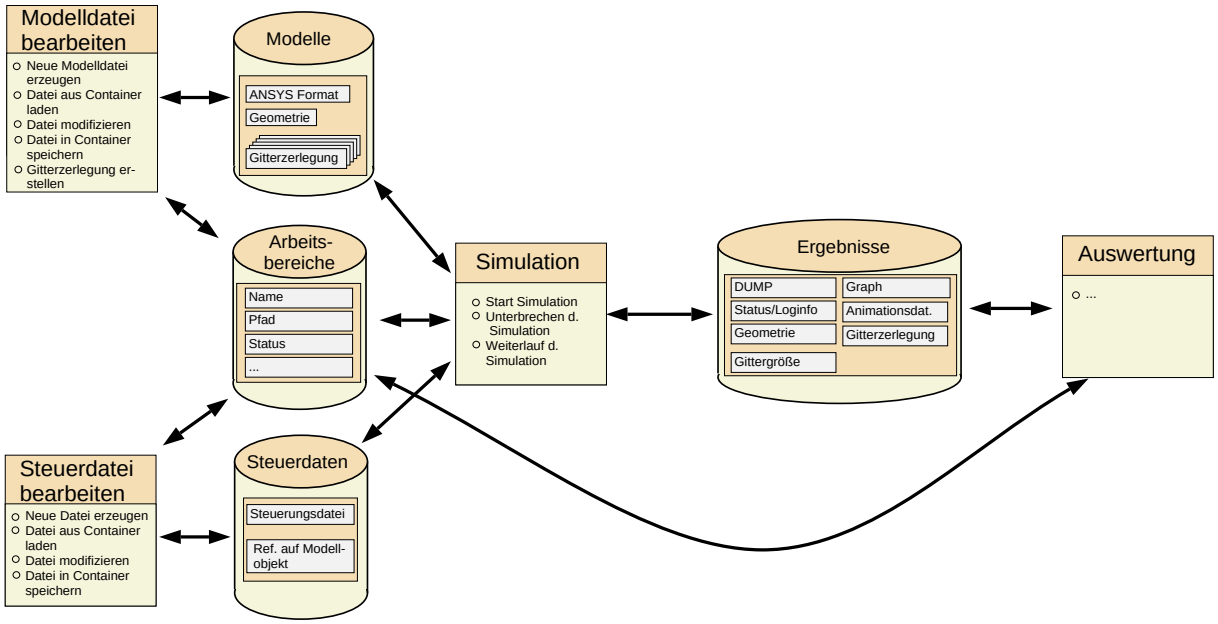


Abbildung 5.9: *Beziehungen zwischen Containern und Aktivitäten des Anwendungsszenarios*

5.2.2.3 Festlegung der Funktionalität der Aktivitätenobjekte

Wie die Festlegung der Funktionalität einer Aktivität erfolgt, wird beispielhaft an der Aktivität *Steuerungsdatei bearbeiten* gezeigt. Zu den Aufgaben gehört die Festlegung der auszuführenden Aktionen, der Definition der Zustände und Zustandsübergänge sowie der speziellen Aktivitätenfunktionalität.

Wie der Abbildung 5.9 zu entnehmen ist, enthält die Aktivität Steuerungsdatei bearbeiten die folgenden Aktionen und Arbeitsschritte:

- Laden einer Steuerungsdatei
- Speichern einer Steuerungsdatei
- Steuerungsdatei modifizieren
- Anlegen einer neuen Steuerungsdatei

Durch den Einsatz der Container kann von der ursprünglichen Dateischnittstelle abstrahiert und mit semantisch höheren Datenobjekten gearbeitet werden. Trotzdem müssen die Dateien den

Programmen an ihrer ursprünglichen Schnittstelle (dem Dateisystem) zur Verfügung stehen. So erwartet *CFX* alle notwendigen Dateien in dem oben beschriebenen Unterverzeichnisbaum, der auch gleichzeitig die Simulationsumgebung repräsentiert.

Eine Unterstützung des Anwenders wird nun dadurch erreicht, daß der zu realisierenden Arbeitsumgebung die Verwaltung des benötigten Verzeichnisbaums übertragen wird, und der Anwender sich auf die Arbeit mit den Containern und den darin enthaltenen Objekten beschränken kann. Detailliert sind folgende Aktionen für die einzelnen Arbeitsschritte der Aktivität *Steuerungsdatei bearbeiten* zu tätigen:

Laden einer Steuerungsdatei: Eine zu spezifizierende Steuerdatei wird aus dem Container in das Simulationsverzeichnis kopiert.

Speichern einer Steuerungsdatei: Die aktuelle, sich im Arbeitsverzeichnis befindende, Steuerdatei wird in den Container übertragen. Dabei werden den Steuerungsdateien bei der Übertragung in den Container zusätzliche Verwaltungsinformationen zugeordnet, die teils automatisch, teils manuell hinzugefügt werden.

Steuerungsdatei modifizieren: Die aktuelle Steuerungsdatei im Arbeitsverzeichnis wird zur Bearbeitung in einen Editor geladen.

Anlegen einer neuen Steuerungsdatei: Eine neue, initiale Steuerungsdatei wird im aktuellen Arbeitsverzeichnis angelegt.

Der zugehörige Zustandsautomat ist in Abbildung 5.10 zu sehen. Er sorgt dafür, daß die Arbeitsumgebung in der Lage ist, festzustellen, ob noch ungespeicherte Steuerungsdateien vorhanden sind.

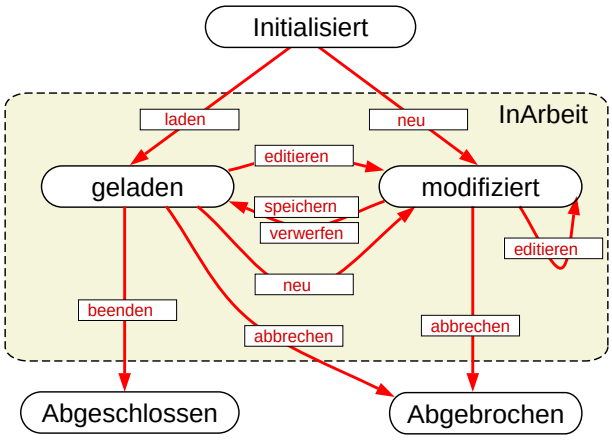


Abbildung 5.10: *Zustandsautomat für die Aktivität* Steuerungsdatei bearbeiten

Neben den obigen Aktionen macht es Sinn, noch weitere Aktionen für die Aktivität *Steuerungsdatei bearbeiten* festzulegen. So ist es beispielsweise bei der Realisierung der Benutzerschnittstelle sinnvoll, dem Benutzer alle vorhanden Steuerungsdateien anzuzeigen, von denen er dann eine auswählen kann. Eine zusätzliche Methode der Aktivität, welche alle Steuerungsdateien ausgibt, kann den Entwickler dabei unterstützen.

5.2.2.4 Realisierung der Funktionalität

In diesem Schritt wird die zuvor festgelegte Funktionalität realisiert. Das erfolgt entweder, indem für ein Aktivitätenobjekt zusätzliche Methoden bereitgestellt werden, wie beispielsweise die Methode zur Auflistung aller im Container vorhanden Steuerungsdateien, oder die Anbindung von Funktionen an die internen und externen Aktionshandler aus Definition 3.12.
Am Beispiel der Aktivität *Steuerungsdatei bearbeiten* werden im folgenden die unterschiedlichen Möglichkeiten gezeigt.

Ein Beispiel für die Anbindung eines externen Skripts, das über die $\mathcal{W}$*FLOW*–API Zugriff auf die Aktivitäteninstanz besitzt (*Whitebox*–Integration), ist der Arbeitsschritt *Steuerungsdatei laden* (Zustandsübergang laden in Abbildung 5.10). Aufgabe des Arbeitsschrittes ist das Kopieren einer Steuerungsdatei aus dem Container in das aktuelle Arbeitsverzeichnis. Dies erfordert den Zugriff auf die Containerdaten und gleichzeitig muß auf das Verzeichnis, innerhalb dem die Simulation ablaufen soll, zugegriffen werden. Dazu wird als erstes im *Toolserver* die *abstrakte Methodenbeschreibung* CFX::download_control_file angelegt. Anschließend wird das Skript aus Beispiel 5.2.1 als konkreter Skript-Eintrag der Methode CFX::download_control_file für das *Solaris* Betriebssystem im *Toolserver* registriert. Bei der Aktivierung wird das Skript auf das Zielsystem übertragen und dort ausgeführt. Das Skript liest mittels der API das aktuelle Simulationsverzeichnis (1) aus der Aktivitäteninstanz aus und greift auf den Container zu (2), indem sich die spezifizierte Steuerungsdatei befindet, liest die Datei aus (3) und kopiert sie in das Arbeitsverzeichnis (4). Anschließend wird der Name der Steuerungsdatei in der Aktivitäteninstanz in einem dafür vorgesehenen Attribut ACTIVE_CONTROL_FILE registriert (5).

```
#!/usr/bin/perl -w

use WildFlow::API;

my $task = WildFlow::Task->new(@ARGV);

my $control_id = $ARGV[0];
begin 'read', sub {

  my $act = $task->get_activity_instance();
  my $path = $act->attribute('ACTIVE_ENVIRONMENT');                    # (1)

  my $container = $act->get_container('Modelle');                      # (2)
  my $obj = $container->get_object($control_id);
  my $cmp = $obj->get_component('Datei');
  my $content = $cmp->get_content();                                   # (3)

  my $file_name = WildFlow::get_unique_name($path);
  open(LOG,"+>$file_name") || die "ERROR $!";                          # (4)
  print LOG $content;                                                  #
  close (LOG);
  $act->attribute('ACTIVE_CONTROL_FILE', $file_name);                  # (5)
};
die "$@\n" if $@;
```

Beispiel 5.2.1: $\mathcal{W}$*FLOW*–Skript zum Kopieren einer Steuerungsdatei in das Arbeitsverzeichnis

Der Arbeitsschritt *Steuerungsdatei modifizieren* ist ein Beispiel für eine *Blackbox*–Integration. Analog zur *Whitebox*–Integration wird eine abstrakte Methode (EDIT im Paket CFX) definiert, für die dann anschließend verschiedene konkrete Programmaufrufe definiert werden können. Im Unterschied zur ersten Methode wird aber kein Skript registriert, sondern ein auf dem jeweiligen System vorhandener Editor aufgerufen. Beispielhaft sind hier Einträge für Sun– und Windows Rechner dargestellt. Im vorliegenden Fall (Abbildung 5.11) wird davon ausgegangen, daß für die beteiligten Arbeitsplatzrechner eine einheitliche Sicht auf das gemeinsame Dateisystem existiert, so daß der Zugriff auf die Datei gewährleistet ist.

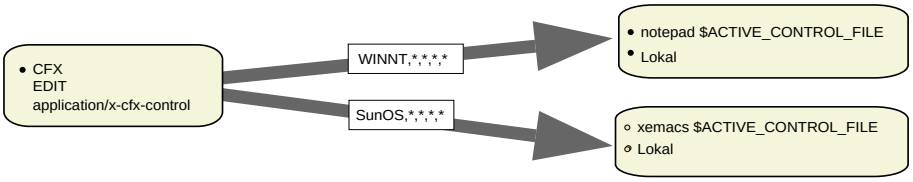


Abbildung 5.11: *Einbindung zweier Editoren mittels* Blackbox*–Integration*

Als ein weiteres Beispiel wird die Realisierung der Methode zur Auflistung aller Namen der sich im Container befindlichen Steuerungsdateien gezeigt. Beispiel 5.2.2 zeigt die Definition (1) und Registrierung (2) der entsprechenden Methode für ein Aktivitätenobjekt. Die Funktionalität wird direkt als Methode des Aktivitätenobjekts *Steuerdatei bearbeiten* realisiert und steht anschließend allen Instanzen des Aktivitätenobjekts zur Verfügung. Die Methode kann anschließend auch über die API aus aufgerufen werden.

```
my $code = sub  {                                                    # (1)
    my $self = shift;

    my $container = $self->get_container('Steuerungsdaten');
    my @objects = $container->object_list();

    my @results;
    foreach my $object (@objects) {
        push @results, $object->NAME();
    }
    return @results;
};

$activity->add_method("get_control_file_names", $code);              # (2)
```

Beispiel 5.2.2: Methode get_control_file_names()

Bei der Durchführung des Arbeitsschrittes *Steuerungsdatei speichern* werden, neben der Übertragung der eigentlichen Datei in den Container, zusätzliche, z. T. vom Anwender anzugebende, Informationen benötigt. Dies erfordert die Kommunikation des System mit dem Anwender. Wie eine Oberfläche für einen Arbeitsschritt der Arbeitsumgebung realisiert wird, wird im folgenden Abschnitt gezeigt.

5.2.2.5 Anbindung an eine Benutzeroberfläche

Nachdem die Aktivität mit der notwendigen Funktionalität versehen wurde, muß noch eine Oberfläche zur Benutzersteuerung realisiert werden. Mittels der Benutzeroberfläche kommuniziert der

Anwender mit der Arbeitsumgebung, initiiert die einzelnen auszuführenden Arbeitsschritte und gibt über sie manuell Informationen ein. Beispielhaft wird hier die Anbindung an ein einfaches WWW-Frontend gezeigt.

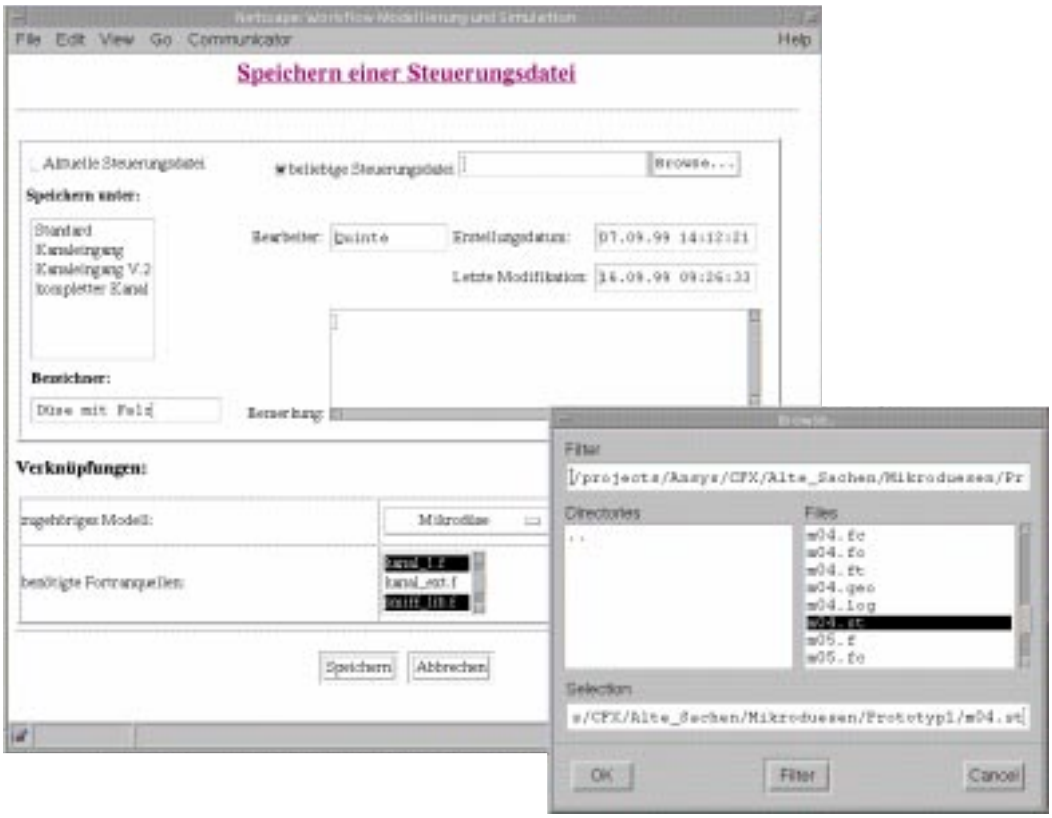


Abbildung 5.12: *Formular zum Speichern einer Steuerungsdatei im Container*

Abbildung 5.12 zeigt das Formular zum Speichern einer Steuerungsdatei im Steuerdaten-Container. Hierbei kann zwischen zwei Möglichkeiten unterschieden werden.

1. Dem Speichern der aktuellen, d. h. der zuvor geladenen Steuerungsdatei. In diesem Fall ist der Umgebung der Name der Steuerungsdatei bekannt und es müssen lediglich die zusätzlichen Informationen im Formular angegeben werden.

2. Das Speichern einer externen, bisher nicht im Container vorhandenen Steuerungsdatei (importieren einer Steuerungsdatei).

Die Abbildung zeigt Fall (2). Innerhalb des dargestellten Formulars ist in der oberen Hälfte eine Liste aller bereits vorhandenen Steuerungsdaten-Objekte und die relevanten administrativen Informationen hierzu zu sehen. In der unteren Hälfte wird die Beziehung zum Simulations-Modell-Objekt und die benötigten *Fortran* Module repräsentiert.

Durch den Einsatz eines Browsers als Benutzerfrontend kann dessen *File-Upload* Feature genutzt werden. Der Browser stellt hierbei von sich aus die Möglichkeit zur Verfügung, eine Datei auf dem lokalen Rechner zu selektieren und an ein CGI–Programm, das innerhalb eines WWW-Servers läuft, zu übertragen, von wo es dann in ein $\mathcal{W}*FLOW$–Container-Objekt eingetragen wird. Das CGI–Programm bildet bei der Realisierung einer WWW-basierten Oberfläche die

Schnittstelle zwischen Browser und $\mathcal{W}*FLOW$-Engine, ist es doch für die Repräsentation der Seiten im Browser, der Interaktion mit dem Benutzer und den Zugriff auf die $\mathcal{W}*FLOW$–Engine zuständig. Dieser Sachverhalt ist in Abbildung 5.13 nochmals dargestellt.

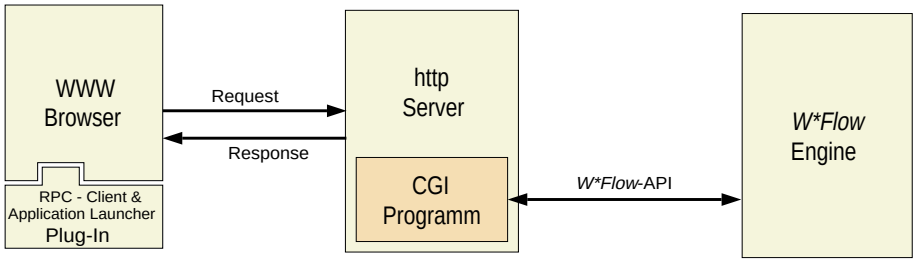


Abbildung 5.13: *Funktionsweise eines CGI–Programms als Gateway zwischen Anwender und Anwendung*

Abbildung 5.14 zeigt die Oberfläche für eine Instanz der Aktivität Steuerungsdatei bearbeiten. Im oberen Bereich sind die relevanten Informationen, wie Status der Instanz, Bearbeiter und die verbundenen Container, etc. dargestellt. Die möglichen Arbeitsschritte (untere Hälfte) entsprechen den Übergängen aus Abbildung 5.10. Ist ein Arbeitsschritt, aufgrund des aktuellen Zustandes der Instanz nicht ausführbar, so ist dieser inaktiv, wie beispielsweise in der Abbildung die Aktion "Speichern".

Abbildung 5.14: *WWW-Benutzerfrontend für die Aktivität* Steuerungsdatei bearbeiten

Die Realisierung der Oberfläche stellt, analog zur Erstellung der Oberfläche für den Container aus Abschnitt 5.2.1, kein Problem dar, stellt die $\mathcal{W}*FLOW$–API (Anhang B) doch alle notwendigen Methoden für den Zugriff auf die Aktivitäteninstanz zur Verfügung.

Die Möglichkeit, WWW-Browser um Zusatzfunktionalität zu erweitern kann weiterhin dazu genutzt werden, den *Toolstarter* in den Browser zu integrieren. WWW–Browser kennen von sich aus eine Reihe von MIME–Typen. Diese sind im allgemeinen[3]:

- text/html,
- text/plain,
- image/gif.

Moderne Browser sind darüber hinaus in der Lage, noch eine ganze Reihe weiterer MIME–Typen zu verstehen. Versteht ein Browser einen bestimmten MIME–Typ nicht, so ist der Benutzer in der Lage, die Fähigkeit seines Browsers durch Angabe von sogenannten *Helper Applications* zu erweitern. Diese werden im *MIME Configuration File* abgelegt und ordnen einem bestimmten MIME–Typ eine Applikation zu.

So kann z. B. der Netscape Browser durch folgenden Eintrag in seine *mailcap*–Datei dahingehend erweitert werden, daß er PostScript versteht:

```
application/postscript; start gnu ghostview %s
```

Empfängt der Browser nun ein Dokument, das vom MIME-Typ application/postscript ist, so wird das ghostview Programm aufgerufen und das entsprechende Dokument angezeigt.

Neben der Definition von *Helper Applications* ist die Erweiterung der Fähigkeiten eines Browsers durch sogenannte *Plug-Ins*[4] möglich. Bei *Plug-Ins* handelt es sich um dynamische Bibliotheken, die eine spezielle API des Browsers nutzen und weiterhin dem Browser eine Anzahl von Funktionen zur Verfügung stellen. Im Gegensatz zu den *Helper Applications* ist hierbei eine engere Kopplung zwischen Browser und Zusatzapplikation möglich, da auf die Funktionalität des Browsers zugegriffen und das Verhalten des Browsers vom *Plug-In* aus beeinflußt werden kann.

Zur Einbindung des *Toolstarters* wird der Browser in die Lage versetzt, Daten vom Typ application/x-praxis-rpc (Abschnitt 4.5.3) an ein geeignetes *Toolstarter-Plugin* zu senden. Dazu muß in einem ersten Schritt das *Plug-In* geschrieben und dem Browser bekannt gemacht werden. Empfängt der Browser anschließend eine Datei vom Typ application/x-praxis-rpc, so überläßt er die Abarbeitung der Datei dem *Plug-In*, das wiederum nur den eigentlichen *Toolstarter* aktivieren muß, der dann die Interpretation der Datei durchführt und das entprechende Programm startet. Die Nutzung des Browsers zum Starten von Anwendungen wird in [DKY96], ebenfalls auf der Verwendung eines eigenen MIME-Typs und einer *Helper Application* beschrieben. Im Unterschied zum hier realisierten *Toolstarter* wird aber kein Transport von benötigten Daten unterstützt.

5.2.2.6 Zusammenspiel der Aktivitäten

Zum Abschluß der Vorstellung soll noch ein einfaches Beispiel für die Synchronisationsmöglichkeiten zwischen einzelnen Aktivitäten vorgestellt werden. So ist es beispielsweise nicht sinnvoll eine Simulation zu starten, wenn die von *CFX* benötigten Dateien noch nicht geladen sind. Zumindest müssen zuvor eine Steuerungsdatei und eine Modelldatei ins Arbeitsverzeichnis geladen werden.

[3] Es gibt jedoch auch noch rein textbasierte Browser wie lynx, die keine Grafik darstellen und für den Einsatz mit rein textorientierten Terminals gedacht sind.

[4] Plug-Ins werden momentan von Netscape und Microsoft Browsern unterstützt.

Am einfachsten kann das im Rahmen einer Vorbedingung für einen Zustandsübergang überprüft werden. So erlaubt es die $\mathcal{W}$*FLOW*–API, jeden Übergang mit einer Vor– und Nachbedingung (siehe Definition 3.13 und Anhang B.5) zu versehen. Beispiel 5.2.3 zeigt dies exemplarisch, indem die Zustände der beiden Aktivitäteninstanzen *Steuerungsdatei bearbeiten* und *Modelldatei bearbeiten* im Rahmen einer Vorbedingung (2) für den Übergang in den Zustand Laufend (1) überprüft werden.

```
...
my $ch = $act->define_handler('Initialisiert'=>'Laufend',           # (1)
                              NAME=>'Start_SIM,
                              PACKAGE=>$package,
                              METHOD=>$method);

$ch->setprecondition(q(
     my $mod  = $self->parent->get_instance('MODELL_BEARBEITUNG');   # (2)

     my $ctrl = $self->parent->get_instance('STEUER_BEARBEITUNG');
     my $m_state    = $mod->state;
     my $c_state = $ctrl->state;
     return (($m_state eq "geladen" || $m_state eq "modifiziert") &&
             ($c_state eq "geladen" || $c_state eq "modifiziert"));
));
...
```

Beispiel 5.2.3: Precondition für den Start eines Simulationslaufes

5.3 Weitergehende Einsatzmöglichkeiten und Erweiterungen

Nachdem nun der Einsatz der $\mathcal{W}$*FLOW*–Komponenten innerhalb einer zu realisierenden Arbeitsumgebung an einigen, wenigen Arbeitsschritten des Beispiel-Szenarios gezeigt wurde, sollen in diesem Abschnitt noch weitergehende Einsatzmöglichkeiten von $\mathcal{W}$*FLOW* Techniken im Rahmen des Beispielszenarios angedeutet werden.

So kann ein automatischer Vergleich zwischen den am realen Objekt gemessenen Durchflußzeiten und den simulierten Zeiten durchgeführt werden. Dies kann mit einem kurzen, mittels der $\mathcal{W}$*FLOW*–API realisierten *Perl*–Programms realisiert werden, das zum einen die zentrale Ergebnisdatei (DUMP) analysiert und dort die simulierten Durchlaufzeiten extrahiert und weiterhin aus dem Container mit den realen Messergebnissen die Durchlaufzeiten ermittelt und vergleicht. Liegt die Differenz innerhalb eines definierten Grenzwertes, so kann der Benutzer darüber informiert werden, ohne daß er die erzielten Ergebnisse persönlich in Augenschein nehmen muß.

Alternativ kann mit obigem Programm eine Iteration über eine Reihe von Simulationsläufen mit unterschiedlichen Parametern durchgeführt werden. Auf diese Weise lassen sich alle Simulationsläufe bestimmen, bei denen die Durchlaufzeiten innerhalb einer vorgegebenen Schranke liegen.

Weitere Möglichkeiten bestehen darin, Groupeware Funktionalitäten zu integrieren. So kann eine automatische Notifikation der Modellierer, z. B. mittels e-mail, erfolgen, wenn neue reale Messergebnisse oder Mikrostrukturen vorliegen.

5.4 Überblick über die realisierten Softwaremodule

Nachdem im vorliegenden Kapitel der Einsatz der Komponenten an einem realen Szenario gezeigt wurde, soll abschließend ein Überblick über die im Rahmen der Arbeit erstellten Softwaremodule gegeben werden.

5.4.1 Komponenten-Programmiersprache

Wie in Abschnitt 2.1.4.2 erläutert, können die Komponenten in einer beliebigen Programmiersprache realisiert werden. Einzige Bedingung ist, daß eine Schnittstelle zu der Sprache existiert, die als Baukasten-Programmiersprache eingesetzt wird. In der vorliegenden Arbeit wurde, wie in Abschnitt 2.2.2.1 ausführlich dargelegt, die Skriptsprache *Perl* als Baukastenprogrammiersprache ausgewählt. Die Realisierung der einzelnen Komponenten erfolgte ebenfalls in dieser Sprache. Die Gründe für den Einsatz von *Perl* als Komponenten-Programmiersprache waren die folgenden:

- *Perl* eignet sich hervorragend zur Erstellung von Prototypen und es lassen sich damit sehr schnell komplexe Anwendungen entwickeln.
- Dadurch, daß die Baukasten-Programmiersprache und die Sprache, in der die Komponenten entwickelt worden sind, identisch sind, entfällt der Aufwand einer Abbildung von der einen auf die andere Sprache.
- Die Nachteile einer Skriptsprache liegen hauptsächlich in der geringeren Ablaufgeschwindigkeit im Vergleich zu einer Systemprogrammiersprache (Faktor 10–20 [Ous98]). In einer prototypischen Realisierung, in der es eher um einen "*proof of concept*" geht, ist der Aspekt der Geschwindigkeit kein so vorrangiges Kriterium wie im produktiven Einsatz. Durch die Ausrichtung der Architektur von $\mathcal{W}*FLOW$ auf eine möglichst hohe Skalierbarkeit können die zu realisierenden Systeme aber auch bei hoher Last so konfiguriert werden, daß diese entsprechend verteilt werden kann.

In einer Realisierung für ein großes Produktions-System werden aber bevorzugt Komponenten eingesetzt, die in einer Systemprogrammiersprache, wie beispielsweise C++, realisiert sind. In diesem Falle verschiebt sich die XS–Schnittstelle, welche die Abbildung zwischen C++ und *Perl* vornimmt, innerhalb der Komponente weiter nach oben, wie in Abbildung 5.15 beim Vergleich beider Ansätze zu sehen ist. Das *Perl ObjStore*–Modul, das im prototypischen System (linke Seite) die Abbildung der speziellen *ObjectStore*–C++–Datentypen auf die *Perl* Datentypen vorgenommen hat, kann in diesem Fall entfallen, da die Realisierung der Komponentenfunktionalität direkt in C++ (rechte Seite, mittlere Schicht) erfolgt und somit bei der Realisierung der Komponenten direkt die von *ObjectStore* zur Verfügung gestellten C++–Datentypen genutzt werden können. Es muß lediglich noch eine Abbildung der in C++ vorliegenden Komponenten-API auf *Perl* erfolgen (rechte Seite, oberste Schicht).

5.4.2 Unterstützte Plattformen

Durch die Verfügbarkeit von *Perl* für eine Vielzahl von verschiedenen Rechnerarchitekturen und Betriebssystemen ist der entwickelte Baukasten prinzipiell auf einer großen Anzahl von Plattformen einsetzbar. Eine Einschränkung bildet jedoch die zugrundeliegende *ObjectStore*–Datenbank, die aber auch auf einer Reihe von UNIX Betriebssystemen (*Solaris*, *HP-UX*, *IRIX*, *AIX*, *Digital UNIX*) und unter *Microsoft Windows* (95/98/NT) eingesetzt werden kann. Im

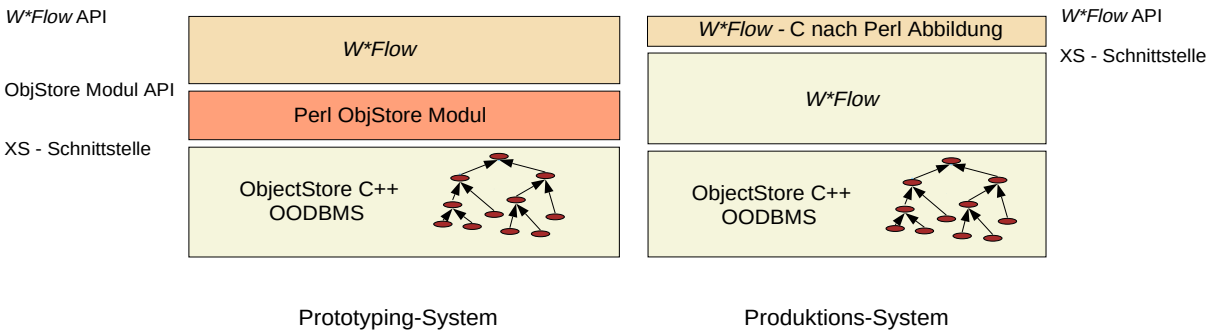


Abbildung 5.15: *Vergleich der Realisierung eines Prototyping-Systems mit einem Produktions-Systems*

Rahmen der vorliegenden Arbeit wurde das System unter *Solaris 2.5/2.6* und *Windows NT* eingesetzt. Durch die konsequente Schichtenarchitektur der Komponenten (Abbildung 5.15) ist es aber auch möglich eine andere Datenbank einzusetzen und damit auch andere, momentan nicht unterstützte Plattformen bedienen zu können.

5.4.3 Module

Nach der Begründung für die Wahl der Sprache zur Realisierung der Komponenten und einem Überblick über die unterstützten Plattformen, werden im folgenden die realisierten Module kurz vorgestellt. Die Realisierung ist in einzelne Module unterteilt, die jeweils eine Reihe von Klassen[5] im objektorientierten Sinne enthalten. Die Vorstellung der Module umfaßt die Beschreibung der Funktionalität der wichtigsten Klassen. Ergänzend ist in Anhang B eine Zusammenfassung der programmatischen API abgedruckt.

5.4.3.1 Modul Object

In diesem Modul ist die gemeinsame Oberklasse (`WildFlow::Object`) aller $\mathcal{W}*FLOW$ Komponenten realisiert, wie sie in Abschnitt 2.3 beschrieben wurde. Sie enthält Mechanismen und Methoden zur Verwaltung der Attribute, der Kommunikation und zur Realisierung der Persistenz. Alle im folgenden vorgestellten Klassen erben die Funktionalität, die von der Klasse `WildFlow::Object` bereitgestellt wird.

5.4.3.2 Modul Workflow

Die Klasse `WildFlow::Workflow` dient als Behälter für die Aufnahme von Aktivitäten. Sie stellt die Basisfunktionalität für den Aufbau komplexerer Arbeitsabläufe bereit und ist somit verantwortlich für die funktionsbezogenen Aspekte eines Workflows. Die Klasse ist nur rudimentär ausgeprägt, da eine weitergehende Funktionalität zum Verlust der Verfahrensunabhängigkeit führen würde.

5.4.3.3 Modul Activity

Das Aktivitätenmodul umfaßt die Hauptklassen `WildFlow::Activity`, `WildFlow::ActivityHandler`, `WildFlow::Task` und `WildFlow::ActivityInstance`. Die Funktionalität, die von diesen Klassen realisiert wird, deckt sich mit der in Abschnitt 3.2 vorgestellten Aktivitätenkomponente. Die Klasse `WildFlow::Activity` stellt Methoden zum Aufbau bestimmter, anwendungsspezifischer Aktivitäten bereit, von der dann anschließend Instanzen vom Typ

[5] In *Perl* werden Klassen als *Packages* bezeichnet.

`WildFlow::ActivityInstance` erzeugt werden, die das spezielle, zuvor mittels der Klasse `WildFlow::Activity` definierte, Verhalten aufweisen. Weiterhin gibt es noch die Klasse `WildFlow::Task`, die das Bindeglied zwischen einer Aktivitäteninstanz und den in Kapitel 4 vorgestellten *Toolservices* darstellt. Eine Instanz der Klasse `WildFlow::Task` entspricht genau einer Aktion, die im Rahmen eines Zustandsübergangs ausgeführt wird. Im Falle einer *Whitebox*–Integration einer Anwendung erlaubt die `WildFlow::Task`–Instanz den Zugriff auf die korrrespondierende Instanz vom Typ `WildFlow::ActivityInstance` und auf alle relevanten in der $\mathcal{W}*FLOW$–Engine vorhandenen Informationen. Ein Codefragment für den Einsatz des Task-Objektes ist in Beispiel 4.4.2 zu sehen.

5.4.3.4 Modul Container

Das Modul Container umfaßt alle Klassen, die mit der Verwaltung der Anwendungsdaten in den Containern zu tun haben. Im einzelnen handelt es sich dabei um die Klassen `WildFlow::Container`, `WildFlow::ContainerRepository`, `WildFlow::ContainerObject` und `WildFlow::Component`, wobei letztere Klasse nochmals in Unterklassen für die Behandlung von mengenwertigen und versionierten Komponenten unterteilt ist. Die Funktionalität der Klassen entspricht den gleichnamigen, in Abschnitt 3.1.3 vorgestellten, Teilkomponenten eines Containers.

5.4.3.5 Modul Toolservices

Das Modul Toolservices enthält die beiden Hauptklassen `WildFlow::Toolserver` und `WildFlow::ToolStarter`. Die `WildFlow::Toolstarter` Klasse implementiert die in Abschnitt 4.5.3 beschriebene Funktionalität der *Toolstarter* Komponente, beispielsweise die Kommunikation mit dem *Toolserver*, den Datentransport und den Start der Anwendung. Die Klasse `WildFlow::Toolserver` stellt Methoden zur Administration der Einträge im Server und zur Abfrage zur Laufzeit bereit.

5.4.4 Einsatz der Komponenten

Die entwickelten Komponenten werden bereits zum Teil in institutsinternen Anwendungen, bzw. innerhalb des Verbundprojektes DEMIS eingesetzt. So werden die *Toolservices* und *Aktivitäten* im Rahmen einer verteilten Optimierungsumgebung [UNJ+99] für den Aufruf von Simulationswerkzeugen genutzt (DEMIS). In einer weiteren Anwendung werden die $\mathcal{W}*FLOW$–Container zur Ablage und genaueren Beschreibung von *ANSYS* Ergebnissen eingesetzt [Sti99]. Neben den zugehörigen Ein– und Ausgabedateien werden zusätzliche Informationen über Material, Parameter, Vernetzung, Constraints, Geometrie, Last, etc. mit abgelegt. Zusätzlich geht auch noch eine Interpretation des Ergebnisses durch einem Experten mit in den Ergebnisdatensatz ein. Weiterhin werden die $\mathcal{W}*FLOW$–Container zur Aufnahme von Ergebnisdokumenten im Rahmen des DEMIS Projektes [Dü98] eingesetzt. Für die beiden letztgenannten Anwendungen, wurde jeweils eine webbasierte Oberfläche für die Container entwickelt.

5.4.5 Weitere Softwarekomponenten

Neben den Modulen zur Realisierung der Komponentenfunktionalität wurden im Rahmen der Arbeit noch eine Reihe weiterer Programme realisiert. So entstand beispielsweise eine komplette WWW-basierte Administrationsoberfläche für den *Toolserver*, so wie sie in Abschnitt 4.5.2 des Toolserver-Kapitels vorgestellt wurde.

Daneben wurde noch ein *Interceptor* Modul zur engen Ankopplung, Adaption und Automatisierung kommandozeilenorientierter Anwendungen an WEB-basierte Oberflächen, basierend auf

einen Standard Telnet Applet [JM99], entwickelt. Das Modul erlaubt es auf einfache Art und Weise, kommandozeilenorientierte Anwendungen eine neue, plattformunabhängige[6] verteilte Benutzerschnittstelle zur Verfügung zu stellen, neue zusätzliche Kommandos (Makros) zu erstellen und bestimmte Schritte zu automatisieren (z. B. das Laden von bestimmten Eingabedateien, etc.).

[6]Das *Interceptor*–Modul läuft momentan nur auf Rechnern mit *UNIX–Betriebssystem*. Die neue Benutzerschnittstelle der ursprünglichen Anwendung ist aber auf jedem Rechner einsetzbar für die ein *Java*–fähiger WWW-Browser verfügbar ist.

Kapitel 6

Zusammenfassung

Im Rahmen der vorliegenden Arbeit wurde ein Konzept für einen Workflow-Baukasten entwickelt und realisiert, der den Aufbau von verteilten Workflowsystemen und Entwicklungsumgebungen mit dem Schwerpunkt wissenschaftlich–technischer Anwendungen unterstützt. Besonderes Gewicht wurde bei diesem Konzept sowohl auf die zusätzlichen Anforderungen des primären Einsatzgebietes in bezug auf Einsatzumgebung, Applikationsintegration, Datenverwaltung und Flexibilität in der Abarbeitung gelegt als auch auf die Vermeidung bekannter Schwachpunkte existierender Workflowsysteme in den Bereichen Funktionalität und Architektur. Das Ergebnis ist ein komponentenorientierter Baukasten, der eine Reihe von verfahrensunabhängigen Basiskomponenten zum Aufbau von verteilten Anwendungen im Workflowbereich enthält und als $\mathcal{W}*FLOW$ bezeichnet wird. Der Baukasten besteht aus:

- *Containern*
- *Aktivitäten*
- *Toolservices*

In Kapitel 2 der Dissertation wurde die Machbarkeit der Zielsetzung nachgewiesen. Die Konzeption des Baukastensystems $\mathcal{W}*FLOW$ umfaßte die Integration verschiedener Technologien in Form eigenständiger Komponenten. Bei den integrierten Technologien handelt es sich um ein Workflow-Konzept, dessen Aufgabe die Koordination der Arbeitsschritte ist und das in Gestalt der *Aktivitäten*–Komponente in das System integriert wurde. Die Datenmanagent Technologie wurde in Form der *Container*–Komponente integriert und das Applikations-Framework wurde durch die *Toolservices* realisiert. Die drei realisierten Komponenten können in Abbildung 1.5 der Einleitung, zusammen mit den externen Anwendungen, ausgemacht werden. Oben die existierenden, einzubindenden Anwendungen, dann die Schnittstelle zur Anbindung an den Baukasten (*Toolservices*), d. h. an die Ablaufsteuerung (*Aktivitäten*) und unten die Datenorganisation (*Container*).
Ein wichtiger Punkt betraf die Auswahl geeigneter Schnittstellen für das Systems. Es wurde ausführlich gezeigt, daß der Einsatz einer Skriptsprache sowohl als "Klebstoff" zwischen den Komponenten, d. h. als Baukastensprache, als auch als Laufzeitschnittstelle des Systems, Vorteile gegenüber einer Systemprogrammiersprache, wie sie von den meisten kommerziellen Workflowsystemen eingesetzt werden, bietet. Die anschließend vorgestellte Architektur berücksichtigt insbesondere Verteilungsgesichtspunkte mit dem Ziel einer hohen Skalierbarkeit.

In Kapitel 3 wurden die zwei zentralen Basiskomponenten *Container* und *Aktivität* entwickelt. Eine der primären Anforderungen an ein Workflowsystem im wissenschaftlich–technischen Umfeld ist der Umgang mit großen, komplexen und miteinander in Beziehung stehenden Datenmengen. Zur Bewältigung dieser Anforderung wird bei $\mathcal{W}*FLOW$ eine eigene Datenhaltungskomponente (*Container*) eingesetzt, die in Abschnitt 3.1, unter Berücksichtigung der Anforderungen und Probleme aus dem Bereich *Scientific Database*, konzipiert wurde.
Abschnitt 3.2 beginnt mit einer einleitenden Untersuchung des *Aktivitäten*–Begriffs. Die Untersuchung zeigte, daß in der Literatur keine einheitliche Definition des Begriffs existiert. Eine sich anschließende strukturelle Betrachtung von wissenschaftlich–technischen Workflows ergab eine starke Vernetzung zwischen auf den gleichen Ressourcen operierenden Arbeitsschritten. Davon ausgehend wurde eine objektorientierte Modellierung von vernetzten Arbeitsschritten in Form eines Aktivitätenobjekts eingeführt. Hierbei wurden die durchzuführenden Arbeitsschritte als Methoden des Objekts betrachtet. Die im Anschluß daran diskutierten Anforderungen an Transaktionsmodelle im Workflowbereich machten deutlich, daß die bekannten, aus dem klassischen Datenbank–Transaktionsmodell stammenden Mechanismen versagen und neue, flexible, auf der Semantik der Anwendung basierende Transaktionsmodelle entwickelt werden müssen.
Eine der zentralen Anforderungen an $\mathcal{W}*FLOW$ beim Einsatz im wissenschaftlich–technischen Umfeld ist die notwendige Flexibilität bei der Durchführung von Workflows. Aus diesem Grund kommt der Realisierung der Ablaufsteuerung besondere Bedeutung zu. Die von $\mathcal{W}*FLOW$ zur Verfügung gestellten Elemente erlauben die Formulierung der verhaltensbezogenen Aspekte auf Basis von Bedingungen, die auf den Daten, Metadaten, Zuständen und externen Quellen definiert werden können. Weiterhin wurde ein asynchroner Kommunikationsmechanismus entwickelt, der zur Modellierung unterschiedlicher Beziehungstypen und damit verschiedener aktiver und passiver Kontrollmechanismen eingesetzt werden kann.

Kapitel 4 befaßte sich mit der Anbindung und Integration externer Programme an $\mathcal{W}*FLOW$. Eine Charakteristik von Workflows im wissenschaftlich–technischen Umfeld ist die Heterogenität sowohl im Hard– als auch im Softwarebereich. Dies führt zu großen Problemen bei der Organisation der einzusetzenden Programme, z. B. bei der Verknüpfung der im Workflow definierten Arbeitsschritte mit den eigentlichen Programmen. Im Rahmen der Vorstellung des *Toolservices* werden zu Beginn eine Reihe von Mängeln existierender kommerzieller Ansätze aufgezeigt und Lösungen hierzu erarbeitet. Die Notwendigkeit verschiedene Arten der Anbindung (*Blackbox*–, *Greybox*– und *Whitebox*–Integration) auf lokalen und entfernten Rechnern zu unterstützen, mit heterogenen Hard– und Softwareplattformen umgehen zu müssen und flexibel auf Anwenderpräferenzen einzugehen, führte zur Entwicklung der beiden Komponenten *Toolserver* und *Toolstarter*, die einen vollautomatischen, benutzerdefinierbaren Abbildungsmechanismus zwischen einer abstrakten, rechner– und plattformunabhängigen Beschreibung einer Aktion und einem konkreten Programm realisieren.

Kapitel 5 zeigte an einem durchgängigen, realen Szenario, wie die Komponenten zum Aufbau eines verteilten Entwicklungssystems eingesetzt werden können. Es wurde darin eine Methodik zum Aufbau von Entwicklungsumgebungen mittels $\mathcal{W}*FLOW$ vorgestellt und beispielhaft wichtige Aspekte der Arbeit mit dem Baukasten erläutert.

Die wesentlichen Ergebnisse der Arbeit sind:

1. Entwicklung eines Konzeptes zur Integration der Technologien Workflowmanagement, Datenmanagement und Frameworkfunktionalität innerhalb eines Baukastens zum Aufbau von Workflow–Management–Systemen und Entwicklungsumgebungen.

2. Konzeption und Realisierung der *Container*–, *Aktivitäten*– und *Toolservices*– Komponen-

ten unter dem Aspekt des Aufbaus von Systemen mit vollständig verteilter Architektur.

3. Entwicklung einer einheitlichen, netzwerktransparenten Skriptsprachen–API zum Aufbau der zu realisierenden Systeme und zur Integration und Anbindung von Anwendungen zur Laufzeit. Dadurch wird sowohl eine sehr enge Anbindung externer Anwendungen ermöglicht, als auch die Möglichkeit einer dynamischen Erweiterung der Funktionalität des Systems zur Laufzeit gegeben. Die Offenheit des Systems gegenüber Erweiterungen läßt die Grenzen zwischen Entwicklung und Konfiguration eines solchen Systems gewollt verschwinden.

4. Aufbau und Erprobung der *Container*-Komponente als vollständige Datenbankkomponente zur flexiblen Verwaltung von komplexen, verteilten und heterogenen Anwendungsdaten auf semantisch[1] hohem Niveau im Hinblick auf den datenintensiven Einsatz im wissenschaftlich–technischen Bereich und die Bereitstellung einer einheitlichen Schnittstelle, im Sinne einer neuen Middleware-Komponente, für den Zugriff auf diese Daten.

5. Aufbau und Erprobung der *Aktivitäten*–Komponente als zentrales Objekt der Ablaufsteuerung. Eine objektorientierte Interpretation der Aktivitäten innerhalb eines Workflows erlaubt die einheitliche Betrachtung von workflow–internen Aktionen und extern anzubindenden Programmen als Methoden einer Aktivität. Damit ist die Möglichkeit einer engen Integration externer Anwendungen in den Workflow–Kontext gegeben.

6. Integration von Basismechanismen zur Transaktionsverwaltung in das Aktivitätenkonzept, basierend auf den objektorientierten Konzepten *Kapselung* und *private Daten*, die für die Realisierung von semantischen, anwendungsspezifischen Transaktionsmodellen genutzt werden können.

7. Integration einer flexiblen Ablaufsteuerung in das Aktivitätenkonzept, das ein breites Spektrum von unterschiedlichen Arten der Unterstützung (aktiv, passiv) des Benutzers erlaubt und somit den besonderen Anforderungen nach hoher Flexibilität bei der Abarbeitung von Arbeitsabläufen im wissenschaftlich–technischen Bereich nachkommt.

8. Entwicklung und Realisierung der *Toolservices* auf Basis eines benutzerdefinierbaren, vollautomatisierten Abbildungsmechanismus, der eine dynamische Abbildung auf die konkrete Anwendung, basierend auf sogenannten *Laufzeitinformationen*, vornimmt. Somit wird eine flexible und einfach zu konfigurierende Administration der anzubindenden Programme und Skripte ermöglicht.

9. Erprobung des Gesamtkonzeptes anhand eines realen Anwendungsbeispiels, das die Tragfähigkeit des entwickelten Ansatzes zeigt, sowie der Einsatz von einzelnen Komponenten im Rahmen des DEMIS Verbundprojektes und innerhab von institutsinternen Anwendungen.

Zum Abschluß seien mögliche sich anschließende Forschungsaktivitäten sowie weitere Anwendungsfelder von $\mathcal{W}*FLOW$ betrachtet.

Mögliche weiterführende Arbeiten sind:

Die Integration einer allgemeingültigen Komponente zur Organisationsverwaltung. Heutige Systeme weisen in diesem Bereich meist eigene, rudimentäre und unflexible Komponenten auf, die

[1] d. h. um zusätzliche, im Hinblick auf die Anwendung relevante Informationen erweitert.

oft nur schwer an die Bedürfnisse eines Unternehmens angepaßt werden können. Typischerweise stellen die Systeme ein "Metamodell" zur Verfügung, das zur Beschreibung der Organisationsstruktur eines Unternehmens dienen soll. Derartige Modelle enthalten im allgemeinen Objekte wie *Stelle*, *organisatorische Einheit (OE)*, *Akteur*, *Stellvertreter*, etc. Daneben werden aber auch bereits die Attribute der Objekte und die Beziehungen zwischen ihnen festgelegt, was die allgemeine Verwendbarkeit stark einschränkt. Einen Ansatz zur Lösung dieses Problems wird in [Bußg7] vorgeschlagen. Es wird darin ein sehr generischer Ansatz zur Modellierung einer Organisationsverwaltung vorgestellt, indem auf eine Metaebene der oben beschriebenen Modelle gewechselt wird. Dabei werden neue Elemente wie *Organisationsobjekttyp* und *-beziehungstyp* eingeführt, die dann zur Modellierung einer Organisationsstruktur mit möglichen Ausprägungen *Stelle*, *Rolle* und *Organisationseinheit* eingesetzt werden können.

Ein weiterer Punkt, wie auch schon innerhalb der Arbeit angesprochen, ist die Integration von CORBA–Technologien. CORBA kann zum einen als Ersatz der momentan eingesetzten *Perl*–basierten Remote Method Invocation Realisierung genutzt werden. Die Integration ist weiterhin sinnvoll, da im Rahmen von CORBA ein Transaktionsmanager integriert ist, der verteilte Transaktionen unterstützt. Daneben bietet sich der CORBA–Ereignis–Service an, der als Basis des im Rahmen von $\mathcal{W}$*FLOW* momentan nur sehr rudimentär implementierten asynchronen Kommunikationsmechanismus eingesetzt werden kann und so eine zuverlässige, verteilte Kommunikation erlaubt. Weiterhin ist anzunehmen, daß CORBA in Zukunft eine wichtige Rolle spielen wird und eine Reihe von anzubindenden Applikationen eine CORBA–Schnittstelle aufweisen werden. Dies erscheint speziell unter dem Aspekt der Integration von CORBA, *Perl* und der $\mathcal{W}$*FLOW*–API sehr interessant.

Da es sich bei $\mathcal{W}$*FLOW* um einen Baukasten zum Aufbau von Workflowsystemen handelt, bietet es sich an, den Baukasten als Ausgangspunkt für weitere Forschungsaktivitäten im Bereich der Workflowsysteme oberhalb der $\mathcal{W}$*FLOW* Schnittstelle einzusetzen. Dieser Forschungszweig bietet zum momentanen Zeitpunkt noch eine Reihe von ungelösten Problemen oder zusätzlichen Anwendungsfeldern, wie etwa der bereits in Abschnitt 3.2.2 angesprochenen transaktionalen Unterstützung, der dynamischen Schema–Modifikation, Verteilungs– und Verfügbarkeitsaspekte, etc. Der Vorteil des Einsatzes von $\mathcal{W}$*FLOW* liegt vor allem darin, innerhalb kurzer Zeit ein maßgeschneidertes System, das als Entwicklungsrahmen für weitere prototypische und experimentelle Entwicklungen dienen kann, realisieren zu können.

Ein besonders interessantes Thema in dieser Hinsicht ist die Bereitstellung einer "Präsentationsschicht" oberhalb der $\mathcal{W}$*FLOW*–API. Diese kann unter zu Hilfenahme von Mechanismen wie *XML* [GP98] beispielsweise Containerdaten in einheitlicher Form Anwendungen zum Datenaustausch anbieten oder einen standardisierten Mechanismus zur Beschreibung von Workflowsemantik liefern und damit z. B. die Kooperation von verschiedenen Workflowsystemen erlauben. Weitere Möglichkeiten umfassen z. B. auch eine automatische Generierung von WWW–basierten Oberflächen.

Die Anwendung der $\mathcal{W}$*FLOW*–API in konkreten Projekten wird auch dem $\mathcal{W}$*FLOW*–Baukasten zu Gute kommen, da eine weitere Verifikation in Bezug auf Allgemeingültigkeit des Ansatzes sowie Verfahrensunabhängigkeit erfolgen kann. Oberhalb der $\mathcal{W}$*FLOW* Schnittstelle kann so eine Sammlung von alternativen Komponenten mit zusätzlichen Funktionalitäten entstehen, die dann wiederum den Aufwand zum Aufbau von speziellen, verteilten Entwicklungsumgebungen verringert. Diese Komponenten umfassen beispielsweise auch visuelle Objekte, die Teil einer Benutzerschnittstelle sein können. Neben den bereits von den $\mathcal{W}$*FLOW* Komponenten realisierten Komponentenmodellkonzepten *Properties*, *Ereignisbehandlung*, *Persistenz*, *Anpassung* und *Verteilungsaspekte* bieten sich die Komponenten weiterhin für den Einsatz im Rahmen

eines zu realisierenden Applikationsbuilders an, was die Realisierung der Konzepte *Introspection* und eventuell *Layout* erforderlich macht.

Ein interessanter Aspekt, besonders im Hinblick auf Plattformunabhängigkeit und die Entwicklung von Oberflächen, besteht in der Realisierung der $\mathcal{W}*FLOW$–API in *Java*. Durch die Entscheidung *Perl* als Schnittstellensprache einzusetzen, besteht mittels des *JPL*–Interfaces eine einfache Möglichkeit, die API nach *Java* zu portieren. Dies ermöglicht die Realisierung anspruchsvoller plattformunabhängiger, Web–basierter Oberflächen mittels den zwei möglicherweise populärsten Sprachen zur Realisierung von WWW–Anwendungen.

Der Einsatz von $\mathcal{W}*FLOW$ ist aber nicht ausschließlich auf den Aufbau von Workflowsystemen oder verteilten Entwicklungsumgebungen beschränkt. Es existieren daneben noch eine Reihe von weiteren möglichen Einsatzgebieten. Als Beispiel seien hier *e–commerce* Anwendungen genannt, deren Hauptaufgaben bzw. Anforderungen in der Kommunikation, der Adaptierbarkeit an die zugrundeliegende Geschäftslogik und Integration bestehender Anwendungen liegen – Aufgaben die komplett von $\mathcal{W}*FLOW$ abgedeckt werden können.

Literaturverzeichnis

[AAAM97] G. Alonso, D. Agrawal, A. E. Abbadi und C. Mohan: *Functionality and Limitations of Current Workflow Management Systems.* IEEE-Expert, 12(5), September 1997.

[AAEA+96] G. Alonso, D. Agrawal, A. El-Abbadi, M. Kamath, R. Günthör und C. Mohan: *Advanced Transaction Models in Workflow Contexts.* In: *Proceedings of the 12th International Conference on Data Engineering*, Seiten 574–583, Washington - Brussels - Tokyo, Februar 1996. IEEE Computer Society.

[AHST97a] G. Alonso, C. Hagen, H. J. Schek und M. Tresch: *Distributed Processing over Stand-alone Systems and Applications.* In: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, Seiten 575–579, 1997.

[AHST97b] G. Alonso, C. Hagen, H.-J. Schek und M. Tresch: *Towards a Platform for Distributed Application Development.* In: Dogac, A., L. Kalinichenko, T. Ozsu und A. Sheth (Herausgeber): *Proc. of the NATO Advance Studies Institute (ASI), 1995*, Istanbul, Turkey, August 1997. NATO Advance Studies Institute (ASI).

[AIL98] A. Ailamaki, Y. E. Ioannidis und M. Livny: *Scientific Workflow Management by Database Management.* In: *Proc. 10th International Conference on Scientific and Statistical Database Management*, Capri, Italy, Juli 1998.

[Alo96] G. Alonso: *The Role of Database Technology in Workflow Management.* Technischer Bericht Swiss Federal Institute of Technology, Zürich, Juni 1996.

[Alo97] G. Alonso: *Processes + Transactions = Distributed Applications.* In: *Proceedings of the High Performance Transaction Processing (HPTS) Workshop*, Asilomar, California, September 1997.

[Ama99] *Amazon.com Interview: Tom Christiansen*, Januar 1999. Online Artikel: `http://language.perl.com/ama-int/interview.html`.

[AMG+95] G. Alonso, C. Mohan, R. Günthör, D. Agrawal, A. El-Abbadi und M. Kamath: *Exotica/FMQM: A Persistent Message Based Architecture for Distributed Workflow Management.* In: *Proceedings of the IFIP Working Conference on Information Systems for Decentralized Organisations*, Trondheim, Norway, August 1995. IEEE Computer Society.

[ANS86] ANSI: *Database Language – SQL.* Technischer Bericht ANSI, Oktober 1986. Specifications for SQL-DDL and SQL-DML, at two levels. Annex on embedding in host languages.

[AS96] G. Alonso und H. J. Schek: *Database Technology in Workflow Environment.* Informatik, 1996.

[BDE+96] P. Buchberger, C. Düpmeier, H. Eggert, K.P. Scherer, P. Stiller und U. Stucky: *Softwarewerkzeuge zur Entwurfsunterstützung von LIGA–Strukturen.* In: W. John, H. Eggert und U. Hamm (Herausgeber): *Methoden und Werkzeugentwicklung für den Mikrosystementwurf. 4. Statusseminar zum BMBF Verbundprojekt METEOR*, Forschungszentrum Karlsruhe, Germany, November 1996.

[BDS+93] Y. Breitbart, A. Deacon, H.-J. Schek, A. Sheth und G. Weikum: *Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows.* SIGMOD Record (ACM Special Interest Group on Management of Data), 22(3):23–30, September 1993.

[BF93] N. Borenstein und N. Freed: *RFC 1521: MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, September 1993.

[BGH91] B. Bürg, H. Guth und A. Hellmann: *Parametric Optical Measurements of Micromechanical Structures with Arbitrary Plane Surface Geometries: The COSMOS-2D System.* In: Krahn, R. (Herausgeber): *Micro System Technologies. 2. Internationaler Fachkongress*, Seiten 280–287. VDE Verlag Berlin, Offenbach, 1991.

[BH96] A. P. Barros und A. H. M. Hofstede: *A Business Transaction Workflow Case Study: Road Closures in Queensland.* Technischer Bericht 393, Department of Computer Science, The University of Queensland, Brisbane, Australia, Dezember 1996.

[BHP96] A. P. Barros, A. H. M. Hofstede und H. A. Proper: *Towards Real-Scale Business Transaction Workflow Modelling.* Technischer Bericht 390, Department of Computer Science, The University of Queensland, Brisbane, Australia, November 1996.

[BLMM94] T. Berners-Lee, L. Masinter und M. McCahill: *RFC 1738: Uniform Resource Locators (URL)*, Dezember 1994.

[Blo92] J. Bloomer: *Power programming with RPC.* O'Reilly & Associates, Inc., Februar 1992.

[BMR94] D. Barbara, S. Mehrota und M. Rusinkiewicz: *INCAS: A Computation Model for Dynamic Workflows in Autonomous Distributed Environments.* Technischer Bericht Matsushita Information Technology Laboratory, 1994.

[BMVW95] C. Bauzer Medeiros, G. Vossen und M. Weske: *WASA: A Workflow-Based Architecture to Support Scientific Database Applications.* In: *Proceedings of the 6th International Conference on Database and Expert Systems Applications. Lecture Notes in Computer Science*, Band 978, Seiten 574–583. Springer Verlag, 1995.

[BMWCJ98] M. Böhm, K. Meyer-Wegener, C. Cap und S. Jablonski: *Ein konstruktiver Ansatz zur systematischen Entwicklung von Ausführungsanweisungen für Workflows.* Gemeinsamer Technischer Bericht, Lehrstuhl Datenbanken – Technische Universität Dresden, Institut für Informatik – Universität Zürich, Lehrstuhl für Datenbanksysteme – Universität Erlangen–Nürnberg, April 1998.

[BMWS96] M. Böhm, K. Meyer-Wegener und W. Schulze: *Formale Beschreibung und Transformation von Realisierungsalternativen für Geschäftsprozesse.* In: *Workshop "Workflow Management" des GI–Arbeitskreis Workflow Management; GI-Jahrestagung Informatik'96.* Klagenfurt, Österreich, September 1996.

[Bö97] M. Böhm: *Workflow-Management: Entwicklung und Anwendung von Systemen*, Kapitel 8 Modellierung von Workflows. dpunkt Verlag, Heidelberg, 1997.

[Bö98] M. Böhm: *Integration externer Applikationen im Workflow–Management.* Workflow Management Systems, 1998.

[Bre96] U. Breymann: *Die C++ Standard Template Library. Einführung, Anwendung, Konstruktion neuer Komponenten.* Addison-Wesley, 1996.

[BRS96] S. Blott, L. Relly und H. J. Schek: *An open abstract-object storage system.* In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Band 25, 2 der Reihe *ACM SIGMOD Record*, Seiten 330–340, New York, Juni 1996. ACM Press.

[BSR96] A. Bonner, A. Shrufi und S. Rozen: *LabFlow-1: A Database Benchmark for High-Throughput Workflow Management.* Lecture Notes in Computer Science, 1057:463–478, 1996.

[Buß97] C. Bussler: *Organisationsverwaltung in Workflow-Management-Systemen.* Doktorarbeit, Institut für Mathematische Maschinen und Datenverarbeitung (IMMD) Universität Erlangen–Nürnberg, Germany, 1997.

[Bul93] Bull: *Integration and Programming Guide FlowPATH, Release 01.01.00* , 1993. Document Number 44 A262XM Rev 1.

[Bus90] J. Van den Bussche: *A formal basis for extending SQL to object-oriented databases.* Bulletin of the European Association for Theoretical Computer Science, 40:207–216, Februar 1990.

[BV95a] S. Blott und A. Vckovski: *Accessing Geographical Metafiles through a Database Storage System.* Lecture Notes in Computer Science, 951:117ff., 1995.

[BV95b] S. Blott und A. Vckovski: *Extending a database storage system for geographical metadata.* In: *Proceedings of the Fourth International Symposium on Advances in Spatial Database Systems (SSD95)*, Lecture Notes in Computer Science, Portland, Maine, August 1995. Springer Verlag.

[BW96] A. Brayner und M. Weske: *Einsatz von FlowMark in der Molekularbiologie.* Fachbericht Angewandte Mathematik und Informatik 01/96-I, Lehrstuhl für Informatik, Universität Münster, 1996.

[Car96] T. E. Carone: *Middleware and Three-tier Client/Server Development.* Dr. Dobb's Journal of Software Tools, 21(11):16ff., November 1996.

[CCI88] CCITT: *Specification of abstract syntax notation one (ASN.1).* Technischer Bericht International Telegraph and Telephone Consultative Committee, 1988. Recommendation X.208.

[CCPP96] F. Casati, S. Ceri, B. Pernici und G. Pozzi: *Workflow Evolution.* In: *Proceedings of the 15th ER'96 international Conference*, Seiten 10–21. Springer Verlag, Oktober 1996.

[CFI92] CAD Framework Initiative, Inc.: *Tool Encapsulation Specification Version 1.0.0*, 1992. Panel Report.

[CFM98] F. Casati, M. G. Fugini und I. Mirbel: *An environment for designing exceptions in workflows.* Lecture Notes in Computer Science, 1413:139ff., 1998.

[CFX97] AEA Technology, Oxfordshire, United Kingdom: *CFX-4.2: Environment*, Dezember 1997.

[Chr95] G. Chroust: *Interpretable Process Models for Software Development and Workflow.* Lecture Notes in Computer Science, 913:144ff., 1995.

[Chr98] T. Christiansen: *MoxPerl WWW Database*, Juli 1998. `http://mox.perl.com/deckmaster/`.

[CL92] R. Colon et. al. (Tutor) und M. Litmaati (Reference): `vi` *Tutor and* `vi` *Reference*, 2.1 (Tutor), 8 (Reference) Auflage, 1992.

[Cow85] M. F. Cowlishaw: *The REXX language : a practical approach to programming.* Prentice Hall, 1985.

[CPA99] *Comprehensive Perl Archive Network*, 1999. Online Resource., z. B. `ftp://ftp.uni-hamburg.de/pub/soft/lang/perl/CPAN/CPAN.html`.

[CPA00] *search.cpan.org: Networking Devices IPC*, 2000. Online Resource., z. B. `http://search.cpan.org/Catalog/Networking_Devices_IPC/`.

[Cro82] D. Crocker: *RFC 822: Standard for the format of ARPA Internet text messages*, August 1982.

[CTW98] T. Christiansen, N. Torkington und L. Wall: *Perl Cookbook.* O'Reilly & Associates, Inc., August 1998.

[Dat99] Argonne National Laboratory, Argonne, IL: *Datorr: Desktop access to remote resources*, 1999. `http://www-fp.mcs.anl.gov/~gregor/datorr/`.

[DC99] *The Dublin Core: A Simple Content Description Model for Electronic Resources*, Februar 1999. `http://purl.org/dc/`.

[DE89] P. A. D. DeMaine und M. Edhala: *The QBE/SOLID Interface.* Technischer Bericht CSE89-08, Auburn University, September 18, 1989.

[Den97] A. Denning: *ActiveX Steuerelemente.* Microsoft Press, München, 1997.

[DG94] W. Deiters und V. Gruhn: *The FUNSOFT Net Approach to Software Process Management.* International Journal of Software Engineering and Knowledge Engineering, 4(2):229–256, 1994.

[DGS98a] C. Düpmeier, Ch. Griess und A. Schmidt: *PRAXIS – Benutzerdokumentation der Toolservices*, Juni 1998.

[DGS98b] C. Düpmeier, Ch. Griess und A. Schmidt: *Ein graphischer Editor zur Erstellung von Workflows für PRAXIS.* Technischer Bericht FZKA 5866, Forschungszentrum Karlsruhe, 1998.

[DGS98c] C. Düpmeier, Ch. Griess und A. Schmidt: *PRAXIS: Eine komponentenbasierte Umgebung zum Aufbau von Workflows in technisch/wissenschaftlichen Anwendungsfeldern.* Technischer Bericht Forschungszentrum Karlsruhe, Karlsruhe, 1998. Wissenschaftliche Berichte.

[DKY96] S. Dossick, G. Kaiser und J. J. Yan: *Distributed Tool Services Via the World Wide Web.* Technischer Bericht Department of Computer Science, Columbia University, New York, NY, Oktober 1996.

[Dow98] T. B. Downing: *Java RMI: Remote Method Invocation.* IDG Books, San Mateo, CA, USA, Januar 1998.

[Dü98] C. Düpmeier, 1998. Persönliche Mitteilung.

[EDG+96] H. Eggert, C. Düpmeier, H. Guth, K.P. Scherer und W. Süss: *Eine Informationsverarbeitungsumgebung zur Konstruktion und Vermessung von Mikrostrukturen.* In: *Informationstechnik für Mikrosysteme*, Seiten 10–21, Stuttgart–Büsnau, Januar 1996. VDE/VDI – Gesellschaft für Mikroelektronik (GME).

[EN96] C. A. Ellis und G. J. Nutt: *Workflow: The Process Spectrum.* In: *Proceedings NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions*, 1996.

[Eng93] H. Engesser (Herausgeber): *Duden Informatik.* Dudenverlag, Mannheim, zweite Auflage, 1993.

[Eng96] E. Eng: *The Qt Toolkit.* Linux Journal, November 1996.

[Eng97] R. Englander: *Developing Java Beans.* O'Reilly & Associates, Inc., 1997.

[Eur98] Euroforum: *Workflowbasiertes Prozeßmanagement*, München, Februar 1998. Reihe von Praxisberichten zur Einführung von Workflowsystemen aus verschiedenen Branchen.

[Fie95] R. Fielding: *RFC 1808: Relative Uniform Resource Locators*, Juni 1995.

[FJ98] E. Forster-Johnson: *The Holy Grail in Beta.* UNIX Review, März 1998.

[FJP90] J. French, A. Jones und J. Pfaltz: *Summary of the Final Report of the NSF Workshop on Scientific Database Management.* sigmod, 19(4):32–40, Dezember 1990.

[FM96] J. Feiler und A. Meadow: *Essential OpenDoc.* Addison-Wesley, 1996.

[Fri97] J. E. F. Friedl: *Mastering Regular Expressions – Powerful Techniques for Perl and Other Tools.* O'Reilly & Associates, Inc., Januar 1997.

[FSH+97] O. K. Ferstl, E. J. Sinz, Ch. Hammel, M. Schlitt und S. Wolf: *Bausteine für komponentenbasierte Anwendungssysteme.* HMD – Theorie und Praxis der Wirtschaftsinformatik, (197):24–46, September 1997.

[GD96] Ch. Griess und C. Düpmeier: *Ein graphischer Editor zur Spezifikation von Arbeitsabläufen.* Technischer Bericht Institut für Angewandte Informatik, Forschungszentrum Karlsruhe, Oktober 1996.

[GHJV96] E. Gamma, R. Helm, R. Johnson und J. Vlissides: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software.* Addison Wesley, 1996.

[GHS95] D. Georgakopoulos, M. Hornick und A. Sheth: *An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure.* Journal on Distributed and Parallel Databases, 3(2):119–153, April 1995.

[GJSO91] D. K. Gifford, P. Jouvelot, M. A. Sheldon und J. W. O'Toole: *Semantic file systems.* In: *Proceedings of 13th ACM Symposium on Operating Systems Principles*, Seiten 16–25. Association for Computing Machinery SIGOPS, Oktober 1991.

[GLO98] *Das Internetlexikon - Widmann marketing*, 1998. Online Glossar: `http://www.widmann-marketing.com/service/glossar/I\_lexikon.html`.

[GLO99a] *Kassandra(s) Projekt: Das Internet - Lexikon*, 1999. Online Glossar: `http://www.sigma-elektronik.de/kassandra/projekt/projekt.HTM`.

[GLO99b] *No Fronts: Wörterbuch*, 1999. Online Glossar: `http://www.nofronts.de/wbuch/wordbook.htm`.

[GLO99c] *RZ-Online (Internet): Computer-Lexikon*, 1999. Online Glossar: `http://rhein-zeitung.de/internet/lexikon/begriffe.html`.

[GMS87] H. Garcia-Molina und K. Salem: *SAGAS.* In: Dayal, U. und I. Traiger (Herausgeber): *Proceedings of the SIGMod 1987 Annual Conference*, Seiten 249–259, San Francisco, CA, Mai 1987. ACM Press.

[Gos84] J. Gosling: *Emacs user's manual.* Pyramid Technology Corporation, Mountain View, CA, USA, erste Auflage, 1984.

[GP98] C. F. Goldfarb und P. Prescod: *The XML Handbook.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1998.

[GRS94a] N. Goodman, S. Rozen und L. Stein: *Building a Laboratory Information System Around a C++- Based Object-Oriented DBMS.* In: Bocca, J., M. Jarke und C. Zaniolo (Herausgeber): *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, Seiten 722–729, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

[GRS94b] N. Goodman, S. Rozen und L. Stein: *Requirements for a deductive query language in the MapBase genome-mapping database.* In: Ramakrishnan, R. (Herausgeber): *Proceedings of the Workshop on Programming with Logic Databases In Conjunction with ILPS*, Seiten 18–32, Vancouver, B.C, 1994.

[GRS95] N. Goodman, S. Rozen und L. D. Stein: *Workflow Management Software for Genome–Laboratory Informatics.* Technischer Bericht Whitehead/MIT Center for Genome Research, Whitehead Institute for Biomedical Research, Cambridge MA, USA, August 1995.

[HA98a] C. Hagen und G. Alonso.: *Flexible Exception Handling in Process Support Systems.* Technischer Bericht 290, ETH Zürich, Institute of Information Systems, Februar 1998.

[HA98b] C. Hagen und G. Alonso.: *Flexible Exception Handling in the OPERA Process Support System.* In: *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS 98)*, Amsterdam, The Netherlands, May 1998. 18th International Conference on Distributed Computing Systems (ICDCS 98).

[Har88] D. Harel: *On Visual Formalisms.* Communications of the ACM, 31(5):514–530, Mai 1988.

[HG94] R. Herzig und M. Gogolla: *An SQL-like Query Calculus for Object-Oriented Databases.* In: Urban, S. und E. Bertino (Herausgeber): *Proceedings, Object-Oriented Methodologies and Systems*, LNCS 858, Seiten 20–39. Springer-Verlag, 1994.

[HH97] E. Hochrath und R. Hochrath: *Internet Wörterbuch Englisch - Deutsch.* Langenscheidt, München, September 1997.

[HL91] K. Hales und M. Lavery: *Workflow Management Software: the Business Opportunity.* Ovum Ltd., London, UK, 1991.

[Hol94a] D. Hollingsworth: *The workflow reference model.* Technischer Bericht TC00-1003, Workflow Management Coalition, Dezember 1994.

[Hol94b] D. Hollingsworth: *WfMC – The Workflow Reference Model.* Technischer Bericht WFMC-TC-1003 V 1.1, The Workflow Management Coalition, November 1994.

[Hor99] T. Horn: *Internet, Intranet, Extranet – Potentiale im Unternehmen.* R. Oldenbourg Verlag, 1999. Online Glossar: `ftp://ftp.oldenbourg.de/pub/daten/25129/Glossar0.htm`.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt und R. Sherman: *On the Formal Semantics of Statecharts.* In: *Symposium on Logic in Computer Science (LICS '87)*, Seiten 54–64, Washington, D.C., USA, Juni 1987. IEEE Computer Society Press.

[HS97] P. Heinl und H. Schuster: *Anwendung von Implementierungstechniken zur Integration von externen Applikationen*, Kapitel 16. dpunkt Verlag, Heidelberg, 1997.

[Hug96] D. J. Hughes: *Mini SQL – A lightweight Database Engine Release 1.0.11.* Hughes Technologies Ltd., Australia, Januar 1996.

[HW95] K. Hamann und W. Wirth: *Microsoft Windows 95 auf einen Blick.* Microsoft Press Deutschland, Unterschleissheim, 1995.

[IAB96] IABG Industrieanlagen-Betriebsgesellschaft mbH, Ottobrunn, Deutschland: *ProMInanD*, 1996.

[IBM96] IBM: *IBM FlowMark Modeling Workflow*, 1996. `http://publib.boulder.ibm.com:80/cgi-bin/bookmgr/BOOKS/EXMW2A00/CCONTEN%TS`.

[ILGP96] Y. Ioannidis, M. Livny, S. Gupta und N. Ponnekanti: *ZOO: A Desktop Experiment Management Environment*. In: *Proc. 22nd International VLDB Conference*, Seiten 274–285, Bombay, India, September 1996.

[Int87] International Organization for Standardization: *ISO 9000: 1987 – Quality management and quality assurance standards – Guidelines for selection and use*. Geneva, Switzerland, 1987.

[Ioa98] Y. Ioannidis: *Scientific Workflow Management*. In: *Second European Conference on Research and Advanced Technology for Digital Libraries*, Heraklion, Crete, Greece., September 1998.

[JB96] S. Jablonski und C. Bussler: *Workflow–Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, 1996.

[JBS97] S. Jablonski, M. Böhm und W. Schulze (Herausgeber): *Workflow-Management: Entwicklung und Anwendung von Systemen*. dpunkt Verlag, Heidelberg, erste Auflage, 1997.

[Jep97] B. Jepson: *Perl Utilities Guide*. O'Reilly & Associates, Inc., 1997. Part of the Perl Resource Kit.

[JET97] B. Joos, R. Endl und D. Tombros: *Evaluation von Workflow Management Systemen*. Technischer Bericht 3 SWORDIES, Universität Zürich – Institut für Informatik, Juni 1997.

[JGL99] W. John, W. Gross und R. Laur (Herausgeber): *7. GMM–Workshop. Methoden und Werkzeuge zum Entwurf von Mikrosystemen*, Paderborn, Januar 1999.

[JM99] M. L. Jugel und M. Meissner: *The Java Telnet Applet*, 1999. `http://www.first.gmd.de/persons/leo/java/Telnet/`.

[JMQ+96] W. Jakob, S. Meinzer, A. Quinte, W. Süss, M. Gorges-Schleuder und H. Eggert: *Partial Automated Design Optimzation based on adaptive search techniques*. In: *Proceedings of the 2nd International Conference Adaptive Computing in Engineering Design and Control'96*, University of Plymouth, März 1996.

[JMQC96] W. Jakob, S. Meinzer, A. Quinte und G. Clemens: *Simulator und genetischer Algorithmus verkürzen die Entwicklungsphase*. Electronic Industries, (5):56–69, 1996.

[KC93] K. Koymen und Q. Cai: *SQL*: a recursive SQL*. Information Systems, 18(2):121–128, März 1993.

[KL95] G. Koch und K. Loney: *Oracle: The Complete Reference*. Osborne/McGraw-Hill, Berkeley, CA, USA, dritte Auflage, 1995.

[KRB85] W. Kim, D. S. Reiner und D. S. Batory: *Query Processing in Database Systems.* Springer, Berlin, 1985.

[KSK93] N. Kamel, T. Song und M. N. Kamel: *An Approach for Building an Integrated Environment for Molecular Biology Databases and Software Tools.* Distributed and Parallel Databases, 1(3):303–327, Juli 1993.

[KSS96] S. Kleiman, D. Shah und B. Smaalders: *Programming with THREADS.* Prentice Hall, Mountain View, 1996.

[Kul94] K. G. Kulkarni: *Object-Oriented Extensions in SQL3: A Status Report.* SIGMOD Record (ACM Special Interest Group on Management of Data), 23(2):478, Juni 1994.

[Ley95] A. Leybusch: *Delphi.* Tewi Verlag, 1995.

[Lie98] W. Liebhart: *Fehler- und Ausnahmebehandlung im Workflow Management.* Doktorarbeit, Universität Klagenfurt, Fakultät für Wirtschaftswissenschaften und Informatik, Februar 1998.

[LL95] S. M. Lang und P. C. Lockemann: *Datenbankeinsatz.* Springer Verlag Berlin, 1995.

[LS87] P. C. Lockemann und J. W. Schmidt: *Datenbank-Handbuch.* Springer Verlag Berlin, Heidelberg, New York, 1987.

[LS98a] C. Laird und K. Soraiz: *1998: Breakthrough year for scripting.* SunWorld Online, August 1998. `http://www.sunworld.com/sunworldonline/swol-08-1998/swol-08-regex.html\%#1`.

[LS98b] C. Laird und K. Soraiz: *GUI toolkits: What are your options?* SunWorld Online, März 1998.

[LS98c] C. Laird und K. Soraiz: *Putting Perl together.* SunWorld Online, Mai 1998. `http://www.sunworld.com/swol-05-1998/swol-05-perl.html`.

[Lut96] M. Lutz: *Programming Python.* O'Reilly & Associates, Inc., 1996.

[MAA+95] C. Mohan, D. Agrawal, G. Alonso, A. El Abbadi, R. Guenthoer und M. Kamath: *Exotica: A Project on Advanced Transaction Management and Workflow Systems.* SIGOIS Bulletin (Special Issue on Business Process Management Systems: Concepts, Methods and Technology), 16:45–50, August 1995.

[MAGK95] C. Mohan, G. Alonso, R. Günthör und M. Kamath: *Exotica: A Research Perspective ob Workflow Management Systems.* Data Engineering Bulletin, 18(1):19–26, 1995.

[MC94] T. W. Malone und K. Crowston: *The Interdisciplinary Study of Coordination.* ACM Computing Surveys, 26(1):87–119, März 1994.

[Mei97] S. Meinhardt: *Componentware.* HMD – Theorie und Praxis der Wirtschaftsinformatik, (197):5–6, September 1997.

[Mei98] S. Meinzer: *Entwicklung von Verfahren zur Erstellung adaptierter Makromodelle für den Einsatz bei der Designoptimierung von Mikrosystemen.* Doktorarbeit, Fachbereich 1 Physik/Elektrotechnik, Universität Bremen, Februar 1998.

[Mey88] B. Meyer: *Objektorientierte Softwareentwicklung.* Prentice-Hall, 1988.

[Mic98] *Microsoft Visual Basic 6.0 Programmierhandbuch.* Microsoft Press, München, 1998.

[MMWFF92] R. Medina-Mora, T. Winograd, R. Flores und F. Flores: *The action workflow approach to workflow management technology.* In: *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, Emerging technologies for cooperative work, Seiten 281–288, Toronto, Ontario, 1992. ACM Press.

[Moh94] C. Mohan: *A Survey and Critique of Advanced Transaction Models.* SIGMOD Record (ACM Special Interest Group on Management of Data), 23(2):521ff., 1994.

[Moh97] C. Mohan: *Recent Trends in Workflow Management Products, Standards and Research.* In: Dogac, A., L. Kalinichenko, T. Ozsu und A. Sheth (Herausgeber): *Proc. of the Advanced Study Institute (ASI) on Workflow Management Systems and Interoperability*, Istanbul, Turkey, August 1997.

[MWBM96] M. Weske, G. Vossen und C. Bauzer Medeiros: *Scientific Workflow Management: WASA Architecture and Applications.* Technischer Bericht 03/96-I, Lehrstuhl für Informatik, Universität Münster, 1996.

[Nat95] R. B. Natan: *CORBA: A guide to a common request brooker architecture.* MacGraw–Hill, 1995.

[NL97] O. Nierstrasz und M. Lumpe: *Komponenten, Komponentenframeworks und Gluing*, Seiten 8–23. Nummer 197. dpunkt Verlag, September 1997.

[NSF96] *Report from the NSF Workshop on Workflow and Process Automation in Information Systems*, SIGMOD Record 25(4), 1996.

[Obe94] A. Oberweis: *Workflow Management in Software Engineering Projects.* Technischer Bericht AIFB – Universität Karlsruhe, Karlsruhe, 1994.

[Obj96a] Object Design Inc., Burlington, MA: *Reference Manual: Object Store Release 4.0 for UNIX Systems*, 1996.

[Obj96b] Object Design Inc., Burlington, MA: *User Guide: LI Object Store Release 4.0*, 1996.

[Oka98] J. Okamoto: *Perlguts - Perl's Internal Functions*, 1998. `ftp://ftp.gmd.de/packages/CPAN/doc/manual/html/perlguts.html`.

[OMG95a] *CORBA: Architecture and Specification.* Object Management Group, Inc., 1995.

[OMG95b] *CORBAfacilities: Common Facilities Architekture.* Technischer Bericht Revision 4.0, Object Management Group, Inc., 1995.

[OMG95c] *CORBAservices.* Technischer Bericht Object Management Group, Inc., März 1995.

[OMG96] *CORBAservices.* Technischer Bericht Object Management Group, Inc., Juli 1996.

[OMG97] *Workflow Management Facility – Request for Proposal.* Technischer Bericht cf/97-05-06, Object Management Group, Inc., Mai 1997.

[OMG98] *OMG BODTF RFP #2 Submission Workflow Management Facility.* Technischer Bericht bom/98-06-07, Object Management Group, Inc., Juli 1998.

[OS96] A. Oberweis und P. Sander: *Information System Behavior Specification by High Level Petri Nets.* ACM Transactions on Information Systems, 14(4):380–420, 1996.

[Ous94] J. K. Ousterhout: *Tcl and Tk toolkit.* Addison–Wesley, 1994.

[Ous98] J. K. Ousterhout: *Scripting: Higher-Level Programming for the 21st Century.* IEEE Computer, 31(3):23–30, 1998.

[OV96] H. Österle und P. Vogler (Herausgeber): *Praxis des Workflow Management.* Vieweg Verlag, Braunschweig/Wiesbaden, 1996.

[Owe93] J. Owen: *STEP: An Introduction.* Information Geometers, 1993.

[Pfa95] J. Pfaltz WWW Homepage, Juni 1995. `http://www.cs.virginia.edu/~jlp/`.

[Pri99] J. Pritikin: *ObjStore - Perl Extension For ObjectStore OODBMS*, Februar 1999.

[PS91] G. Piatetsky-Shapiro: *Knowledge discovery in databases.* IEEE Expert-Intelligent Systems & Their Applications, 6:74–76, Oktober 1991. Discussion of second AAAI workshop on KDD.

[PSH+99] R.-P. Peters, C. Schön, S. Halstenberg, A.Quinte und H. Eggert: *Entwicklung medizinischer Teststreifen.* In: John, W., W. Gross und R. Laur (Herausgeber): *MIMOSYS/DEMIS Statusseminar*, Paderborn, 1999.

[Qui98] A. Quinte: *Entwicklung von Verfahren für den Einsatz der Finite–Elemente–Methode in der Systemoptimierung von Mikrosystemkomponenten.* Doktorarbeit, Fachbereich 1 Physik/Elektrotechnik, Universität Bremen, Februar 1998.

[RB95] L. Relly und S. Blott: *Ein Speichersystem für abstrakte Objekte.* Technischer Bericht Swiss Federal Institute of Technology, Zürich, März, 1995.

[RC97] K. Ramamritham und P. K. Chrysanthis: *Executive Briefing: Advances in Concurrency Control and Transaction Processing.* IEEE Computer Society Press, 1997.

[Rei93] B. Reinwald: *Workflow-Management in verteilten Systemen.* Teubner-Verlag, Stuttgart, 1993.

[Rez92] L. Reznick: *Using Regular Expressions.* Sys Admin: The Journal for UNIX Systems Administrators, 1(3):59ff., September/Oktober 1992.

[Roe96] D. Roehrich: *perlxs – XS language reference manual*, 1996.

[Rog97] P. Rogers: *Object Database Management Systems*, Februar 1997.

[RS87] K. Ramamohanarao und J. Shepherd: *Answering Queries in Deductive Database Systems.* In: Lassez, J. L. (Herausgeber): *Proceedings of the Fourth International Conference on Logic Programming (ICLP '87)*, Seiten 1014–1033, Melbourne, Australia, Mai 1987. MIT Press.

[RS93] R. Rabenseifner und A. Schuch: *Comparison of DCE RPC, DFN-RPC, ONC and PVM.* Lecture Notes in Computer Science, 731:39–46, 1993.

[RSG95] S. Rozen, L. Stein und N. Goodman: *LabBase: A Database to Manage Laboratory Data in a Large–Scale Genome–Mapping Project.* IEEE Engineering in Medicine and Biology, November 1995.

[RzM96] M. Rosemann und M. zur Mühlen: *Der Lösungsbeitrag von Metamodellen beim Vergleich von Workflowmanagementsystemen.* Arbeitsberichte des Instituts für Wirtschaftsinformatik der Wilhelms Universität Münster, Juni 1996.

[SBES96] K.P. Scherer, P. Buchberger, H. Eggert und P. Stiller: *A Tool for Fabrication Related Design of Microstructures.* In: Reichl, H. und A. Heuberger (Herausgeber): *Micro System Technologies '96, 5th International Conference and Exhibition on Micro Electro, Opto, Mechanical Systems and Components*, Seiten 313–318, Berlin, 1996. VDE-Verlag.

[Sch93] R. L. Schwartz: *Learning Perl.* O'Reilly & Associates, Inc., 1993.

[Sch96a] S. Schreyjak: *Aspects of the integration of a componentware system with a workflow system.* In: *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications. OOPSLA '96*, Oktober 1996. Business Object Workshop II.

[Sch96b] H. Schuster: *Middlewareunterstützung für das verteilte Workflow Management System MOBILE.* In: *Proceedings Workshop W2: Geschäftsprozeßmodellierung und Workflowsysteme. Informatik'96*, Seiten 63–70, Klagenfurt, September 1996.

[Sch97] F. Schuster: *Architektur verteilter Workflow-Management-Systeme.* Doktorarbeit, Institut für Mathematische Maschinen und Datenverarbeitung (IMMD) Universität Erlangen–Nürnberg, Germany, 1997.

[Sch98a] A. Schmidt: *Dokumentation der W*FLOW–API Klassen.* Interner Bericht, Institut für Angewandte Informatik, Forschungszentrum Karlsruhe, Juni 1998.

[Sch98b] B. Schuller: *COPE – Web of CORBA Perl*, 1998. `http://www1.lunatech.com/cope/`.

[Sea69] J. R. Searle: *Speech Acts: An Essay in the Philosophy of Language.* Cambridge University Press, Cambridge, United Kingdom, 1969.

[SEGJ98] I. Sieber, H. Eggert, H. Guth und W. Jakob: *Design Simulation and Optimization of Microoptical Components.* In: *Novel Optical System Design and Large-Aperture Imaging*, Band 3430. SPIE, 1998.

[SG92] R. W. Scheifler und J. Gettys: *X Window System.* Digital Press, dritte Auflage, 1992.

[She94] A. Sheth: *Transactional Workflows: Research, Enabling Technologies, and Applications.* In: Elmagarmid, A. K. und E. Neuhold (Herausgeber): *Proceedings of the 10th International Conference on Data Engineering*, Seiten 403–403, Houston, TX, Februar 1994. IEEE Computer Society Press.

[Sie96] J. Siegel: *CORBA Fundamentals and programming.* Wiley, 1996.

[SIM95] Siemens Nixdorf Informationssysteme: *WorkParty Benutzerhandbuch*, 1995.

[SJHB96] H. Schuster, S. Jablonski, P. Heinl und Ch. Bussler: *A General Framework for the Execution of Heterogenous Programs.* In: *First IFCIS International Conference on Cooperative Information Systems*, Seiten 104–113, Brussels, Belgium, Juni 1996. IEEE-CS Press.

[SJKB94] H. Schuster, S. Jablonski, T. Kirsche und C. Bussler: *A Client/Server Architecture for Distributed Workflow Management Systems.* In: *Parallel and Distributed Information Systems (PDIS'94)*, Seiten 253–256, Los Alamitos, CA., USA, September 1994. IEEE Computer Society Press.

[SLS+93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos und J. Thomas: *The Rufus System: Information Organization for Semi–Structured Data.* In: Agrawal, R., S. Baker und D. Bell (Herausgeber): *Very large data bases, VLDB '93: proceedings of the 19th International Conference on Very Large Data Bases, August 24–27, 1993, Dublin, Ireland*, Seiten 97–107, Palo Alto, Calif., USA, 1993. Morgan Kaufmann Publishers.

[SM96] Wolfgang Schulze und Böhm Markus: *Geschaeftsprozessmodellierung und Workflow-Management*, Kapitel 16 - Klassifikation von Vorgangsverwaltungssystemen. Thomson Publishers, Bonn, 1 Auflage, 1996.

[SR93] A. Sheth und M. Rusinkiewicz: *On Transactional Workflows.* IEEE Data Eng. Bull., 16(2):37, Juni 1993.

[Sri97] S. Srinivasan: *Advanced Perl Programming.* O'Reilly & Associates, Inc., 1997.

[SSP99] E. Siever, S. Spainhour und N. Patwardhan: *Perl in a Nutshell*, Kapitel 8 Packages, Modules and Objects, Seiten 287–289. O'Reilly & Associates, Inc., Januar 1999.

[Sta87] R. Stallman: *GNU Emacs Manual.* Free Software Foundation, 1987.

[Ste98a] S. Steenbergen: *Positioning Perl to management* . SunWorld Online, September 1998. `http://www.sunworld.com/swol-09-1998/swol-09-champperl.html?0901a`.

[Ste98b] W. R. Stevens: *UNIX network programming: Networking APIs: Sockets and XTI*, Band 1. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, zweite Auflage, 1998.

[Sti99] P. Stiller, 1999. Persönliche Mitteilung.

[SU94] N. Sakamoto und K. Ushijima: *Designing and Integrating Human Genome Databases with Object-Oriented Technology.* Lecture Notes in Computer Science, 856:145ff., 1994.

[Sun87] Sun Microsystems Inc.: *RFC 1014: XDR: External Data Representation Standard*, Juni 1987. `ftp://ftp.math.utah.edu/pub/rfc/rfc1014.txt`.

[SUN99] *InfoBus 1.2 – Specification.* Mountain View, CA, 1999. `http://java.sun.com/beans/infobus/`.

[SV96] M. P. Singh und M. A. Vouk: *Scientific Workflows: Scientific Computing Meets Transactional Workflows.* In: *Proceedings NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions*, 1996.

[Szy98] C. A. Szyperski: *Component Software.* Addison-Wesley, 1998.

[Tan88] A. S. Tanenbaum: *Computer Networks.* Prentice Hall, Englewood Cliffs, NJ, zweite Auflage, 1988.

[TG97] D. Tombros und A. Geppert: *Managing Heterogeneity in Commercially Available Workflow Management Systems: A Critical Evaluation.* Technischer Bericht 4 SWORDIES, Universität Zürich – Institut für Informatik, Juni 1997.

[Tro98] Troll Tech AS, Etterstad, Norway: *Qt – Online Reference Documentation*, 1998. `http://www.troll.no/qt/`.

[UNJ+99] A. Uhlig, H. Neidholz, W. Jakob, A. Schmidt, W. Süss und H. Eggert: *ITI-SIM und SIMOT – Ein Werkzeugverbund für Simulation und Designoptimierung.* In: John, W., W. Gross und R. Laur (Herausgeber): *MIMOSYS/DEMIS Statusseminar*, Paderborn, Januar 1999.

[VK96] G. Valetto und G. E. Kaiser: *Enveloping Sophisticated Tools into Process-Centered Environments.* Journal of Automated Software Engineering, 3:309–345, 1996.

[Vos97] G. Vossen: *Transactional Workflows.* Lecture Notes in Computer Science, 1341:20ff., 1997.

[VWW96] G. Vossen, M. Weske und G. Wittkowski: *Prototypische Realisierung von WASA: Flexibles und plattformunabhängiges Workflow Management.* Technischer Bericht 23/96-I, Lehrstuhl für Informatik, Universität Münster, September 1996.

[WCS96] L. Wall, T. Christiansen und R. L. Schwartz: *Programming Perl.* O'Reilly & Associates, Inc., zweite Auflage, 1996.

[Wei98] T. Weis: *OpenParts – ein freies Objektmodell für Unix.* IX, August 1998.

[WF87] T. Winograd und R. Flores: *Understanding Computers and Cognition.* Addison-Wesley, Reading, Massachusetts, 1987.

[WFM96a] *WfMC – Interoperability – Abstract Specification V 1.0.* Technischer Bericht WFMC-TC-1012, The Workflow Management Coalition, Oktober 1996.

[WFM96b] *WfMC – Overview.* Technischer Bericht The Workflow Management Coalition, 1996.

[WFM98a] *WfMC – Conformance to Interface Standards.* Technischer Bericht The Workflow Management Coalition, September 1998. Online: `http://www.aiim.org/wfmc/standards/conformance.htm`.

[WFM98b] *WfMC – Interoperability – e-mail MIME Binding V 1.1.* Technischer Bericht WFMC-TC-1018, The Workflow Management Coalition, Oktober 1998.

[WFM98c] *WfMC – Process Definition Interchange V 1.0 Final.* Technischer Bericht WFMC-TC-1016-P, The Workflow Management Coalition, November 1998.

[WFM98d] *WfMC – Reference Model Workflow Interoperabity, the Key to E-Commerce and to Process Scalability*, Oktober 1998. Online Artikel: `http://www.aiim.org/wfmc/if4article.pdf`.

[WFM98e] *Workflow Client Application Programming Interface (WAPI) Naming Conventions V 2.0.* Technischer Bericht WFMC-TC-1009, The Workflow Management Coalition, 1998.

[WFM98f] *Workflow Management Coalition defines Interface between Workflow Engines and Process Definition Information*, September 1998. Online Artikel: `http://www.aiim.org/wfmc/if1article.pdf`.

[WFM99] *WfMC – Terminology & Glossary.* Technischer Bericht WFMC-TC-1011, The Workflow Management Coalition, Februar 1999.

[Wie98] J. Wiedmann: *Perl extensions for writing pRPC servers and clients*, 1998. Online Resource., z. B. `ftp://ftp.uni-hamburg.de/pub/soft/lang/perl/CPAN/modules/by-module/RPC/%JWIED/`.

[WR91] H. Waechter und A. Reuter: *The Contract Model.* Technischer Bericht Universität Stuttgart, Stuttgart, 1991.

[WWVBM96] J. Wainer, M. Weske, G. Vossen und C. Bauzer Medeiros: *Scientific workflow systems.* In: *Proceedings NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions*, 1996.

[Zei76] B. P. Zeigler: *Theory of Modeling and Simulation.* Wiley & Sons, New York, 1976.

[Zim99] B. Zimmermann: *Bend Zimmermann's Internet+WWW Kurs*, 1999. Online Glossar: `http://www.erlangen.netsurf.de/kurs/glossar.htm`.

Anhang A

Glossar

Der Glossar enthält eine Reihe von Begriffserläuterungen. Die einzelnen Begriffserläuterungen wurden aus verschiedenen Glossars [GLO98, GLO99a, GLO99b, GLO99c, Zim99, Hor99] oder Lexika [Eng93, HH97] entnommen und zum Teil aus dem englischen übersetzt.

4GL: (Fourth Generation Language)
Programmierstandard, Sprachen z. B. zur Programmierung von Frontends für Datenbanken.

ACID: ACID–Prinzip bei Transaktionen: (A) Atomizität, (C) Konkurrierende Zugriffe, (I) Isolation, (D) Dauerhaftigkeit.

ActiveX: Die *ActiveX*-Technologie ist eine Produktfamilie, die von Microsoft und anderen Firmen entwickelt wurde, um Web-Seiten Multimedia-Möglichkeiten wie Video, Audio, Animation und Virtuelle Realitäten hinzuzufügen. Dazu verwenden die Entwickler sogenannte ActiveX-Controls zum Verbinden und Einbetten von Objekten 'Object Linking and Embedding' (OLE). ActiveX ist zwar als plattformübergreifende Lösung gedacht, wird aber nicht von allen Betriebssystemen unterstützt.

API: (Application Programming Interface)
Schnittstelle für Anwendungsprogrammierung. Eine API ist eine Gruppe von Routinen, die von einem Programm aufgerufen werden können, um bestimmte Aufgaben zu erledigen. Betriebssysteme und Gerätetreiber besitzen eine API, über die sie mit einer Anwendung Daten austauschen können.

Applet: Eine "kleine Applikation", meist Java-Applet, wird innerhalb des HTML-Codes einer Web-Seite aufgerufen und vom Browser ausgeführt.

ASCII: (American Standard Code for Information Interchange)
7-bit Zeichencode, der von praktisch jedem Computerhersteller unterstützt wird, um Buchstaben, Zahlen und Sonderzeichen darzustellen. Dateien, die ausschließlich im ASCII-Textformat erzeugt wurden, enthalten keinerlei Gestaltung und/oder Schriftarten, aber sie können von jedem Computer gelesen werden.

Cache: Hintergrundspeicher, der als Zwischenspeicher für öfters benötige Daten dient. Die Nutzung des Caches ist für den Anwender transparent.

CGI: (Common Gateway Interface)
Standardisierte Programmierschnittstelle zum Datenaustausch zwischen Browser und Programmen auf dem Web-Server. Überwiegend sind diese Programme in Perl geschrieben und dienen hauptsächlich der Auswertung von HTML-Formularen.

CORBA: (Common Object Request Broker Architecture)
Spezifikation, die die Kommunikation von Objekten (Programmteilen) gleicher oder unterschiedlicher Programme beschreibt und 1991 für den Einsatz in objektorientierten Umgebungen entwickelt wurde.

CPAN: (Comprehensive Perl Archive Network)
CPAN bezeichnet eine Reihe von Servern, auf denen Perl selbst, als auch eine große Anzahl von entwickelten und frei verfügbaren Modulen zu finden sind.

DDE: (Dynamic Data Exchange)
Technik unter Windows, welche es Anwenderprogrammen ermöglicht, Daten miteinander auszutauschen. Der Datenaustausch selbst erfolgt dabei dynamisch. Wird eine der mittels DDE verbundenen Dateien geändert, erfolgt die Übernahme der Änderung in alle mit der betreffenden Datei kommunizierenden Dateien automatisch.

DDL: (Database Description Language)
Datenbank-Beschreibungssprache.

DML: (Data Manipulation Language)
Datenbank-Abfragesprache.

DTD: (Document Type Definition)
Die Document Type Definition ist eine formale Definition, wie ein SGML-Dokument strukturiert ist. So ist z. B. ein HTML-Dokument ein Dokument, das nach einer ganz bestimmten DTD aufgebaut ist, also den formalen Ansprüchen einer DTD entspricht.

Gopher: Vorgänger des World Wide Web und der erste Versuch, die immense Datenfülle des Internets zu strukturieren. Gopher ist menügesteuert und im Gegensatz zu den Hyperlinks im WWW hierarchisch gegliedert. Man kann also nicht nach Belieben von einer Seite zur anderen springen, sondern muß stets zum Ausgangspunkt zurück, um von dort eine andere Abzweigung in der Baumstruktur zu nehmen; veraltendes System.

E-Commerce : (electronic commerce)
Unter E-Commerce versteht man alle Formen von elektronischer Vermarktung und den Handel von Waren und Dienstleistungen über elektronische Medien wie das Internet.

EDI: (Electronic Data Interchange)
Datendienst für den papierlosen Austausch von Informationen in und

zwischen Unternehmen, der durch bestimmte Datenformate fest definiert ist und zunehmend auch übers Internet stattfindet.

FTP: (File Transfer Protocol)
Technischer Kommunikationsstandard, der die Dateiübertragung via Internet regelt.

Gateway: Netzverbindungsrechner, der Daten zwischen zwei sonst inkompatiblen Netzwerksystemen überträgt.

GUI: (Graphical User Interface)
Vorzufinden bei Software, die das Benutzen eines Systems oder einer Applikation durch den Einsatz von Mausklicktechnik, Icons und Scroll-Balken komfortabel macht.

HTML: (HyperText Markup Language)
Seitenbeschreibungssprache zum Erstellen eines Dokuments im World Wide Web.

HTTP: (Hypertext Transfer/Transmission Protocol)
Eines von vielen Internet-Protokollen, das für die Übertragung und Verknüpfung von Web-Seiten zuständig ist. Web-Adressen muß formell ein "http:" vorangestellt werden: Daran erkennt der Web-Browser, dass für die Übertragung das HTTP-Protokoll verwendet wird.

Java: Programmiersprache der Firma Sun Microsystems, die besonders geeignet ist zur Entwicklung von interaktiven Programmen innerhalb von Web-Seiten. Das Besondere an Java-Programmen ist, dass sie unabhängig vom jeweiligen Betriebssystem laufen, also z. B. gleichermaßen auf Apple-Computern wie auf Windows PCs.

JavaScript: Ursprünglich von der Firma Netscape Communications Corporation definierte und am meisten verbreitete Scriptsprache zur Verknüpfung von Programmcode mit statischen HTML-Seiten.

Middle–Tier: Logische Schicht in einer *Multi–Tier–Architektur* (s.u.), die von darunterliegenden Schichten wie etwa Betriebssystem, Hardware abstrahiert. Siehe auch *Middleware*.

Middleware: Software mit Schnittstellencharakter, die als Zwischenschicht zwischen Server- und Client-Komponenten eines verteilten Systems sitzt. Für den Anwender ist sie in der Regel unsichtbar, wenn verschiedene Anwendungen, Computer- oder Betriebssysteme z. B. mit Servertechniken verbunden werden.

Multi–Tier–Architektur: Unter einer *Multi–Tier–Architektur* oder auch *n–Tier–Architektur* versteht man die Verteilung einer Anwendung auf mehrere Softwareschichten. Hierbei beschreiben die einzelnen *Tiers* logische Programmeinheiten, etwa Benutzerfrontend, die Kommunikationsschicht (Middle Tier), die eigentliche Applikation und eine dahinterliegende Datenbank.

MIME: (Multipurpose Internet Mail Extensions)
Erweiterung der textbasierten e-Mail um Verfahren für den Versand von Binärdateien, Grafik, Video- und Audiodaten oder Faxen.

NFS: (Network File System)
Ein Dateisystem-Protokoll, das von Sun Microsystems entwickelt wurde. Es erlaubt den Zugriff auf Dateien im Netz, als ob sie auf einer lokalen Festplatte gespeichert wären.

Parser: Ein Software-Modul, das Dokumente oder Quelltexte syntaktisch analysiert und für die Weiterverarbeitung aufbereitet.

Perl: (Practical Extension and Report Language)
Eine frei verfügbare skriptbasierte Programmiersprache, die besonders beim Schreiben von CGI-Skripten auf Internet-Servern verwendet wird.

ODBC: (Open Database Connectivity)
Von der Firma Microsoft Anfang der 90er Jahre entwickeltes standardisiertes Verfahren, das den Zugriff auf Datenbanken erleichtert. Dadurch können Anwendungsprogramme Datenbanken der unterschiedlichsten Art beispielsweise ohne Kenntnis der Dateiformate nutzen.

OMG: (Object Management Group)
Die Object Management Group hat sich zum Ziel gesetzt, die Verbreitung der Objekttechnologie zu fördern und hat mit CORBA[1] eine Referenzarchitektur [OMG95a] vorgestellt. CORBA stellt eine plattform- und sprachunabhängige Infrastruktur für verteilte Objekte bereit. Die OMG hat sich anschließend entschlossen, ihre Standardisierungsbemühungen auf weitere Gebiete auszuweiten und mit den *CORBA Facilities* [OMG95b] Standardisierungsbestrebungen für eine Reihe von weiteren Teilgebieten vorgeschlagen, die als *Dienste*[2] bezeichnet werden. All diesen *Facilities* ist gemeinsam, daß sie auf der von CORBA bereitgestellten Objekttechnologie aufsetzen.

PlugIn: Zusatzprogramm für einen Web-Browser, das es dem Browser ermöglicht, Extrafunktionen darzustellen, die nicht im HTML-Format vorliegen, wie etwa in Web-Seiten enthaltene Video-Clips, 3D-Bilder oder Multimedia-Elemente. Ein Plug-in integriert sich voll in die Oberfläche der betreffenden Software und ist nicht ohne weiteres als Zusatz zu erkennen.

PVM: (Parallel Virtual Machine)
Bibliothek von Systemaufrufen für parallele Programmierung. PVM bildet eine abstrakte Schnittstelle oberhalb von miteinander verbundene Computern und stellt dem Programmierer eine virtuelle Maschine zur zur Lösung seiner Aufgaben zur Verfügung.

[1] **C**ommon **O**bject **R**equest **B**roker **A**rchitecture

[2] Es werden hierbei horizontale (Anwendungsgebiet unabhängige) und vertikale (auf Anwendungsgebiete spezialisierte) Dienste unterschieden.

RAD: (Rapid Application Development)
Ein Rapid Application Development System ist eine Software zur schnellen Erstellung von Anwendungen.

RDBMS: (Relational Database Management System)
In einem RDBMS werden die Daten in Tabellen (Relationen) gespeichert. Jeder einzelne Datensatz ist durch sein(e) Schlüsselattribut(e) eindeutig identifiziert. Beziehungen zwischen einzelnen Datensätzen werden über die Schlüsselattribute hergestellt.

RDF: (Resource Description Framework)
XML-Sprache, die das W3C zur Beschreibung von Metadaten wie site maps und Meta-Suchmaschinen empfiehlt

RFC: (Request For Comments)
Die "Request For Comments" sind Textdateien, die durch Arbeitsgruppen der Internet Society (ISOC) verfasst wurden. Neue Standards werden zunächst vorgeschlagen und zur Diskussion gestellt (daher "mit der Bitte um Stellungnahme"). Erst nachdem sie ausdiskutiert und für gut befunden worden sind, werden sie unter einer RFC-Nummer veröffentlicht, z. B. RFC 1166, RFC 959 (File Transfer Protocol (FTP)) oder RFC 1118 (Hitchhiker's Guide to the Internet).

RMI: (Remote Method Invocation)
RMI erlaubt es auf einem anderen Rechner eine Funktion aufzurufen, als wäre es auf dem eigenen.

RPC: (Remote Procedure Call)
Programm- oder Prozeduraufruf über ein Netzwerk.

Servlet: Name für Programme/Anwendungen, die in der Programmiersprache Java geschrieben sind und im Gegensatz zu Java-Applets nicht auf einem Client, sondern auf einem WWW-Server ausgeführt werden. Ein Java-Servlet könnte z. B. der Datenbank-Zugriff auf einem Server sein.

Script: Eine Textdatei, in der für einen Interpreter lesbare Befehle stehen. Beispiele für Skriptsprachen sind JavaScript, Basic und Perl.

SMTP: (Simple Mail Transfer Protocol)
Standardprotokoll zum Versenden von E-Mails.

Socket: Vverbindungsendpunkt der bei der Kommunikation mittels TCP/IP eingesetzt wird.

SQL: (Structured Query Language)
Standard-Abfragesprache für relationale Datenbanken.

Tags: Tags (englisch für Etiketten) sind Steuersymbole zur Formatierung von von HTML- oder SGML-Dokumenten.

TCP/IP: (Transmission Control Protocol/Internet Protocol)
Satz von Protokollen, nämlich TCP und IP, auf deren Zusammenwirken das Internet basiert. Da beide sich ergänzen (TCP ist Kontrollinstanz für IP), werden sie häufig zusammen erwähnt.

Telnet: Kategorie von Programmen, die ähnlich einem Terminal-Programm dem User direkten Zugriff auf einen anderen Computer im Internet ermöglichen. Wenn man die entsprechenden Zugriffsrechte auf diesen Rechner hat, kann man aus der Ferne Programme starten und Dateien bearbeiten.

Template: Vorlage, Muster, Schablone, Schema.

TIFF: (Tagged Image File Format)
Dateiformat für Rastergrafiken. Verschiedene Formatierungen (Tags) erlauben es Anwendungen, Teile der Grafik zu verarbeiten oder zu ignorieren.

Treiber: Ein Programm, das dem Computer mitteilt, wie er mit einem Drucker, einer Maus, einer Soundkarte etc. umgehen soll.

UNIX: Ende der 60er Jahre entwickeltes Betriebssystem für mittlere und große Computer. UNIX ist in der Programmiersprache *C* geschrieben und auf Multi-User- und Multi-Tasking-Betrieb ausgelegt. Die meisten Internet-Dienste benutzen Protokolle und Verfahren, die bereits Grundbestandteile von UNIX sind und erst viel später an andere Rechnersysteme angepaßt wurden.

URL: (Uniform Resource Locator)
Dieses Adressiersystem beschreibt im Internet den Pfad für eine Information.

UDP: (User Datagram Protocol)
Paketbasiertes Übertragungsprotokoll, das eine ungesicherte Kommunikation zwischen zwei Rechnern realisiert. Bei der Kommunikation mittels UDP kann nicht garantiert werden, daß ein abgesendetes Paket beim Empfänger ankommt.

USENET: Eigenständiges Netzwerk innerhalb des Internets, das sich in tausende, thematisch sortierte Unterbereiche, sogenannte Newsgroups, teilt. Hier werden Neuigkeiten und Dateien ausgetauscht, es wird diskutiert, philosophiert und bei technischen Problemen Hilfestellung geleistet. Wie im Internet üblich, ist das Usenet dezentral angelegt, d. h. es ist keine Zensur und kaum eine Kontrolle möglich.

W3: Spitzname, der die drei Ws von WWW bzw. World Wide Web bezeichnet.

WAIS: (Wide Area Information Servers)
Protokoll zum Auffinden von Internet-Ressourcen. (Das "S" wird auch schon mal mit System oder Search übersetzt). Indizierte Datensammlungen werden nach Wörtern oder Sätzen durchsucht. Auf Fundstellen wird dann verwiesen. Viele WWW-Browser, darunter NCSA-Mosaic und Netscape können auch mit WAIS-Diensten umgehen. Software zum Abrufen von Informationen aus Datenbanken, die über das gesamte Internet, also weltweit, verteilt sind.

Wildcard: Die Sonderzeichen * und ?, die, z. B. in Suchanfragen, als Platzhalter stellvertretend eingegeben werden können, entweder für ein einziges, beliebiges Zeichen (?) oder für einen oder mehrere Buchstaben (*).

WFMC: (Workflow Management Coalition)
Non–Profit Interessenverband der von Workflow-Management-System Anbietern und Anwendern gegründet wurde. Ziel der WFMC ist es, einen Standard im Bereich Workflowsysteme zu etablieren, der die Interoperabilität zwischen Produkten verschiedener Hersteller ermöglicht und die Verbreitung der Workflow Technologie fördern soll [WFM96b].

WPDL: (Workflow Process Definition Language)
Die WPDL soll die standardisierte Formulierung von Arbeitsabläufe erlauben [WFM98c, WFM98f].

Wrapper: Ein Wrapper ist ein Programm, das, um ein anderes Programm gelegt, für dieses eine neue Schnittstelle bereitstellt. Gründe können etwa die Anpassung der ursprünglichen Schnittstelle oder die Bereitstellung zusätzlicher Funktionalität sein.

WWW: (World Wide Web)
Populärster Internet-Dienst auf der Basis weltweit verteilter Hypertext-Dokumente, die Verweise auf weitere Dokumente, Multimedia-Elemente oder Software jeder Art enthält. Beschreibungssprache von WWW-Dateien ist HTML.

XML: (Extensible Markup Language)
Metasprache zur Erstellung von Dokumenten im World Wide Web. Mit XML lässt sich eine eigene formale Sprache erzeugen und die Struktur eines beliebigen Dokumententyps mit Hilfe einer DTD abbilden. Die syntaktischen Vorgaben selbst sind bei XML strenger als bei HTML. An den Beratungen über die Richtlinie XML 1.0 haben Firmen wie Adobe, Microsoft, Netscape, Sun und Hewlett-Packard mitgearbeitet. XML wurde im Februar 1998 verabschiedet.

XSL: (Extensible Style Language)
Vorschlag für eine XML-Stilsprache, die Ende August 1998 vom W3C als öffentlicher Entwurf vorgestellt und unter anderem von der Firma Microsoft vorangetrieben wurde.

Anhang B

Kurzfassung der $\mathcal{W}*FLOW$–API

Dieser Anhang enthält eine Kurzfassung der $\mathcal{W}*FLOW$–API. Eine ausführlichere Beschreibung der API findet sich in [Sch98a].

B.1 WildFlow::Object

Beschreibung	Abstrakte Klasse zur Definition der Basisfunktionalität aller Objekte (siehe Abschnitt 2.3).
Oberklassen	-keine-
Modul	WildFlow/Object.pm

Klassenmethoden:

siehe abgeleitete Klassen ab Anhang B.2

Instanzmethoden:

connect()

Erlaubt die Registrierung eines Objektes als Interessent an bestimmten Signalen eines anderen Objektes.

Parameter:

Name	Typ	Beschreibung
`$signal`	`STRING`	Signal, an dessen Signalisierung Interesse besteht.
`$obj`	`Instanz v. Typ` DB_Objekt	Objekt, an dessen Signalen Interesse besteht.
`$slot`	`Perlcode`	Auszuführende Methode bei Empfang des Signals.

emit()
Sendet ein Signal aus.

Parameter:

Name	Typ	Beschreibung
`$signal`	`STRING`	Signal, das gesendet wird
`$para`	`ARRAY`	Liste von Parametern, die mit diesem Signal übertragen werden

insert_attribute(`$dict`)
Trägt ein oder mehrere neue Attribute mit Werten für das Objekt ein.

Parameter:

Name	Typ	Beschreibung
`$dict`	`Dictionary`	Attribut-Wert Paare

attribute_list()
Liefert eine Referenz auf ein Array mit den definierten Attributnamen zurück.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des Attributes

attributes()
Liefert eine Referenz auf ein Dictionary zurück, das die Attributenamen und Werte enthält.

set_lock(`$mode`)
Sperrt das entsprechende Objekt.

Parameter:

Name	Typ	Beschreibung
`$mode`	`STRING`	Sperrmodus (READ|WRITE)

release_lock()
Löscht eine Sperre.

is_read_locked()
Liefert TRUE, falls das Objekt mit einer Lesesperre belegt ist.

is_write_locked()
Liefert TRUE, falls auf diesem Objekt eine Schreibsperre sitzt.

is_locked()
Liefert TRUE, falls auf diesem Objekt eine beliebige Sperre sitzt.

attribute()
Liefert den momentanen Wert des spezifizierten Attributes zurück.

is_defined(`$attribute_name`)

Liefert den Wert des spezifizierten Attributes zurück, falls dieses existiert, andernfalls NULL.

Parameter:

Name	Typ	Beschreibung
`$attribute_name`	`STRING`	Name des Attributes.

delete_attribute(`$att_name`)

Löscht den Attributeintrag einschließlich Wert für dieses Objekt.

Parameter:

Name	Typ	Beschreibung
`$att_name`	`STRING`	Name des Attributs, das aus dem Objekt entfernt werden soll.

generische Zugriffsmethoden:

Für jedes Attribut wird dynamisch eine Methode erzeugt, die den Namen des Attributs trägt. Wird die Methode ohne Parameter aufgerufen, so wird der aktuelle Wert des Attributs zurückgeliefert. Wird die Methode mit einem Parameter aufgerufen, so erhält das Attribut den Wert des Parameters.

Parameter:

Name	Typ	Beschreibung
`$value`	`SCALAR`	(optional) Falls dem Attribut, das dem Namen der Methode entspricht, ein neuer Wert zugewiesen werden soll, so erfolgt dies durch Angabe eines Parameters.

B.2 WildFlow::Workflow

Beschreibung	Klasse zur Generierung und Verwaltung von Workflows. Instanzen dieser Klasse repräsentieren komplette Workflows, die wiederum aus Aktivitäten bzw. Workflows bestehen können.
Oberklassen	WildFlow::Object
Modul	WildFlow/WorkFlow.pm

Klassenmethoden:

WildFlow::Workflow->new(`NAME=>$name`)

Konstruktor für eine Workflowinstanz. Als Parameter muß der Name des Workflows übergeben werden. Bei dem Namen handelt es sich um einen Dateinamen (ObjectStore DB). Existiert ein Workflow mit diesem Namen, so wird dieser geöffnet und zurückgeliefert, andernfalls wird ein neuer Workflow mit dem Namen angelegt.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des zu öffnenden bzw. zu erzeugenden Workflows.

WildFlow::Workflow->get(`NAME=>$name`)

Konstruktor für eine Workflowinstanz. Als Parameter muß der Name des Workflows übergeben werden. Bei dem Namen handelt es sich um einen Dateinamen (ObjectStore DB). Es muß ein Workflow mit diesem Namen existieren, andernfalls wird ein Fehler gemeldet.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des zu öffnenden Workflows.

Instanzmethoden:

new_activity(`NAME=>$name`)

Erzeugt eine neue Aktivitäteninstanz innerhalb des Workflows. Als Parameter muß ein eindeutiger Name für die anzulegende Instanz angegeben werden, anhand dessen sie referenziert werden kann.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name der Aktivität.

get_activity(`NAME=>$name`)

Liefert die anhand des Namens spezifizierte Aktivitäteninstanz zurück.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name der gewünschten Aktivität.

insert_activity(`$act_inst`)

Trägt eine Aktivität in den Workflow ein.

Parameter:

Name	Typ	Beschreibung
`$act_inst`	`WildFlow::Activity`	Instanz einer zuvor mittels WildFlow::Activity->new(...) erzeugten Aktivität.

delete_activity(`NAME=>$name`)

Löscht die anhand des Namens spezifizierte Aktivität.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name der zu löschenden Aktivität.

new_workflow(NAME=>$name)

Erzeugt einen neuen Behälter innerhalb der Workflowinstanz. In diesen Behälter können weitere Aktivitäten oder Behälter eingetragen werden. Als Ergebnis wird der neu erzeugte Behälter vom Typ WildFlow::WorkFlow zurückgeliefert. Die Unterscheidung zum Top Level Workflow liegt darin, daß ein mittels dieser Methode erzeugter Workflow stets in der selben Datenbank zu liegen kommt, wie sein Top Level Workflow, bei dem es sich stets um eine eigene Datenbank handelt.

Parameter:

Name	Typ	Beschreibung
$name	STRING	Name des Subworkflows.

insert_workflow(NAME=>$name,WORKFLOW=>$inst)

Neben Aktivitäten und untergeordneten Workflows ist es auch möglich, komplett eigenständige Workflows als Subworkflow einzutragen.

Parameter:

Name	Typ	Beschreibung
$name	STRING	Name des Subworkflows, anhand dessen dieser referenziert werden kann.
$inst	WildFlow::Workflow	Instanz des Workflows. Hierbei muß es sich um einen Top Level Workflow handeln, der durch eine eigene Datenbank repräsentiert wird.

get_workflow(NAME=>$name)

Liefert die anhand des Namens spezifizierte Subworkflow-Instanz zurück.

Parameter:

Name	Typ	Beschreibung
$name	STRING	Name des gewünschten Workflows.

delete_workflow(NAME=>$name)

Löscht den anhand des Namens spezifizierten Subworkflow.

Parameter:

Name	Typ	Beschreibung
$name	STRING	Name des zu löschenden Subworkflows.

parent()

Liefert den Parent Workflow zurück, bzw. NULL, falls es sich um den Top Level Workflow handelt.

top()
Liefert den Top Level Workflow einer Workflowinstanz zurück.

sub_workflow_list()
Liefert eine Referenz auf ein Array mit allen Subworkflows dieser Instanz zurück.

generische Zugriffsmethoden
Mittels dieser Methoden ist es möglich auf einzelne Aktivitäten und Subworkflows anhand des zuvor spezifizierten Namens zuzugreifen.

Beispiel:

```
$wf_2->create_activity(NAME=>'Vorbereitung');
...
$wf2->Vorbereitung()->define_handler(...);
```

Existiert weder eine Aktivität noch ein Subworkflow mit diesem Namen so wird geschaut, ob ein Attribut mit diesem Namen existiert (siehe Klasse WildFlow::Object).

destroy()
Löscht einen Workflow mitsamt seinen Aktivitäten.

B.3 WildFlow::Task

Beschreibung	Klasse zur Repräsentation einer Task. Eine Task repräsentiert ein vom Toolstarter aufgerufenes Programm. Eine Taskinstanz kann innerhalb eines vom ToolServer gestarteten *Perl* Skriptes generiert werden. Die Instanz enthält Informationen über verbundene Container sowie über die übergebenen Umgebungsvariablen.
Oberklassen	WildFlow::Object
Modul	WildFlow/Activity.pm

Klassenmethoden:

WildFlow::Task->new()

Konstruktor für eine Taskinstanz. Der Konstruktor extrahiert die Informationen, welche als Parameter an das Skript übergeben wurden.

Instanzmethoden:

get_activity()

Liefert die Aktivität zurück, innerhalb der diese Aktion mittels Zustandsübergang definiert wurde.

get_argument(`$name`)

Liefert eine Instanz des betreffenden Übergabeparameters zurück.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des Parameters. Die Parameter, die von $\mathcal{W}*FLOW$ übergeben werden, werden nicht aufgrund ihrer Position unterschieden, sondern besitzen einen bestimmten Namen, anhand dessen sie referenziert werden können.

get_activity_instance()

Liefert die Aktivitäteninstanz zurück, von welcher aus diese Task gestartet wurde.

B.4 WildFlow::Activity

Beschreibung	Instanzen dieser Klasse repräsentieren Aktivitäten. Eine Aktivität kann eine beliebige Anzahl von zu definierenden Zuständen besitzen, zwischen denen Zustandsübergänge definiert werden können. Die Instanzen dieser Klasse repräsentieren Schablonen für die Laufzeitinstanzen vom Typ WildFlow::ActivityInstance, welche aus den Instanzen dieser Klasse generiert werden können.
Oberklassen	WildFlow::Object
Modul	WildFlow/Activity.pm

Klassenmethoden:

WildFlow::Activity->new(`NAME=>$name, WORKFLOW=>$workflow`**)**

Erzeugt eine neue Instanz dieser Klasse. Eine Instanz der Klasse WildFlow::Activity kann nur innerhalb eines Workflows vom Typ WildFlow::Workflow angelegt werden. Innerhalb dieses Workflows wird die Aktivität durch ihren Namen identifiziert.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name der Aktivität innerhalb des Workflows.
`$workflow`	`WildFlow::Workflow`	Workflowinstanz innerhalb der die Aktivität eingetragen werden soll.

Instanzenmethoden:

new(`USER=>$user`**)**

Instanzenmethode, mittels der aus einer Aktivitäteninstanz Laufzeitinstanzen erzeugt werden können. Die Laufzeitinstanzen sind vom Typ WildFlow::ActivityInstance und gehorchen den in der Aktivität definierten Zustandsübergängen.

Parameter:

Name	Typ	Beschreibung
`$user`	`STRING`	Name des Benutzers der Laufzeitinstanz.

set_states(`$state_list`)

Mittels dieser Methode werden die möglichen Zustände definiert, welche die von dieser Instanz abgeleiteten Laufzeitinstanzen annehmen können.

Parameter:

Name	Typ	Beschreibung
`$state_list`	`Array`	Liste aller möglichen Zustände.

get_states()

Liefert eine Referenz auf ein Array, in dem alle möglichen Zustände der Aktivität als Strings aufgeführt sind.

get_handler(`STATE=>$state, NAME=>$name`)

Liefert das Handle vom Typ WildFlow::ActivityHandler zurück, das einen bestimmten Zustandsübergang repräsentiert. Das Handle wird anhand des Ausgangszustandes, sowie dem Namen des Übergangs wie er in der Methode define_handler(...) definiert wurde, spezifiziert.

Parameter:

Name	Typ	Beschreibung
`$state`	`STRING`	Name des Ausgangszustand, wie er bei define_handler(...) definiert wurde.
`$name`	`STRING`	Name des Zustandsübergangs, wie er bei define_handler(...) definiert wurde.

set_container(`CONTAINER=>$inst, TYPE=>$type`)

Fügt einen Container in die Containerliste der Aktivität ein.

Parameter:

Name	Typ	Beschreibung
`$container`	`WildFlow::Container`	Instanz des Containers der eingefügt werden soll.
`$type`	`STRING`	'INPUT', 'OUTPUT' bzw 'INOUT'

define_handler(`$state_1=>$state_2, NAME=>$name, PACKAGE=>$packg, METHOD=>$method, CODE=>$code`)

Diese Methode definiert einen Zustandsübergang zwischen zwei durch die Methode set_states(...) definierten Zuständen. Es ist sowohl möglich *Perl*-Code oder Paket-/Methodenname anzugeben. Als Ergebnis des Methodenaufrufs wird eine Instanz vom Typ

WildFlow::ActivityHandler (siehe Anhang B.5) zurückgeliefert, der diesen Zustandsübergang repräsentiert.

Parameter:

Name	Typ	Beschreibung
`$state_1=> $state_2`	`DICTIONARY Eintrag`	Hierbei handelt es sich um den spezifischen Zustandsübergang.
`$name`	`STRING`	Ein eindeutiger Bezeichner für diesen Übergang. Der Name muß dahingehend eindeutig sein, daß kein zweiter Übergang vom Ausgangszustand ($state_1) den gleichen Namen besitzt.
`$packg`	`STRING`	Paketname einer im Toolserver abgelegten Methode. Wird 'PACKAGE' angegeben, so muß auch der Parameter 'METHOD' angegeben werden und 'CODE' (s.u.) darf nicht angegeben werden.
`$method`	`STRING`	Methodenname einer im Toolserver abgelegten Methode. Wird 'METHOD' angegeben, so muß auch der Parameter 'PACKAGE' angegeben werden und 'CODE' (s.u.) darf nicht angegeben werden.
`$code`	`STRING`	*Perl* Code Fragment, das ausgeführt werden soll. Diesen Funktionen können beim Aufruf der Übergänge Parameter mit übergeben werden (s.u.). Wird 'CODE' spezifiziert, so dürfen weder 'PACKAGE' noch 'METHOD' spezifiziert sein.

get_container(`$name`)

Liefert einen zuvor mittels set_container(...) eingetragenen Container zurück.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des Containers der zurückgeliefert werden soll.

get_container_list(`NAME=>$reg, TYPE=>$type`)

Liefert eine Referenz auf ein Array von Containern zurück, die anhand der beiden optionalen Parameter spezifiziert werden können. Wird kein Parameter spezifiziert, so werden alle mit dieser Aktivität verknüpften Container zurückgegeben.

Parameter:

Name	Typ	Beschreibung
`$reg`	`STRING`	Name in Form eines regulären Ausdrucks. Nur Container, deren Namen mit dem regulären Ausdruck übereinstimmen werden zurückgeliefert.
`$type`	`STRING`	Typ des Containers ('INPUT', 'OUTPUT' bzw. 'INOUT').

workflow()

Liefert die Workflowinstanz zurück, in der sich die Aktivität befindet.

get_runtime_instance_list()

Liefert eine Referenz auf ein Array, das alle momentan vorhandenen Laufzeitinstanzen dieser Aktivität zurückliefert.

attributes()

Liefert ein Dictionary mit den Atributnamen und Werten zurück.

B.5 WildFlow::ActivityHandler

Beschreibung	Instanzen dieser Klasse repräsentieren Zustandsübergänge. Diese werden durch Aufruf der Instanzenmethode define_handler(...) der Klasse WildFlow::Activity generiert. Zustandsübergänge sind mit auszuführenden Aktionen verknüpft. Hierbei kann es sich sowohl um *Perl* Code, als auch um Toolservereinträge handeln. Da der Toolserver lokale und entfernte Aufrufe unterstützt, wird gegebenenfalls ein Toolstarter Server Prozeß benötigt, der auf dem Rechner ablaufen muß, auf dem Anwendungen Remote gestartet werden sollen.
Oberklassen	WildFlow::Object
Modul	WildFlow/Activity.pm

Klassenmethoden:

keine

Instanzenmethoden:

set_precondition(`$pre_cond`)

Diese Methode erlaubt die Definition einer Vorbedingung. Wird eine Vorbedingung angegeben, so kann ein Zustandsübergang nur dann stattfinden, wenn diese Vorbedingung zu TRUE evaluiert.

Parameter:

Name	Typ	Beschreibung
`$pre_cond`	`STRING`	String, der einen auszuwertenden *Perl* Ausdruck enthält.

check_precondition()

Liefert das Ergebnis einer eventuell vorhandenen Vorbedingung zurück. Ist keine Vorbedingung angegeben, so wird TRUE zurückgeliefert.

execute(...)

Führt die mittels define_handler(*para*) definierte Aktion aus. Dabei wird zuvor eine eventuell definierte Vorbedingung ausgewertet. Nur wenn diese Bedingung erfüllt ist, wird die spezifizierte Aktion ausgeführt.

Parameter:

Der Methode können eine beliebige Anzahl von Parametern mit übergeben werden.

Handelt es sich bei der auszuführenden Aktion um *Perl* Code, so können die Parameter wie in *Perl* üblich angegeben werden.

Handelt es sich jedoch um eine Aktion die durch den Toolserver aufgelöst wird, so müssen Parameter vom Typ WildFlow::Parameter übergeben werden.

x_praxis_rpc(@*para*)

Liefert die durch define_handler(...) definierte Steuerungsdatei vom Typ application/x-praxis-rpc zurück.

Parameter:

Der Methode können eine beliebige Menge Parameter vom Typ WildFlow::Parameter (Anhang B.7) übergeben werden.

set_postrun($perlcode, $is_fatal)

Dient zur Registrierung einer *Perl* Routine, die nach erfolgreicher Durchführung der mit diesem Übergang verbundenen Aktion ausgeführt werden soll.

Parameter:

Name	Typ	Beschreibung
$perlcode	STRING	Ein String, der den auszuführenden Code enthält.
$is_fatal	BOOLEAN	Normalerweise ist das Scheitern einer Postrunsequenz kein Grund den Übergang als gescheitert anzusehen. Wird dieser Parameter auf '1' gesetzt, so führt ein Compilierungsfehler oder die Rückgabe von 0 zum Scheitern des Überganges.

is_success()

Wurde ein Übergang duchgeführt, so kann mittels dieser Methode überprüft werden, ob die auszuführende Aktion korrekt durchgeführt wurde.

stdout()

Wurde eine Aktion korrekt ausgeführt, so liefert diese Methode die Ausgaben, welche vom Programm auf STDOUT getätigt wurden, zurück.

stderr()

Wurde eine Aktion korrekt ausgeführt, so liefert diese Methode die Ausgaben, welche vom Programm auf STDERR getätigt wurden, zurück.

error()

Wurde eine Aktion nicht korrekt ausgeführt (is_success() liefert FALSE), so kann der Grund hierfür dieser Methode entnommen werden.

B.6 WildFlow::ActivityInstance

Beschreibung	Instanzen dieser Klasse repräsentieren Laufzeitinstanzen von Aktivitäten. Sie durchlaufen verschiedene, zuvor innerhalb einer Aktivitäteninstanz definierte Zustände und führen hierbei bestimmte Aktionen aus. Weitere Informationen zu technischen Aspekten bei der Ausführung von externen Applikationen befinden sich im Abschnitt B.5.
Oberklassen	WildFlow::Object
Modul	WildFlow/Activity.pm

Klassenmethoden:

keine

Instanzenmethoden:

valid_transitions()

Diese Methode liefert, ausgehend vom aktuellen Zustand dieser Laufzeitinstanz, alle ausführbaren Zustandsübergänge zurück. Der Rückgabewert ist eine Referenz auf ein Array, das die Namen der Übergänge enthält.

valid_handler()

Diese Methode liefert, ausgehend vom aktuellen Zustand dieser Laufzeitinstanz, alle möglichen Zustandsübergänge zurück. Die Übergänge werden als Instanz vom Typ WildFlow::ActivityHandler zurückgeliefert. Der Rückgabewert ist eine Referenz auf ein Array, das die Handler enthält.

get_handler(`$trans`)

Liefert das Handle vom Typ WildFlow::ActivityHandler, das einen bestimmten Zustandsübergang repräsentiert. Das Handle wird anhand des aktuellen Zustandes, sowie dem Namen des Übergangs spezifiziert.

Parameter:

Name	Typ	Beschreibung
`$trans`	`STRING`	Name des Zustandsübergangs, wie er bei define_handler(...) definiert wurde.

is_valid_transition(`$trans`)

Liefert TRUE, falls der als Parameter spezifizierte Übergang möglich ist.

Parameter:

Name	Typ	Beschreibung
`$trans`	`STRING`	Name des Zustandsüberganges.

get_states()

Liefert eine Referenz auf ein Array, in dem alle erreichbaren Zustände der Aktivität als Strings aufgeführt sind.

change_state(`$name`)

Führt einen Zustandsübergang vom momentanen Zustand zu dem Zustand durch, der durch den als Parameter übergebenen Zustandsübergang definiert ist. Hierbei wird, soweit vorhanden, die für diesen Handler definierte Postcondition ausgeführt und nur dann der neue Zustand angenommen, wenn diese erfolgreich durchgeführt werden kann.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des Übergangs.

activity()

Liefert die zugehörige Instanz der Klasse WildFlow::Activity zurück.

is_final_state()

Liefert TRUE, falls sich die Instanz in einem finalen Zustand befindet, das heißt, daß keine weiteren Zustände von diesem Zustand aus angesprungen werden können.

get_container(`$name`)

Liefert die Containerinstanz zurück, die für die zugehörige Instanz vom Typ WildFlow::Activity definiert wurde.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des Containers.

get_container_list(`NAME=>$name, TYPE=>$type`)

Liefert einen Zeiger auf eine Liste von Containern zurück, die anhand der beiden optionalen Parameter spezifiziert werden können. Wird kein Parameter spezifiziert, so werden alle mit der zugehörigen Aktivität verknüpften Container zurückgegeben.

Parameter:

Name	Typ	Beschreibung
`$name`	`SRING`	Name in Form eines regulären Ausdrucks.
`$type`	`STRING`	Typ des Containers ('INPUT', 'OUTPUT' bzw. 'INOUT').

receive_event($trans, $para_list)

Führt den als Parameter angegebenen Übergang durch.

Parameter:

Name	Typ	Beschreibung
$trans	STRING	Name des auszuführenden Überganges.
$para_list	ARRAY	Liste von beliebigen Parametern.

x_praxis_rpc($para_list)

Liefert eine bestimmte Steuerungsdatei vom MIME-Typ application/x-praxis-rpc zurück. Die Steuerungsdatei wird mit den an die Methode übergebenen Parametern zurückgeliefert.

Parameter:

Name	Typ	Beschreibung
$para_list	ARRAY	Der Array enthält Instanzen vom Typ WildFlow::Parameter, die in die x-praxis-rpc Datei eingetragen werden.

current_state()

Liefert den Namen des aktuellen Zustandes der Instanz zurück.

generische Methoden

Wie bei der Beschreibung der Methode valid_transitions(...) erwähnt, ist es möglich einen Zustandsübergang dadurch auszulösen, daß eine Methode, welche den Namen des Zustandsübergangs trägt, aufgerufen wird. Dieser Methode können beliebige benötigte Parameter mit übergeben werden.

Existiert kein gültiger Zustandsübergang für diese Instanz, wird untersucht, ob ein Parameter mit diesem Namen existiert. Ist dies der Fall, so wird der Wert zurückgeliefert, bzw. falls ein zusätzlicher Parameter übergeben wurde, der neue Wert gesetzt.

Parameter:

Name	Typ	Beschreibung
$param_list	ARRAY	Liste von beliebigen Parametern.

B.7 WildFlow::Parameter

Beschreibung	Instanzen dieser Klasse repräsentieren Parameter, welche den Aktionen, die als Toolservereinträge vorliegen, übergeben werden können.
Oberklassen	WildFlow::Object
Modul	WildFlow/Activity.pm

Klassenmethoden:

WildFlow::Parameter->new(`TYPE=>$type, CONTENT_TYPE=>$cont_type, CONTENT=>$content`**)**

Konstruktor für eine Parameterinstanz.

Parameter:

Name	Typ	Beschreibung
`$type`	`STRING`	Entweder 'FILE' oder 'PARA'.
`$cont_type`	`STRING`	Der MIME-Typ, falls es sich um einen Parameter vom Typ 'FILE' handelt oder eine der folgenden Kennungen ('STRING','FLOAT', 'INTEGER'), falls es sich um einen Parameter vom Typ 'PARA' handelt.
`$content`	`STRING`	URL, im Falle daß es sich um einen Parameter vom Typ FILE handelt, andernfalls der Wert des Parameters als String.

Instanzenmethoden:

type()

Liefert den Typ des Parameters zurück.

content_type()

Liefert den Inhaltstyp des Parameters zurück.

content()

Liefert den Inhalt bzw. eine Referenz in Form einer URL des Parameters zurück.

B.8 WildFlow::Container

Beschreibung	Klasse zur Generierung und Verwaltung von Containern. Mittels der Methoden dieser Klasse lassen sich beliebige Container zur Aufnahme von Daten erzeugen.
Oberklassen	WildFlow::Object
Modul	WildFlow/Container.pm

Klassenmethoden:

WildFlow::Container->new(`NAME=>$name, REPOSITORY=>$rep, MODE=>$mode, PATH=>$path, HOST=>$host`**)**

Konstruktor für eine Containerinstanz. Je nach übergebenen Parametern wird ein neuer Container erzeugt, bzw. ein existierender Container geöffnet. Im Normalfall wird die Methode jedoch nur zum Erzeugen eines neuen Containers eingesetzt, da es zum Zugriff auf bestehende Container komfortablere Möglichkeiten über das Container Repository gibt.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des Containers.
`$rep`	`STRING` oder `WildFlow::Repository`	Ein Repository (siehe Klasse WildFlow::ContainerRepository(), Anhang B.9) vereint eine Reihe zusammengehörender Container. Es ist durch seinen Pfadnamen eindeutig bestimmt.
`$host`	`STRING`	Rechner, auf dem der Datenbankserver läuft.
`$mode`	`STRING`	Modus der Containergenerierung. Im Mode 'NEW' wird ein neuer Container erzeugt. Im Mode 'GET' muß zusätzlich der Parameter PATH (s.u.) angegeben werden.
`$path`	`STRING`	Name der Datenbankdatei, welche den Container enthält.

WildFlow::Container->get_instance(`$id`)

Diese Methode liefert eine Instanz dieser Klasse zurück. Als Identifikator muß eine ID angegeben werden, welche mit der Instanzenmethode id() (s.u.) erzeugt werden kann.

Parameter:

Name	Typ	Beschreibung
`$id`	`STRING`	ID der zu extrahierenden Instanz.

Instanzmethoden:

id()

Liefert einen internen Identifikator für diesen Container zurück. Der Identifikator kann als Argument beim Aufruf der Klassenmethode get_instance(...) genutzt werden, die dann diese Instanz wieder zurückliefert.

destroy()

Löscht den Container. Ist der Container in ein Repository eingetragen, so wird der Eintrag auch hieraus entfernt.

insert_object(`$obj`)

Trägt ein Containerobjekt in den Container ein.

Parameter:

Name	Typ	Beschreibung
`$obj`	`ContainerObject`	Instanz vom Typ Containerobjekt, die eingetragen werden soll.

new_object(`$dict`)
Erzeugt ein neues Containerobjekt und trägt es in den Container ein. Es können eine Reihe von Parameter-Wert Paare als Attribute an das neue Objekt mitgegeben werden.

Parameter:

Name	Typ	Beschreibung
`$dict`	`DICTIONARY`	Attribute des Objekts.

insert_attribute(`$dict`)
Trägt ein oder mehrere neue Attribute mit Werten für den Container ein.

Parameter:

Name	Typ	Beschreibung
`$dict`	`DICTIONARY`	Attribut-Wert Paare.

get_object(`$obj_name`)
Liefert das Objekt mit dem entsprechenden Namen zurück.

Parameter:

Name	Typ	Beschreibung
`$obj_name`	`STRING`	Name des Objekts, das zurückgeliefert werden soll.

delete_object(`$obj_name`)
Löscht das Objekt mit dem entsprechenden Namen aus dem Container.

Parameter:

Name	Typ	Beschreibung
`$obj_name`	`STRING`	Name des Objekts, das gelöscht werden soll.

get_list(), get_objects()
Liefert einen Zeiger auf ein Array aller Containerobjekte zurück.

set_default(`$obj`)
Setzt das Defaultobjekt innerhalb eines Containers. Das Objekt kann entweder durch seinen Namen oder seiner Instanz festgelegt werden.

Parameter:

Name	Typ	Beschreibung
`$obj`	`STRING bzw. ContainerObject`	Instanz/Name.

get_default()
Liefert das Default Objekt zurück.

get_last()
Liefert das zuletzt eingetragene Objekt zurück.

get_first()
Liefert das zuerst eingetragene Objekt zurück.

path()
Liefert den Pfad zurück, unter dem die Containerdatenbank abgelegt ist.

close()
Schließt die Datenbank, die mit diesem Container verbunden ist.

B.9 WildFlow::ContainerRepository

Beschreibung	Diese Klasse dient zur Gruppierung und Verwaltung einzelner Container. Container können in Repositories zusammengefaßt werden. Ein Repository stellt eine Reihe von Zugriffs- und Verwaltungsmethoden für die Container zur Verfügung. Ein Repository ist eindeutig durch einen Pfad gekennzeichnet, der beim Erzeugen des Repositories angegeben werden muß.
Oberklassen	WildFlow::Object
Modul	WildFlow/Container.pm

Klassenmethoden:

WildFlow::ContainerRepository->create(`$path`)
Erzeugt ein neues Containerrepository bzw. öffnet ein Repository das sich in diesem Verzeichnis befindet. Als Parameter wird der Pfad des Repository angegeben. Hierbei handelt es sich um ein Verzeichnis, das existieren muß. Das Repositoryfile hat einen festen Namen (REPOSITORY.db). Alle mittels create_container() erzeugten Container werden in diesem Verzeichnis abgelegt. Es ist jedoch auch möglich Container die woanders liegen in diesem Repository zu verwalten (mittels insert_container()).

Die zurückgelieferte Instanz wird anschließend zum Zugriff auf das Repository genutzt.

Parameter:

Name	Typ	Beschreibung
`$path`	`STRING`	Verzeichnispfad, innerhalb dem das Repository angelegt werden soll. Dieser Name dient gleichzeitig als Identifikator für ein bestimmtes Repository.

WildFlow::ContainerRepository->open(`$path`)
Öffnet ein existierendes Repository, das sich in dem durch den Parameter bezeichneten Verzeichnis befindet. Als Parameter wird der Pfad des Repository angegeben. Hierbei handelt es sich um ein Verzeichnis, das existieren muß.

Die zurückgelieferte Instanz wird anschließend zum Zugriff auf das Repository genutzt.

Parameter:

Name	Typ	Beschreibung
`$path`	`STRING`	Identifikator (Verzeichnispfad) des existierenden Repositories.

Instanzenmethoden:

container_list()
Liefert eine Referenz auf ein Array mit allen Repositoryeinträgen zurück. Die einzelnen Einträge sind Instanzen vom Typ WildFlow::Container.

get_container(`$name`)
Liefert eine Conatinerinstanz zurück. Die Instanz wird anhand des Namens identifiziert.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des Containers.

clear()
Löscht alle Container aus dem Repository. Die Container werden nicht nur aus dem Repository, sondern auch physikalisch gelöscht.

delete()
Löscht das Repository mit allen Containern. Die Container werden physikalisch gelöscht.

container_exist(`$name`)
Testet, ob ein Container mit dem als Parameter übergebenen Namen im Repository existiert.

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name des Containers.

path()
Liefert den Verzeichnispfad unter dem das Repository identifiziert werden kann.

insert_container(`$cnt`)
Trägt eine Instanz eines Containers ins Repository ein.

Parameter:

Name	Typ	Beschreibung
`$cnt`	`WildFlow::Container`	Instanz, welche ins Repository eingetragen werden soll.

create_container(`$cont_name`**)**

Erzeugt eine neue Containerinstanz innerhalb des Repositories. Die Containerinstanz wird zurückgeliefert. Der übergebene Parameter `$cont_name` wird an die Methode Container->new() weitergereicht.

Parameter:

Name	Typ	Beschreibung
`$cont_name`	`STRING`	Name des neuen Containers.

delete_container(`$cont`**)**

Löscht den Container anhand seines Namens oder seiner Objektreferenz. Es wird sowohl der Eintrag im Repository, als auch der Container gelöscht.

Parameter:

Name	Typ	Beschreibung
`$cont`	`STRING bzw. WildFlow::Container`	Der zu löschende Container kann sowohl durch eine Containerinstanz, als auch durch den eindeutigen Namen des Containers innerhalb des Repositories gelöscht werden.

delete_entry(`$cont`**)**

Löscht den Containereintrag aus dem Repository. Der Container bleibt weiterhin erhalten.

Parameter:

Name	Typ	Beschreibung
`$cont`	`STRING bzw. WildFlow::Container`	Der zu löschende Container kann sowohl durch eine Containerinstanz, als auch durch den eindeutigen Namen des Containers innerhalb des Repositories gelöscht werden.

B.10 WildFlow::ContainerObject

Beschreibung	Klasse zur Verwaltung von Containerobjekten. Mittels den Methoden dieser Klasse können Objektinstanzen erzeugt, ihnen Attribute und Werte sowie Komponenten zugewiesen werden.
Oberklassen	WildFlow::Object
Modul	WildFlow/ContainerObject.pm

Klassenmethoden:

WildFlow::ContainerObject->new(`CONTAINER=>$cnt, DESCRIPTION=>$desc, MODE=>$mode`**)**

Konstruktor zur Erzeugung einer neuen Instanz der Klasse. Die Parameter werden in Form von Attribut-Wert Paaren an die Methode übergeben.

Parameter:

Name	Typ	Beschreibung
`$container`	`WildFlow::Container`	Gibt den Container an, in welchen die Instanz eingetragen werden soll.
`$desc`	`STRING`	Beschreibung
`$mode`	`STRING`	(NEW|GET)

Instanzmethoden:

new_component(`NAME=>$name, DESCRIPTION=>$desc`)

Erzeugt innerhalb des Objektes eine neue Komponente. Die Methode kapselt dazu die Aufrufe WildFlow::Component->new(...) und $self->insert_component(...).

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name der Komponente innerhalb des Objekts.
`$desc`	`STRING`	Beschreibung

new_set_component(`NAME=>$name, DESCRIPTION=>$desc`)

Erzeugt innerhalb des Objektes eine neue Set-Komponente. Die Methode kapselt dazu die Aufrufe WildFlow::SetComponent->new(...) und $self->insert_component(...).

Parameter:

Name	Typ	Beschreibung
`$name`	`STRING`	Name der Komponente innerhalb des Objekts.
`$desc`	`STRING`	Beschreibung

insert_component(`$comp`)

Trägt eine zuvor erzeugte Komponente in diesen Container ein. Die Komponente wird in Form eines Attribut-Wert Paares an die Methode übergeben. Der Name des Attributs entspricht hierbei dem Namen, mittels dem die Komponente später angesprochen werden kann. Beim Wert handelt es sich um die Instanz der Komponente.

Parameter:

Name	Typ	Beschreibung
`$comp`	`WildFlow::Component`	Einzutragende Komponente.

component_list()

Liefert eine Referenz auf eine Liste aller innerhalb dieses Objektes gespeicherten Komponenten vom Typ WildFlow::Component zurück.

container()

Liefert die Containerinstanz zurück, innerhalb der die Komponenteninstanz gespeichert ist.

B.11 WildFlow::Component

Beschreibung	Klasse zur Verwaltung von Komponenten. Bei den Komponenten handelt es sich um die eigentlichen Datensätze der Container. Es wird zwischen verschiedenen Komponenten unterschieden, je nach dem, ob die Daten direkt innerhalb der Komponenten gespeichert werden oder ob die Komponenten die Daten referenzieren. Für den Zugriff auf die Daten ist diese Unterscheidung jedoch transparent.
Oberklassen	WildFlow::Object
Modul	WildFlow/Component.pm

Klassenmethoden:

WildFlow::Component->new(`OBJECT=>$obj, DESCRIPTION=>$desc`**)**

Konstruktormethode zur Erzeugung einer neuen Komponenteninstanz vom Typ WildFlow::SetComponent. Die Parameter weden in Form von Attribut-Wert Paaren übergeben.

Parameter:

Name	Typ	Beschreibung
`$obj`	`ContainerObject`	Gibt das Containerobjekt an, innerhalb dem die Komponente abgelegt werden soll. Dies ist notwendig, damit der Speicher für die Instanz in der richtigen Datenbank allokiert werden kann. Der eigentliche Eintrag in das Containerobjekt erfolgt erst anschließend.
`$desc`	`STRING`	Beschreibung für diese Komponente.

Instanzenmethoden:

object()

Liefert die Instanz eines Containerobjektes zurück, innerhalb dem die Komponente abgelegt wurde.

set_content(`TYPE=>$type, CONTENT_TYPE=>$cont_type, URL=>$url, CONTENT=>$cont`**)**

Setzt den Inhalt der Komponente. Der Inhalt kann sowohl direkt als String angegeben werden, als auch als URL. Wird der Inhalt als URL angegeben, so kann weiterhin spezifiziert werden ob der Datensatz in die Komponente hineinkopiert werden soll oder ob der Datensatz referenziert werden soll. Weiterhin wird der MIME-Typ des Datensatzes angegeben.

Parameter:

Name	Typ	Beschreibung
`$type`	`STRING`	(COPY|REFERENCE) Soll der Datensatz in die Komponente kopiert (COPY) werden oder soll eine Referenz gespeichert werden (REFERENCE).
`$cont_type`	`STRING`	Gibt den Datentyp im MIME-Format an. Wird die Information, um welchen MIME-Type es sich handelt vom Server mitgeliefert, so kann dieser Parameter weggelassen werden.
`$url`	`STRING`	Referenz auf den Datensatz. Hierbei kann es sich um eine beliebige URL handeln. Es spielt dabei keine Rolle ob der Datensatz direkt gespeichert oder referenziert werden soll (dies wird durch den Parameter TYPE spezifiziert).
`$cont`	`STRING`	Neben der Spezifikation des Inhalts der Komponente durch eine URL ist es weiterhin möglich, den Inhalt direkt durch einen String anzugeben. Wenn CONTENT angegeben wird, darf URL nicht gesetzt sein und umgekehrt.

get_content(`$type`)

Liefert den Inhalt oder die Referenz auf den Datensatz zurück. Wurde die Komponente mittels dem URL-Parameter der Methode `set_content()` gefüllt, so kann durch Angabe des Parameters TYPE spezifiziert werden, ob der Inhalt des Datensatzes oder die Referenz auf den Datensatz zurückgeliefert werden soll. Hierbei ist es egal, ob die Komponente intern den eigentlichen Datensatz oder die Referenz auf diesen gespeichert hält.

Parameter:

Name	Typ	Beschreibung
`$type`	`STRING`	(REFERENCE|CONTENT) Spezifiziert, ob der Inhalt oder eine Referenz zurückgeliefert werden soll.

B.12 WildFlow::SetComponent

Beschreibung	Klasse zur Verwaltung von mengenwertigen Komponenten. Hierbei handelt es sich um eine Komponente, die wiederum eine Menge von Einzelkomponenten der Klasse WildFlow::Component aufnehmen kann.
Oberklassen	WildFlow::Component
Modul	WildFlow/Component.pm

Klassenmethoden:

WildFlow::Component->new(`OBJECT=>$obj, DESCRIPTION=>$desc`)

Konstruktormethode zur Erzeugung einer neuen Komponenteninstanz vom Typ WildFlow::Component. Die Parameter werden in Form von Attribut-Wert Paaren übergeben.

Parameter:

Name	Typ	Beschreibung
`$obj`	`ContainerObject`	Gibt das Containerobjekt an, innerhalb dem die Komponente abgelegt werden soll. Dies ist notwendig, damit der Speicher für die Instanz in der richtigen Datenbank allokiert werden kann. Der eigentliche Eintrag in das Containerobjekt erfolgt erst anschließend.
`$desc`	`STRING`	Beschreibung für diese Komponente.

Instanzenmethoden:

new()

Im Gegensatz zur Klassenmethode, die eine neue Instanz einer mengenwertigen Komponente erzeugt, wird bei der Instanzenmethode new() ein neues Komponentenobjekt in die Instanz der Mengenkomponente eingetragen.

Die Methode liefert die erzeugte Instanz zurück. .

add(`$comp`)

Diese Methode trägt eine Instanz der Klasse WildFlow::Component in die Menge ein.

Parameter:

Name	Typ	Beschreibung
`$comp`	`WildFlow::Component`	Instanz der Klasse WildFlow::Component

get_set()

Liefert eine Referenz auf ein Array mit allen Komponenteninstanzen dieser Instanz zurück.

is_set()

Liefert TRUE, falls es sich um die Mengeninstanz handelt. Wenn die Methode TRUE liefert, dürfen die Methoden set_content(...) und get_content(...) nicht angewandt werden.

delete()

Diese Methode kann nur auf Instanzen angewandt werden, die einen einzelnen Eintrag repräsentieren. Wird die Methode auf die Zentralinstanz angewandt, wird eine Fehlermeldung ausgelöst. Sollen alle sich im Containerobjekt befindlichen Komponenten gelöscht werden, so kann die Methode clear() benutzt werden.

clear()

Löscht alle Komponenteninstanzen innerhalb der Mengenkomponente.

B.13 WildFlow::VersionComponent

Beschreibung	Repräsentiert ein versioniertes Objekt. Noch nicht implementiert !!!
Oberklassen	WildFlow::Component
Modul	WildFlow/Component.pm

Danksagung

Die vorliegende Arbeit entstand in den Jahren 1996-1999 am Institut für Angewandte Informatik (IAI) des Forschungszentrums Karlsruhe im Rahmen eines Doktorandenstipendiums.

Für die Übernahme des Referates und die strukturellen Hinweise zum Aufbau der Arbeit möchte ich Herrn Prof. Dr. Georg Bretthauer, vom Institut für Angewandte Informatik, herzlich danken. Mein Dank richtet sich auch an Herrn Prof. Dr. Hartwig Steusloff, vom Fraunhofer Institut für Informations- und Datenverarbeitung (IITB) in Karlsruhe, für die Übernahme des Korreferats. Herrn Dr. Horst Eggert, Leiter der Abteilung Mikrosysteminformatik des IAI, danke ich für die Integration der Arbeit in das F&E-Programm der Abteilung und die zahlreichen Ratschläge.

Dr. Clemens Düpmeier, Leiter der Gruppe Systemintegration der Abteilung Mikrosysteminformatik des IAI, möchte ich herzlichst für die fachliche Betreuung der Arbeit danken. Viele Gespräche und Diskussionen trugen in einer sehr angenehmen Atmosphäre zum Gelingen der Arbeit bei.
Weiterhin gilt mein Dank den Mitarbeitern dieser Abteilung für das angenehme Umfeld und die ständige Hilfsbereitschaft bei Fragen und Problemen.