Universität Leipzig

Fakultät für Mathematik und Informatik

Konzeption und Implementierung eines Workflow-Editors

Diplomarbeit

Aufgabenstellung und Betreuung:

Prof. Dr. habil. E. Rahm

Dipl.-Math. R. Müller

vorgelegt von:

Rainer Böhme

Leipzig, Juli 2000

Vorwort

Inhalt des Projekts *HematoWork* ist die computerbasierte Unterstützung von medizinischen Arbeitsabläufen. Von diesem Ausgangspunkt her wird im Projekt *AgentWork* ein auch für andere Anwendungsgebiete einsetzbares Workflow-Management-System erstellt.

Im Rahmen dieser Diplomarbeit wird ein Workflow-Editor entwickelt, der eine Komponente dieses Workflow-Management-Systems ist.

Ich möchte folgenden Personen danken:

- Herrn Prof. Dr. E. Rahm für die Aufgabenstellung und Unterstützung der Arbeit,
- Herrn Prof. Dr. A. Winter für die hilfreichen Informationen,
- Herrn Dipl.-Math. R. Müller für die umfassende Betreuung der Arbeit,
- meinen Kommilitonen Ulrike Greiner und Alexander Dietzsch für die gute Zusammenarbeit.

Inhaltsverzeichnis

1 Einleitung

Workflow-Management-Systeme sind derzeit ein wichtiges Gebiet der Forschung und Entwicklung im Bereich der Informatik. Der grundlegende Gedanke dabei ist, Arbeitsabläufe in Organisationen durch den Einsatz von Computern zu unterstützen, um einerseits den Arbeitsaufwand zu verringern und andererseits eine höhere Qualität der Ergebnisse zu erreichen.

Ausgangspunkt für den Einsatz von Workflow-Management-Systemen ist die Analyse von bestehenden Arbeitsabläufen (Geschäftsprozeßmodellierung), die zunächst eher informelle Beschreibungen von Arbeitsabläufen liefert. Durch die Verfeinerung dieser Beschreibungen und ihre Darstellung in einem Formalismus wird es möglich, diese als Grundlage zur computerbasierten Unterstützung von Arbeitsabläufen zu verwenden.

Die vorliegende Arbeit beschäftigt sich mit der Konzeption und Implementierung eines Workflow-Editors, also mit dem Werkzeug, mit dessen Hilfe die formalen Beschreibungen von Arbeitsabläufen erstellt werden können und mit den formalen Konstrukten, welche die Grundlage der Beschreibungen liefern.

1.1 Kontext der Arbeit

Ziel des Projekts *HematoWork* ist es, die Arbeitsabläufe bei der Durchführung von Therapiestudien in der Hämato-Onkologie durch den Einsatz von Workflow-Management-Systemen zu unterstützen. Ein solcher Arbeitsablauf kann z.B. die Behandlung eines Patienten sein.

Ein Besonderheit dieses Anwendungsbereiches ist, daß es durch bestimmte Ereignisse während der Ausführung eines Arbeitsablaufes nötig werden kann, diesen dynamisch an die veränderte Situation anzupassen.

In Abbildung 1 ist beispielsweise ein Ausschnitt aus einem Arbeitsablauf dargestellt, der die Behandlung eines Patienten unterstützt. Während des Behandlungsablaufes treten die

Ereignisse "Peripheral Polyneuropathy" und "Leukopenia" ein, die eine veränderte Behandlung des Patienten erfordern. Konkret darf das Medikament Vincristin nicht länger verarbreicht werden, entsprechend wird die Aktivität "Administer Vincristin" entfernt. Außerdem muß ein Antibiotikum zusätzlich verabreicht werden, dazu wird die Aktivität "Administer Antibiotics" eingefügt.

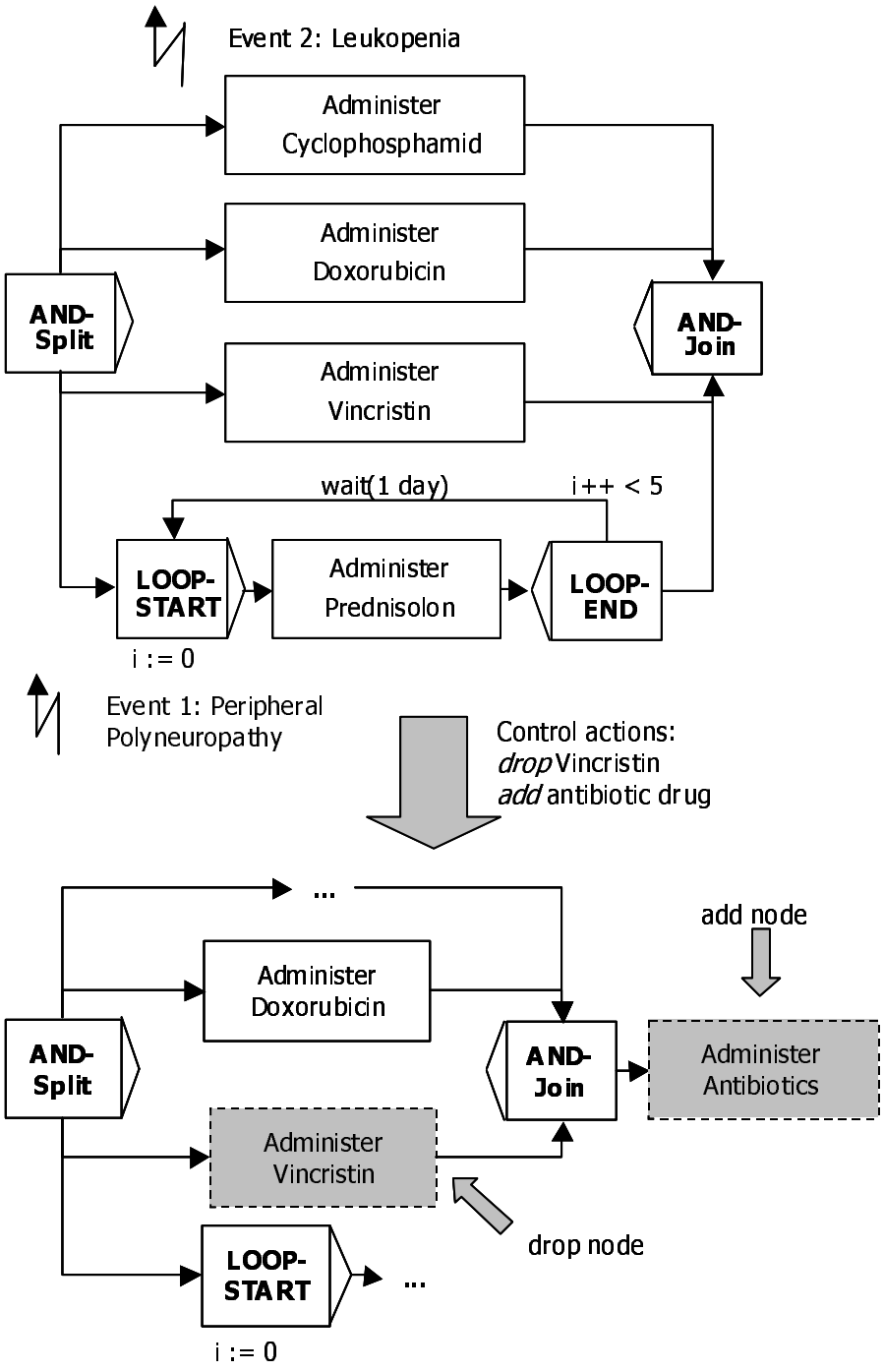


Abbildung 1: Beispiel einer dynamischen Adaptation (aus Projektbestand)

Existierende Workflow-Management-Systeme unterstützen diese dynamischen Anpassungen eines Workflows während seiner Abarbeitung nur unzureichend. Daraus entstand der Gedanke, ein Workflow-Management-System zu erstellen, das diese Möglichkeiten bietet.

Da ein Einsatz eines solchen Workflow-Management-Systems auch für andere Anwendungsgebiete vorstellbar ist, wurde die Erstellung dieses Systems in das separate Projekt *AgentWork* ausgelagert. Die vorliegende Arbeit ist primär im Kontext dieses Projekts entstanden, wobei Anwendungsfälle aus dem *HematoWork*-Projekt als Grundlage für die Konzeption dienten. Dabei kann davon ausgegangen werden, daß diese Anwendungsfälle eine relativ hohe Komplexität aufweisen, so daß auch alle wesentlichen Aspekte anderer Anwendungsbereiche damit abgedeckt werden.

1.2 Zielsetzung der Arbeit

Im Rahmen der dynamischen Adaptationen werden Änderungen an den formalen Beschreibungen der Arbeitsabläufe durchgeführt. Diese Anpassungen sollen weitgehend automatisch ausgeführt werden. Dazu müssen die formalen Arbeitsablaufbeschreibungen einerseits ausreichende Informationen über die Semantik der Aktivitäten enthalten und andererseits eine relativ einfache Struktur aufweisen, um die automatische Analyse und Änderung durchführbar zu machen.

Ein wesentlicher Bestandteil der Arbeit ist daher zunächst die Ausarbeitung der formalen Konstrukte und möglichen Zusammensetzungen, aus denen Arbeitsablaufbeschreibungen bestehen. Die Erarbeitung erfolgt gemeinsam mit den anderen Projektmitarbeitern, um diesbezüglich einen einheitlichen Ausgangsstand zu erreichen und die Berücksichtigung aller speziellen Anforderungen der einzelnen Teile des Projekts zu gewährleisten.

Darauf aufbauend wird die Konzeption und Implementierung des Workflow-Editors erstellt. Der Workflow-Editor soll die Definition der formalen Arbeitsablaufbeschreibungen ermöglichen, wobei diese grafisch dargestellt werden. Nach Abschluß des Definitionsvorgangs sollen die Arbeitsablaufbeschreibungen korrekt sein, das heißt, sie müssen aus den vorgegebenen formalen Konstrukten korrekt zusammengesetzt sein.

1.3 Gliederung

Im Kapitel 2 werden zunächst grundlegende Begriffe und Definitionen aus dem Workflow-Bereich vorgestellt und Funktionen von Workflow-Management-Systemen und ihre Basiskonzepte erläutert.

Kapitel 3 beinhaltet die Analyse der speziellen Anforderungen des medizinischen Anwendungsbereichs und die konzeptionellen Aspekte des *AgentWork*-Systems, die zur Erfüllung dieser Anforderungen beitragen.

Kapitel 4 behandelt die formalen Konstrukte des Workflow-Modells, aus denen die Beschreibungen der Arbeitsabläufe zusammengesetzt werden können. Die Darstellung ist dabei auf die Sichtweise des Anwenders ausgerichtet.

Im Kapitel 5 wird vorgestellt, wie die formalen Konstrukte des Workflow-Modells im UML (Unified Modeling Language)-Buildtime-Modell objektorientiert modelliert werden.

Kapitel 6 beinhaltet dann wesentliche Anforderungen an den Workflow-Editor, seine Konzeption und Aspekte seiner Implementierung.

Abschließend wird in Kapitel 7 eine Zusammenfassung der Erkenntnisse aus der Arbeit und ein Ausblick gegeben.

2 Grundlagen zu Workflow-Systemen

Verschiedene Gruppen beschäftigen sich mit Standardisierungen auf dem Gebiet des Workflow-Managements.

Dazu zählt unter anderem die *Business Object Domain Task Force* der *Object Management Group* (OMG), die in [OMG00] eine *Workflow Management Facility* beschreibt. In ihr werden Schnittstellen festgelegt, die die Interoperabilität verschiedener Workflow-Systeme ermöglichen sollen.

Die *Workflow Management Coalition* (WfMC) ist eine weitere Gruppe von Firmen, die im Bereich des Workflow-Managements tätig sind, sie veröffentlichte das *Workflow Reference Model* [WRM95]. Da in diesem verstärkt Aspekte der Technik und Architektur von Workflow-Systemen behandelt werden, soll es in den folgenden Abschnitten als Basis für die Darstellung dieser Aspekte dienen.

Das Gebiet des Workflow-Managements ist einerseits noch relativ jung und hat andererseits auch einen übergreifenden und vielschichtigen Charakter. Deshalb existiert eine Vielzahl von Begriffen und Definitionen, die teilweise aus angrenzenden Bereichen wie der Unternehmensgestaltung und Organisationsentwicklung stammen. Sie sind dort mit bestimmten Bedeutungen belegt, wie in [JBS97] ausgeführt, werden aber teilweise auch synonym verwendet. In den folgenden Abschnitten sollen deshalb zunächst auch einige Grundbegriffe definiert werden.

2.1 Basisarchitektur von Workflow-Systemen

2.1.1 Workflow

Definition 1:

> *Workflow* ist die computerbasierte Unterstützung oder Automatisierung eines *Arbeitsablaufes*, im Gesamten oder in Teilen.

Häufig wird statt des Begriffes *Arbeitsablauf* der Begriff *Geschäftsprozeß* verwendet, der den engen Zusammenhang mit dem Gebiet der Geschäftsprozessmodellierung verdeutlicht. Allerdings erscheint der Begriff *Arbeitsablauf* im gegebenen medizinischen Kontext als passender.

2.1.2 Workflow-Management-System

Definition 2:

> Ein *Workflow-Management-System* (kurz: Workflow-System) ist ein System, welches Workflows definiert, verwaltet und durch die Ausführung von Software abarbeitet, wobei die Reihenfolge der Ausführung der Software durch eine computerbasierte Repräsentation der Arbeitsablauf-Logik gesteuert wird.

Dabei kann es je nach Anwendungsbereich große Unterschiede zwischen den Arbeitsabläufen geben, wie unterschiedliche Komplexität, Ausführungsdauer, dynamische Änderungen des Ablaufs u.a.m. Diese Unterschiede schlagen sich auch in der Architektur der Workflow-Management-Systeme nieder. Allerdings gibt es auch gemeinsame Merkmale. Diese werden in der folgenden Grafik dargestellt, wobei folgende drei Arten von Bestandteilen unterschieden werden können:

- Software-Komponenten, die verschiedene Funktionen des Workflow-Systems realisieren
- System-Kontroll-Daten und -Definitions-Daten, die von den Software-Komponenten verwendet werden
- Externe Produkte, wie Applikationen und Datenbanken mit Anwendungsdaten, die nicht Teil des Workflow-Systems sind, aber durch dieses gesteuert werden

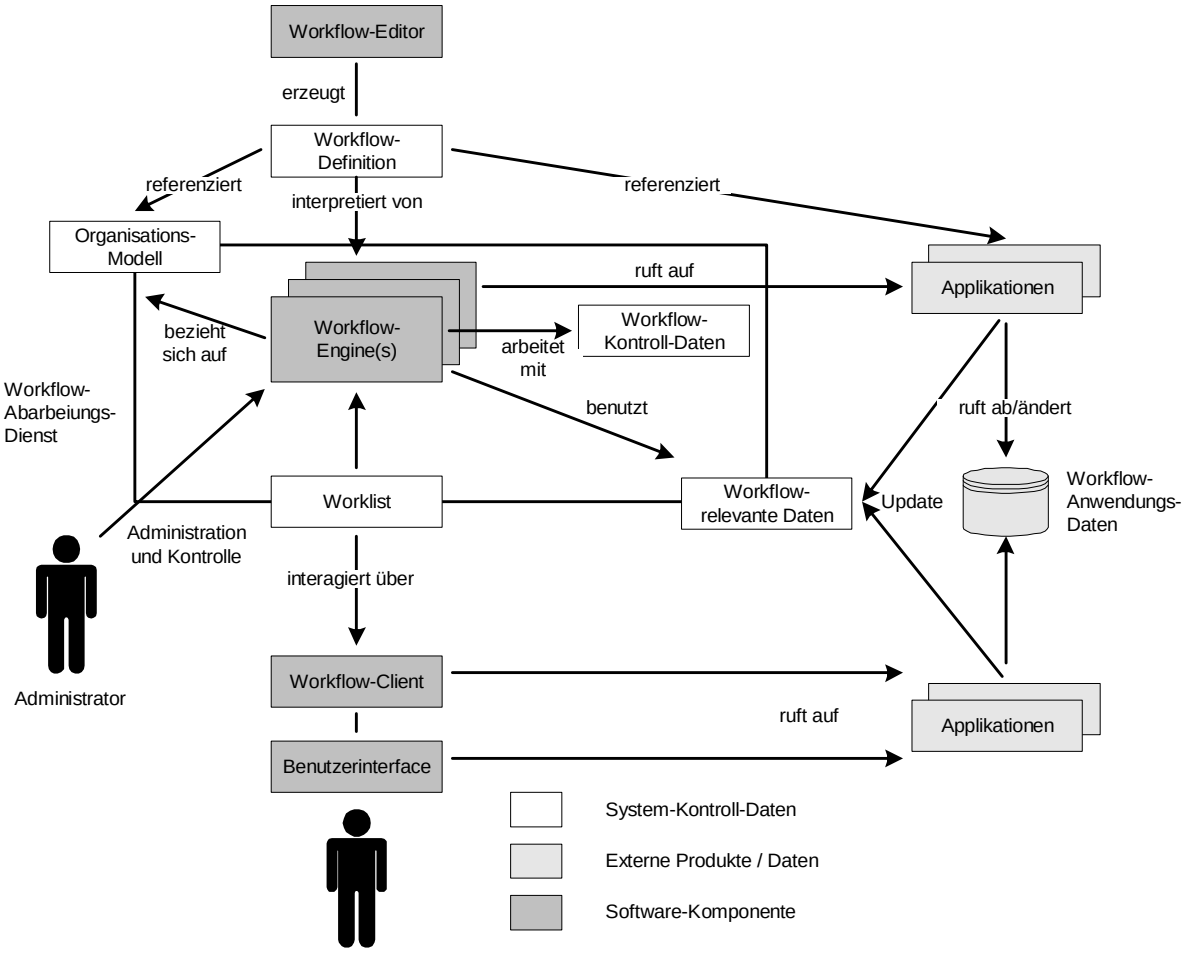


Abbildung 2: Basisarchitektur von Workflow-Systemen (nach [WRM95])

In den folgenden Abschnitten sollen einige der Komponenten in ihre Umgebung eingeordnet und erläutert werden.

2.1.3 Buildtime

In der Buildtime-Umgebung des Workflow-Systems wird die automatisierte Abarbeitung der Arbeitsabläufe vorbereitet.

Ein zentraler Aspekt ist dabei die Transformation eines realen Arbeitsablaufs in eine computerbasierte Darstellung, die *Workflow-Definition* genannt wird.

Definition 3:

> Eine *Workflow-Definition* enthält eine formale, vom Computer abarbeitbare Darstellung der Beschreibung einer Anzahl von einzelnen Aktivitäten mit den zugehörigen automatischen und/oder manuellen Operationen, sowie Regeln, die den Ablauf der Abarbeitung der einzelnen Aktivitäten steuern.

Definition 4:

> Der *Workflow-Editor* beinhaltet die Funktionen zur Erstellung von *Workflow-Definitionen*. Er ist die zentrale funktionale Komponente der Buildtime-Umgebung.

In der Workflow-Definition werden Applikationen referenziert, die zur Ausführung von Aktivitäten benötigt werden. Außerdem existieren in ihr Referenzen auf das Organisationsmodell, in dem Daten über die Personen und ihre Rollen in der Organisation enthalten sind.

2.1.4 Runtime

Die Funktionalität der Runtime bildet die Verbindung zwischen der modellierten Workflow-Definition und dem real ausgeführten Arbeitsablauf.

Definition 5:

> Die *Workflow-Engine* führt die Workflow-Definitionen aus, in dem sie *Instanzen* von diesen erzeugt, den Inhalt der Workflow-Definitionen interpretiert und entsprechend der Beschreibung der Aktivitäten bestimmte Aktionen durchführt.

Die Workflow-Engine ist damit die zentrale Komponente der Runtime-Umgebung.

Die einzelnen Aktivitäten in einem Workflow sind meist mit Benutzer-Interaktionen verbunden, oft in Zusammenhang mit der Verwendung einer bestimmten Applikation, z. B. zum Ausfüllen eines Formulars. Eine solche Applikation kann vom Workflow-System gestartet und gesteuert werden

Zwei Spezialfälle sind dabei

- rein manuelle Aktivitäten, bei denen der Benutzer lediglich darüber informiert wird, daß eine Aktivität auszuführen ist, die jedoch nicht vom Computer unterstützt wird, wie z.B. das Verabreichen eines Medikaments
- rein automatische Aktivitäten, welche die Ausführung bestimmter Applikationen ohne Benutzer-Interaktion beinhalten.

Workflow-relevante Daten sind solche, die durch Applikationen verwendet und geändert werden können. Die Workflow-Engine benutzt sie, um anhand bestimmter Bedingungsprüfungen auf diesen Daten Entscheidungen über den weiteren Ablauf der Workflow-Instanz zu treffen.

Außerdem arbeitet die Workflow-Engine mit Workflow-Kontroll-Daten, die insbesondere Informationen über den aktuellen Status der Abarbeitung von Workflow-Instanzen beinhalten.

2.1.4.1 Verteilung und Systemschnittstellen

Die Fähigkeit, Aufgaben und Informationen in einer IT-Umgebung zu verteilen, ist eine wesentliche Eigenschaft der Workflow-Runtime-Infrastruktur. Die Verteilungsfunktion kann dabei auf verschiedenen Ebenen (Arbeitsgruppe oder Organisation) arbeiten und unterschiedliche Kommunikationsmechanismen (z.B. E-Mail, Message-Passing, verteilte Objekte) benutzen. Die folgende Grafik stellt diesen Verteilungsaspekt dar.

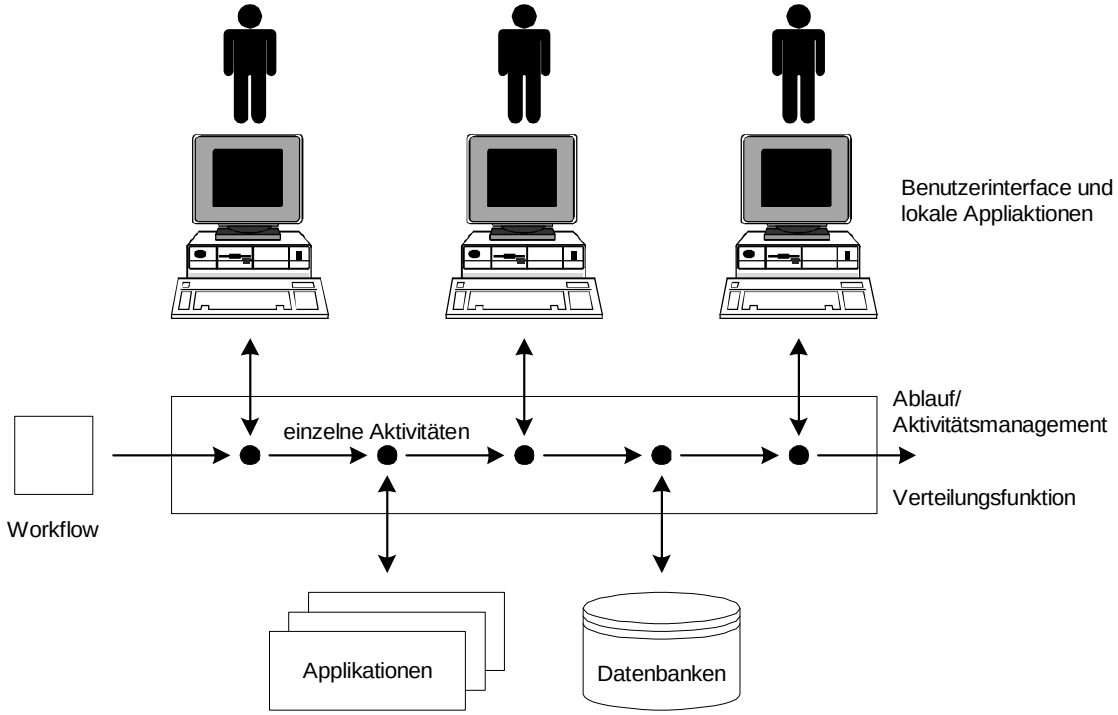


Abbildung 3: Verteilung in der Runtime-Umgebung (nach [WRM95])

2.2 Anwendungsgebiete

Die Anwendungsgebiete von Workflow-Systemen sind vielfältig. Entsprechend unterschiedlich sind auch die Workflow-relevanten Daten, die in Workflow-Systemen verarbeitet werden müssen. Exemplarisch sollen hier unstrukturierte Daten, wie sie in Bilddateien vorliegen, und stark strukturierte Daten, wie in elektronischen Dokumenten, vorgestellt werden.

2.2.1 Bildverarbeitung

Häufig werden Dokumente, die in Papier- oder ähnlicher Form vorliegen, in Bilddateien umgewandelt, da sich diese leichter transportieren, archivieren und abrufen lassen. Die elektronische Form eines Dokuments kann auch wesentlich einfacher in einem Workflow-System verarbeitet werden. Verschiedene Bildverarbeitungsysteme besitzen Schnittstellen, um sie mit Workflow-Systemen zu koppeln.

Die Inhalte der Bilddateien sind dabei vom Workflow-System im Allgemeinen nicht erfaßbar. Sie können demzufolge nicht zur Steuerung des Workflows anhand bestimmter Bedingungen verwendet werden. Erst wenn durch Auswertungen auf den Bilddateien strukturierte Daten entstanden sind, ist dies möglich.

Im medizinischen Bereich treten z.B. Röntgenbilder oder Computertomogramme in Form von Bilddaten auf.

2.2.2 Dokumenten-Management

Dokumentenmanagement-Technologie wird zur Verwaltung von elektronischen Dokumenten über ihren gesamten Lebenszyklus eingesetzt.

Dokumenten-Repositories beinhalten Dokumente, stellen sie den Nutzern zur Verfügung und bieten Möglichkeiten zur Weiterleitung von Dokumenten zwischen den Benutzern, zur Information über deren Inhalt oder Aktualisierung des Inhalts. Damit kann das Dokument Bestandteil eines Arbeitsablaufes in einer Organisation sein.

Im medizinischen Kontext treten z.B. Studienbögen als eine Form von Dokumenten auf. In [Jöd97] wird beispielsweise beschrieben, wie solche stark strukturierten Dokumente in elektronischer Form abgelegt werden können.

Daten dieser Art können im Workflow-System direkt zur Steuerung des Ablaufs anhand bestimmter Bedingungen verwendet werden.

2.3 Abgrenzung zu verwandten Gebieten

Das Gebiet des Workflow-Managements ist breit angelegt und besitzt Überlappungen mit verschiedenen anderen Gebieten. Einige wesentliche Gebiete sollen hier kurz vorgestellt werden.

2.3.1 Geschäftsprozeßmodellierung

Im Rahmen der Geschäftsprozeßmodellierung werden Arbeitsabläufe in einer Organisation analysiert, modelliert und (neu) definiert. Dies beinhaltet die Analyse der Struktur von Arbeitsabläufen und den zugehörigen Informationsfluß, Rollen von einzelnen Personen oder Gruppen im Arbeitsablauf und die Aktionen, die in Reaktion auf bestimmte Ereignisse ausgeführt werden.

Häufig sind die Ergebnisse der Geschäftsprozeßmodellierung Ausgangspunkt für den Einsatz eines Workflow-Management-Systems.

2.3.2 Groupware-Applikationen

Im Bereich der Groupware-Anwendungen existiert eine Vielzahl von Produkten, welche die Interaktionen zwischen Gruppen von Benutzern unterstützen und verbessern sollen. Ihren Ursprung hatten sie in einfachen Funktionen, wie eine elektronische Form von "Schwarzen Brettern" und Gruppen-Terminplanern.

Um eine stärkere Strukturierung der Zusammenarbeit der Benutzer zu erreichen, wurden Workflow-Funktionalitäten in die Produkte integriert. Ihr Schwerpunkt liegt jedoch nach wie vor auf eher schwach strukturierten Arbeitsabläufen.

2.3.3 Transaktionsbasierte Applikationen

Transaktionsbasierte Applikationen existieren bereits seit vielen Jahren, sie wurden häufig mittels TP-Monitoren und/oder Datenbanksystemen implementiert. Vom zunächst zentralisierten Ansatz der Transaktionen ausgehend wurden die Mechanismen weiterentwickelt, so daß auch Transaktionen, die verschiedene verteilte Ressourcen betreffen, ausgeführt werden können.

Mittels Transaktionen ist es möglich, eine höhere Robustheit gegenüber Fehlern verschiedenster Arten zu erreichen. Diese Eigenschaft ist auch in Bezug auf Workflow-Systeme interessant, allerdings können im Workflow-Bereich komplexere Transaktionsmodelle erforderlich sein, um die komplexen Abläufe zu unterstützen.

2.4 Workflow-Modelle

Wie bereits erwähnt, enthält eine Workflow-Definition eine formale, vom Computer abarbeitbare Darstellung eines Arbeitsablaufes. Dies setzt voraus, daß ein Formalismus festgelegt wird, in dem die Workflow-Definitionen formuliert werden, um die Abarbeitung durch einen Computer zu ermöglichen.

Dieser Formalismus wird auch Workflow-Sprache, Workflow-Sprachmodell oder Workflow-Modell genannt.

Wie in [JBS97] S. 77 ff dargestellt, ist dieses Workflow-Modell ein wesentliches Unterscheidungsmerkmal zwischen verschiedenen Workflow-Systemen. Seine Ausdrucksmächtigkeit bestimmt wesentlich die Anwendbarkeit eines Workflow-Systems auf eine gegebene Problemstellung.

2.4.1 Ansätze für Workflow-Modelle

Für die Entwicklung eines Workflow-Modells können verschiedene Konzepte und Theorien zugrunde gelegt werden. Zwei Beispiele dafür sind Petri-Netze und Transaktionskonzepte.

2.4.1.1 Petri-Netze

Petri-Netze werden zur Beschreibung des Verhaltens von Prozessen und Systemen eingesetzt. Die ersten Workflow-Systeme entstanden auf dieser Basis.

Ein Vorteil von Petri-Netzen ist, daß auf ihrer theoretischen Grundlage verschiedene Prüfungen der Korrektheit der modellierten Prozesse (z.B. auf Erreichbarkeit, Verklemmungsfreiheit, Lebendigkeit) durchgeführt werden können, wie u.a. in [AAH98] dargestellt.

Problematisch ist hingegen, daß keine explizite Trennung von Datenfluß (Weitergabe von Daten zwischen den Aktivitäten) und Kontrollfluß (Zustandsübergang zwischen den Aktivitäten) erfolgt.

2.4.1.2 State- und Activity-Charts

State- und Activity-Charts basieren auf einem Formalismus nach [Har87]. Sie bilden zwei verschiedene Sichtweisen auf eine Workflow-Definition.

Activity-Charts sind die hierarchische Zerlegung eines Systems in Fumtkionseinheiten (Activities). Sie spezifizieren den Datenfluß zwischen Aktivitäten in Form eines gerichteten Graphen, wobei die Datenelemente als Beschriftung an den Kanten angegeben sind.

Zu jedem Activity-Chart existiert ein State-Chart, der dessen Verhalten beschreibt. In ihm ist der Kontrollfluß zwischen Aktivitäten in Form von Regeln für Zustandsübergänge enthalten. Jedem Zustandsübergang wird ein Ereignis, eine Bedingung und eine Aktivität zugeordnet. Ist das Ereignis eingetreten und die Bedingung erfüllt, dann wird die Aktivität ausgeführt und der Folgezustand wird erreicht.

Vorteilhaft ist hier die Trennung von Datenfluß und Kontrollfluß und der modulare Aufbau von Activity-Charts, der Gruppierungen zu größeren Funktionseinheiten zuläßt.

2.4.2 Workflow-Basismodell

Ein grundlegendes, häufig anzutreffendes Konzept eines Workflow-Modells soll hier als das *Workflow-Basismodell* vorgestellt werden, es basiert auf dem *Workflow Reference Model* der WfMC.

2.4.2.1 Kontrollfluß

Oft werden Workflow-Modelle in einem gerichteten Graphen repräsentiert. Die Knoten in dem Graphen stellen Aktivitäten dar, die Kanten repräsentieren die Übergänge von einer Aktivität zur nächsten und werden auch als *Transitionen* bezeichnet.

Transitionen können mit *Bedingungen* versehen sein. Diese ermöglichen es, den Start der folgenden Aktivität von bestimmten *zeitlichen Bedingungen* oder Bedingungen auf bestimmten Daten (*wertebasierten Bedingungen*) abhängig zu machen.

Diese Bestandteile einer Workflow-Definition heißen *Kontrollfluß*, da sie die kontrollierte Abfolge der Aktivitäten spezifizieren.

Gegebenenfalls können spezielle *Kontrollflußknoten* existieren, die eine zusätzliche Steuerung des Ablaufs ermöglichen, wie z.B. die wiederholte, bedingte oder parallele Ausführung bestimmter Kontrollflußabschnitte. Durch die Verwendung von Kontrollflußknoten wird die Modellierung der Kontrollflußlogik explizit gemacht, woraus sich nach [JBS97] eine bessere Überprüfbarkeit und damit eine höhere Schemaqualität ergibt.

2.4.2.2 Datenfluß

Für die Ausführung einzelner Aktivitäten werden meist Eingabedaten benötigt, die diese verarbeiten und ggf. daraus resultierende Ausgabedaten bereitstellen. Diese Ausgabedaten können wiederum als Eingabedaten einer späteren Aktivität dienen, welche jedoch nicht unbedingt die im Kontrollfluß direkt folgende Aktivität sein muß. Dieser Zusammenhang von Ausgabe- und Eingabedaten verschiedener Aktivitäten wird als Datenfluß bezeichnet und grafisch durch eine zweite Art von Kanten dargestellt, *Datenflußkanten* genannt. Der Datenfluß zwischen Aktivitäten wird als *interner Datenfluß* bezeichnet.

Außerdem können am Datenfluß externe Datenquellen wie z.B. Datenbanken beteiligt sein, die Ausgangsdaten für Aktivitäten bereitstellen bzw. in denen Endergebnisse zur späteren Verwendung abgelegt werden. Dieser Datenfluß wird entsprechend *externer Datenfluß* genannt.

2.5 Begriffe der Workflow-Modellierung

Zur Systematisierung und Ergänzung einiger Begriffe der Workflow-Modellierung soll für die Zwecke der Diplomarbeit die Abbildung 4 dienen.

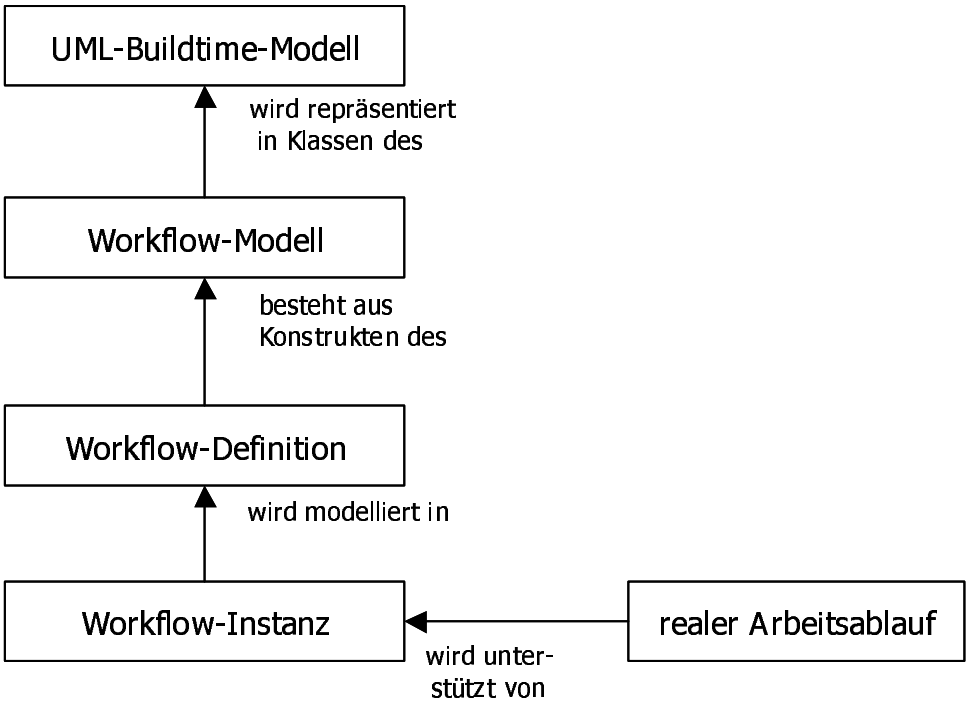


Abbildung 4: Begriffe der Workflow-Modellierung

Eine *Workflow-Instanz* dient zur Unterstützung von realen Arbeitsabläufen.

Die *Workflow-Definition* enthält eine Beschreibung der Aktivitäten, die in einer Workflow-Instanz ausgeführt werden. Sie besteht aus verschiedenen formalen Konstrukten, die im *Workflow-Modell* definiert sind.

Im Projekt *AgentWork* wird im *UML-Buildtime-Modell* eine objektorientierte Sicht auf die Konstrukte des Workflow-Modells dargestellt. Es dient als Grundlage für die Implementierung. Eine konkrete Workflow-Definition existiert dann in Form von Objektinstanzen der Klassen des UML-Buildtime-Modells. UML (Unified Modeling Language) ist ein Standard der OMG zur objektorientierten Modellierung [UML99]. Sie wurde verwendet, um eine kompakte und übersichtliche Darstellung des Objektmodells zu erreichen.

Das Workflow-Modell wird in Abschnitt 3.3 einführend dargestellt und in Kapitel 4 werden die enthaltenen Konstrukte ausführlich erläutert

Daran anschließend wird in Kapitel 5 das UML-Buildtime-Modell vorgestellt.

3 AgentWork-System

In Kapitel 2 wurde das Hintergrundwissen zu Workflow-Systemen dargestellt. Im folgenden soll nun zunächst der medizinische Anwendungsbereich geschildert werden. Daraus werden die Anforderungen abgeleitet, die sich aus diesem Anwendungsbereich ergeben.

3.1 Medizinischer Anwendungsbereich

3.1.1 Hämato-Onkologie

Die Hämato-Onkologie befasst sich mit der Behandlung von nicht-soliden Tumoren, wie Lymphomen oder Leukämien. Da die Erkrankung an einem solchen Tumor oft zum Tode des Patienten führt, werden auf diesem Gebiet große Forschungsanstrengungen unternommen, um die Behandlung zu verbessern.

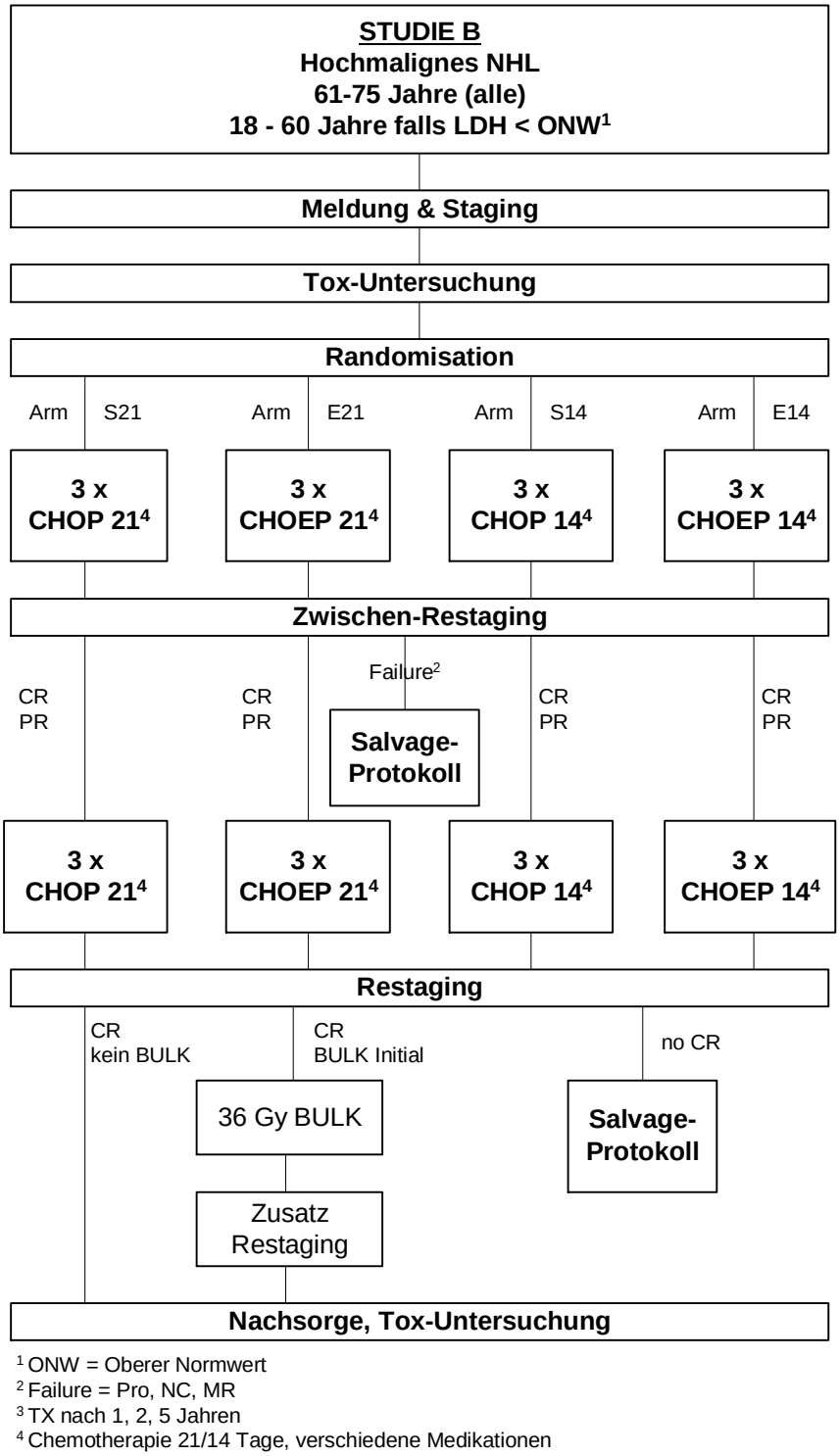
Ein wesentlicher Teil bei diesen Forschungen sind klinische Studien zur Erprobung der Wirksamkeit neuer Therapien. Um die Qualität der Studienergebnisse sicherzustellen, werden dabei standardisierte Behandlungspläne eingesetzt, wie in Abbildung 5 dargestellt.

In diesen Behandlungsplänen sind verschiedene Aspekte des Ablaufs einer Therapie enthalten, sowohl was organisatorische Aktivitäten, wie die Aufnahme in eine Studie betrifft, als auch durchzuführende Untersuchungen, wie die Tox-Untersuchung und durchzuführende Behandlungen, wie Chemotherapien. Diese Aktivitäten werden in ihrer zeitlichen Reihenfolge festgelegt. So ist in Abbildung 5 der Ablauf einer Behandlung beginnend mit der Prüfung der Vorbedingungen über Meldung & Staging, verschiedene weitere Untersuchungs- und Behandlungsschritte bis hin zur Nachsorge und Tox-Untersuchung dargestellt.

Für die einzelnen Aktivitäten existieren dann wiederum detaillierte Pläne, wie diese auszuführen sind.

Die durchzuführenden Arbeitsabläufe sind somit stark strukturiert und der Einsatz eines Workflow-Systems bietet sich an, um die Handhabung der umfangreichen Datenmengen,

die im Rahmen der Studien entstehen, zu erleichtern, die Qualität der Ergebnisse zu verbessern und eine zeitnahe Auswertung zu ermöglichen.



[1] ONW = Oberer Normwert
[2] Failure = Pro, NC, MR
[3] TX nach 1, 2, 5 Jahren
[4] Chemotherapie 21/14 Tage, verschiedene Medikationen

Abbildung 5: Hochmalignes NHL, Behandlungsplan Studie B (nach [LP94])

3.1.2 Spezielle Anforderungen im medizinischen Kontext

Wenn die Arbeitsabläufe in dem medizinischen Kontext auch zunächst gut strukturiert sind, so sind aber auch eine große Anzahl von Ausnahmefällen gegeben, wie z.B. allergische Reaktionen eines Patienten auf ein bestimmtes Medikament, die in den Arbeitsabläufen berücksichtigt werden müssen, in dem die Aktivität zur Verabreichung der Medikamente entfernt wird und gegebenenfalls alternative Aktivitäten eingefügt werden, wie dies bereits in Abbildung 1 dargestellt wurde.

Diese Ausnahmefälle sollen nun als *logische Fehler* bezeichnet werden, da sie im Sinne einer existierenden Workflow-Definition einen Fehler in der Workflow-Umgebung darstellen, der zunächst nicht berücksichtigt ist. Die Bezeichnung *logischer Fehler* dient auch zur Unterscheidung von *technischen Fehlern*, wie dem Ausfall eines Datenbanksystems. Die Behandlung von technischen Fehlern kann beispielsweise durch *erweiterte Transaktionskonzepte* erfolgen, wie sie in [HR99] dargestellt sind. Diese sollen im Rahmen der Arbeit nicht näher diskutiert werden.

Aufgrund der Vielzahl möglicher logischer Fehler ist es nicht sinnvoll, in jeder Workflow-Definition alle denkbaren logischen Fehler bereits von vornherein zu modellieren, soweit dies überhaupt möglich ist. Vielmehr muß es auch während der Abarbeitung einer Workflow-Definition möglich sein, diese dynamisch an geänderte Bedingungen anzupassen. Dazu müssen Aktivitäten hinzugefügt, entfernt oder durch alternative Aktivitäten ersetzt werden. Dies wird *Adaptation* genannt.

Da das Auftreten von logischen Fehlern relativ häufig sein kann, ist es wünschenswert, daß die Adaptationen der Workflow-Definitionen weitgehend automatisch ausgeführt werden, wobei nur zur Bestätigung der Änderungen und in Zweifelsfällen der Eingriff eines Benutzers nötig sein soll.

Wichtig für die Durchführung der Adaptation ist die Unterbrechbarkeit der Workflow-Abarbeitung. Dazu muß das Workflow-System geeignete Mechanismen bereitstellen, um die Abarbeitung einer Workflow-Instanz kontrolliert anzuhalten und später fortzusetzen. Dies ist bei den bisher verfügbaren Systemen aber nicht in ausreichendem Maß gegeben. Somit erscheint die Erstellung eines Workflow-Systems nötig, das diese Funktionalität unterstützt. Das Gesamtkonzept des Workflow-Systems wird in [Mül00] dargestellt, im folgenden soll dies kurz vorgestellt werden.

3.2 Architektur des Workflow-Systems

Die oben angeführten Adaptationen von Workflow-Definitionen machen es einerseits nötig, die in Abschnitt 2.1 vorgestellte Basisarchitektur von Workflow-Systemen um Komponenten zu erweitern, die entsprechende Funktionen bereitstellen, aber andererseits auch die bereits vorgestellten Komponenten um zusätzliche Funktionen zu ergänzen.

Die Architektur des *AgentWork*-Systems ist in Abbildung 6 dargestellt und wird im folgenden erläutert.

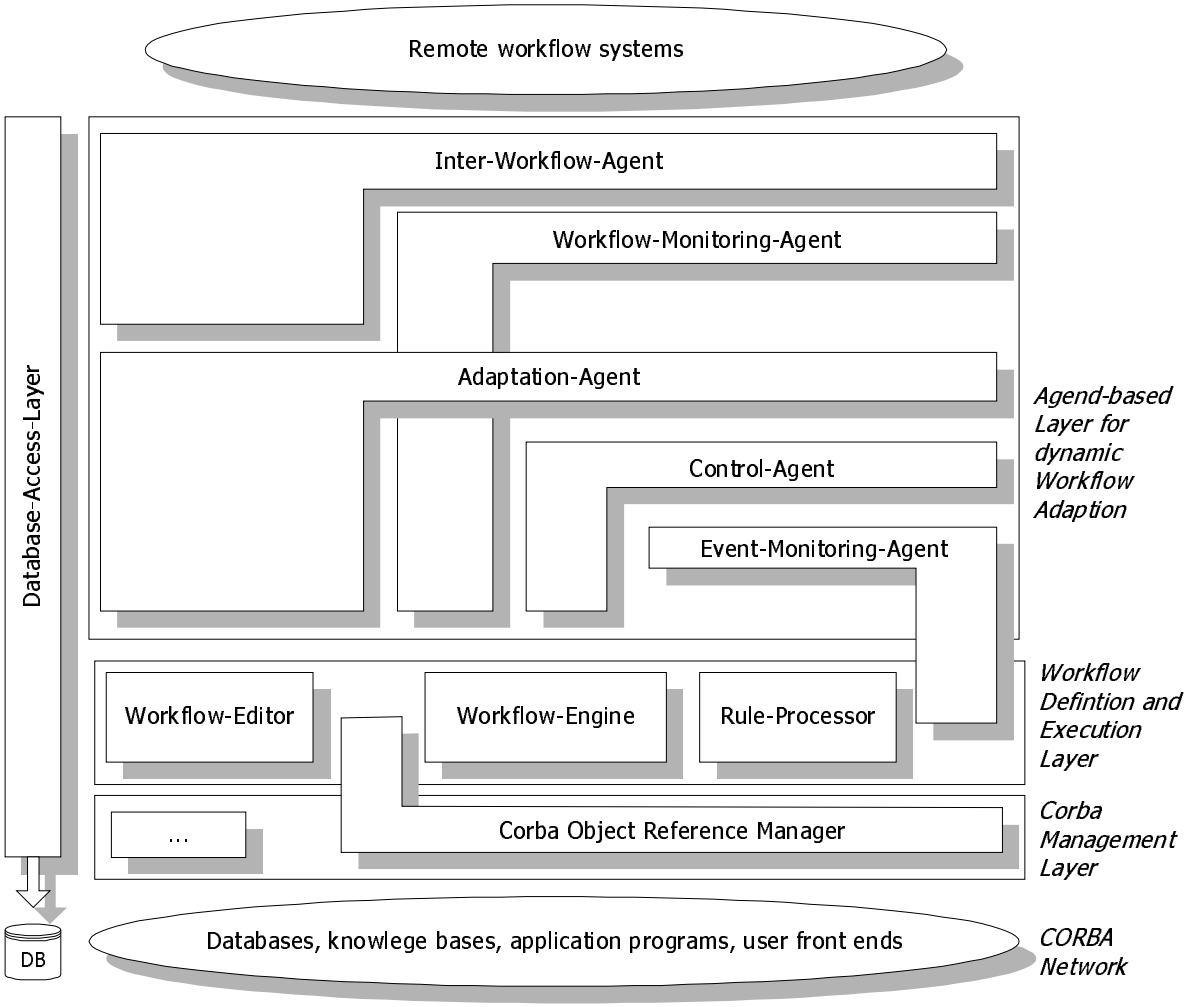


Abbildung 6:Architektur des AgentWork-Systems (aus Projektbestand)

3.2.1 Workflow-Editor

Im Workflow-Editor werden die Workflow-Definitionen erzeugt. Diese werden in einer Datenbank abgelegt. Über die *Database-Access-Layer* können sowohl der Workflow-Editor als auch andere Komponenten des Systems auf sie zugreifen.

Der Workflow-Editor wird in Kapitel 6 ausführlich beschrieben.

3.2.2 Workflow-Engine

Die Workflow-Engine instanziiert die Workflow-Definitionen und führt die Workflow-Instanzen aus.

Für die Adaptation von Workflow-Instanzen während der Laufzeit ist es nötig, daß die Abarbeitung einer Workflow-Instanz angehalten werden kann, um Anpassungen durch den Adaptation-Agenten zu ermöglichen und die Abarbeitung der geänderten Workflow-Instanz anschließend an dem Punkt fortgesetzt werden kann, an dem sie angehalten wurde. Diese Eigenschaft der *Unterbrechbarkeit* ist ein wesentliches Merkmal der Workflow-Engine, welches sie von den meisten existierenden Workflow-Engines unterscheidet.

Die Workflow-Engine wird in [Die00] beschrieben.

3.2.3 Event-Monitoring-Agent

Der Event-Monitoring-Agent überwacht auftretende Ereignisse und prüft anhand einer Regelbasis mit Hilfe des *Rule-Processors*, ob diese einen logischen Fehler in einer Workflow-Instanz verursachen. Wenn dies der Fall ist, dann wird eine entsprechende Meldung an den Control-Agenten weitergegeben.

3.2.4 Control-Agent

Der Control-Agent prüft bei Auftreten eines logischen Fehlers, welche laufenden Workflow-Instanzen und welche Aktivitäten darin von diesem betroffen sind. Diese Information wird verwendet, um entsprechende Unterbrechungen der Workflow-Instanzen in der Workflow-Engine zu veranlassen und die an der Workflow-Definition durchzuführenden Änderungen an den Adaptation-Agenten zu melden.

3.2.5 Adaptation-Agent

Der Adaptation-Agent ändert die Workflow-Definition, die Bestandteil der Workflow-Instanz ist, entsprechend den in der Regelbasis gegebenen Regeln ab. Da jede Workflow-Instanz eine eigene Kopie der Workflow-Definition besitzt, sind diese Änderungen jeweils nur für diese Workflow-Instanz gültig.

Der Adaptation-Agent wird in [Grei00] ausführlich dargestellt. Hier soll nur der auch für das Workflow-Modell relevante Aspekt der Berechnung von *Adaptationsregionen* kurz vorgestellt werden.

Eine Regel für die Änderung einer Workflow-Definition kann z.B. besagen, daß eine bestimmte Aktivität für eine bestimmte Zeitdauer nicht durchgeführt werden darf, dann aber wieder auszuführen ist.

Wenn diese Änderungsregel auf eine Workflow-Definition angewendet werden soll, muß dazu bekannt sein, zu welchem Zeitpunkt eine bestimmte Aktion durchgeführt wird, um festzustellen, ob diese Aktivität entfernt werden soll oder nicht. Dies ist problematisch, da i. A. nicht bekannt ist, wie lange Aktivitäten dauern, und somit auch nicht vorhergesagt werden kann, wann eine in der Zukunft liegende Aktivität ausgeführt wird.

Die *Adaptationsregion* bezeichnet den Abschnitt in einer Workflow-Definition, der in dem Zeitraum liegt, für den eine Änderungsregel gilt. Die Berechnung der Adaptationsregion wird im Adaptation-Agenten durchgeführt und basiert auf geschätzten Werten für die Dauer von Aktivitäten, die in der Workflow-Definition angegeben werden. Die berechnete Adaptationsregion ist somit nur ein Abschätzung.

3.2.6 Workflow-Monitoring-Agent

Wird eine Workflow-Instanz nach einer Änderung fortgesetzt, dann wird dies oft dazu führen, daß die tatsächlichen Dauern von Aktivitäten von den geschätzten Werten abweichen, was auch dazu führt, daß die geschätzte Adaptationsregion nicht mit der tatsächlich nötigen übereinstimmt. Der Workflow-Monitoring-Agent überwacht das Auftreten dieser Abweichungen und löst gegebenenfalls eine erneute Änderung einer Workflow-Instanz durch den Adaptations-Agenten aus.

3.2.7 CORBA Management Layer

CORBA (Common Object Request Broker Architecture) ist ein Standard der OMG (Object Management Group) [OMG99]. Er beinhaltet die Interaktion von Objekten in einer verteilten Umgebung.

Darauf aufbauend realisiert der CORBA Management Layer (CML) des *AgentWork*-Systems drei wesentliche Funktionen im Workflow-System:

- Unterstützung des Verteilungsaspektes in der Runtime-Infrastruktur, wie in Abschnitt 2.1.4.1 dargestellt
- Anbieten einheitlicher Sichten auf Objekte in der Workflow-Umgebung in Form von CORBA-Interfaces, dies betrifft insbesondere
  - Schnittstellen zur Ansteuerung der Applikationen
  - Schnittstellen zur Abfrage und Modifikation auf heterogenen externen Datenquellen mit Kapselung der Daten in CORBA-Objekten
- Unterstützung von Transaktionen der Workflow-Engine durch den CORBA-Transaktions-Service (s. [OMG98])

Insbesondere die Bereitstellung der Funktionalität für den Zugriff auf allgemein heterogene externe Datenquellen mit Abfrage- und Modifikationsmöglichkeiten beinhaltet eine erhebliche Komplexität. Allerdings muß davon ausgegangen werden, daß im allgemeinen Fall eine solche heterogene Umgebung vorliegt, in der das Workflow-System zum Einsatz kommen soll.

Im speziellen medizinischen Kontext wird jedoch das System *Medical Control Center (MCC®)* der Meierhofer AG zum Einsatz kommen, welches die Daten - wie in [Mei00] aufgeführt - in einem relationalen Datenbanksystem (MS SQL Server bzw. Oracle) ablegt. Es kann davon ausgegangen werden, daß alle Workflow-relevanten Daten in diesem System abgelegt sind. Somit reduziert sich die Komplexität der CORBA Management Layer in diesem Fall etwas, da auf alle Daten über eine einheitliche relationale Schnittstelle (z.B. ODBC) zugegriffen werden kann.

3.3 Anforderungen an das Workflow-Modell

Die Adaptation der Workflow-Definitionen zur Runtime bedingen auch einige Anforderungen an das Workflow-Modell. Dem entsprechend muß das in Abschnitt 2.4.2 vorgestellte Workflow-Basismodell erweitert bzw. konkret ausgestaltet werden.

Im folgenden sollen die Anforderungen an das Workflow-Modell und seine entsprechende Gestaltung im Überblick vorgestellt werden. Die ausführliche Beschreibung der Konstrukte des Workflow-Modells ist in Kapitel 4 zu finden.

3.3.1 Basisanforderungen

Die automatische Adaptation der Workflow-Definitionen erfordert, daß diese eine möglichst klare und einfache Struktur besitzen.

Aus diesem Grund erscheinen Petri-Netze für diesen Zweck weniger geeignet, da sich der Dualismus zwischen Stellen und Transitionen und die fehlende Trennung von Kontroll- und Datenfluß negativ auswirkt. Wie in [RD98] ausgeführt, eignen sich auch andere allgemeine Workflow-Modelle weniger für dynamische Adaptationen, da die Analyse komplexerer Workflow-Definitionen zur Runtime bei diesen zu aufwendig ist.

Dort wird außerdem erläutert, daß diese Modelle für Nicht-Modellexperten wenig intuitiv sind, was aber wünschenswert ist, da nicht bei jeder Adaptation ein Experte hinzugezogen werden kann, um die geänderte Workflow-Definition zu überprüfen.

In [RD98] wird deshalb das $Adept_{flex}$-Modell vorgestellt, welches an das Modell der WfMC angelehnt ist. Darin sind insbesondere symmetrische Kontrollstrukturen enthalten, welche aus der strukturierten Programmierung bekannt ist. Wesentliche Bestandteile sind:

- Sequenzen von Aktivitäten
- Verzweigungen
- Schleifen

Diese sind alle als symmetrische Blöcke definiert, mit jeweils wohldefinierten Anfangs- und Ende-Knoten - , welche Kontrollflußknoten sind.

Die symmetrischen Blöcke dürfen dabei beliebig ineinander verschachtelt sein, dürfen sich jedoch nicht überlappen.

Durch diese einfachen Strukturen wird die dynamische Adaptation zur Runtime ermöglicht, sie werden deshalb auch im Workflow-Modell des *AgentWork*-Projekts verwendet.

Die einzelnen Kontrollflußkonstrukte werden in Abschnitt 4.1.3 ausführlich vorgestellt.

3.3.2 Spezifische Anforderungen

Um Aktivitäten automatisch entfernen, ersetzen oder hinzufügen zu können, müssen diese eindeutig identifizierbar sein. Im Workflow-Modell der WfMC ist eine Aktivität im wesentlichen nur durch ihren Namen gekennzeichnet. Dies erweist sich nur dann als ausreichend, wenn einfache Änderungen, wie "Entferne Medikation ASS" erfolgen sollen.

Allerdings treten auch komplexere Änderungsregeln auf, beispielsweise "Entferne Medikation ASS mit Dosis > 100 mg". Diese sind nur über den Namen der Aktivität nicht auswertbar.

Bei den Aktivitäten muß deshalb eine genauere Information über ihre Semantik vorliegen. In [Mül00] werden dazu *feingranuläre Aktivitätsdefinitionen* eingeführt, die diese Semantik in Form von Aktivitätsnamen und zusätzlichen Parametern enthalten.

3.3.3 Wiederverwendung

Die Erstellung von Workflow-Definitionen für einen gegebenen organisatorischen Rahmen ist ein aufwendiger Prozeß. der üblicherweise mehrere Stufen beinhaltet, u.a.:

- *umgangssprachliche Formulierung* eines Arbeitsablaufs
- *grobe Formalisierung*, Identifikation von Aktivitäten, Kontrollfluß und Datenfluß
- *feinere Ausarbeitung*, Zuordnung von Applikationen zu Aktivitäten

Wünschenswert ist es daher, Workflow-Definitionen zwischen verschiedenen Organisationen auszutauschen und wiederzuverwenden. In [JBS97] wird dieser Ansatz als *Branchenschemata* bezeichnet. [VB96] beschreibt einen analogen Ansatz, wobei die Nutzung eines gemeinsamen Organisationsmodells (Unternehmensmodells) als wesentliche Voraussetzung angesehen wird.

In beiden Ansätzen wird davon ausgegangen, daß bestimmte Arbeitsabläufe in verschiedenen Organisationen einer Branche zumindest auf einer konzeptionellen Ebene, wie der Stufe der *groben Formalisierung*, sehr ähnlich sind. Im gegebenen medizinischen Kontext erscheint die Voraussetzung auf Grund der standardisierten Behandlungspläne zumindest grundlegend erfüllt zu sein.

Im *AgentWork*-Projekt soll deshalb auch der Ansatz verfolgt werden, die konzeptionelle Ebene eines Arbeitsablaufs, der aus Aktivitäten besteht, von den technischen Aspekten der Implementierung mittels verschiedener Applikationen zu trennen.

Konkret wird dies dadurch unterstützt, daß einer Aktivitätsdefinition verschiedene Applikationen zugeordnet werden können, wobei diese Zuordnung in *verschiedenen* Organisationen unterschiedlich durchgeführt werden kann. Damit werden Workflow-Definitionen zwischen den verschiedenen Organisationen wiederverwendbar. Im folgenden Abschnitt

wird u.a. noch eine weitere Art der Wiederverwendung aufgeführt, die innerhalb *einer* Organisation stattfindet.

3.3.4 Modularisierung von Workflow-Definitionen

Die Behandlungspläne der Hämato-Onkologie sind, wie bereits erwähnt, modular aufgebaut. Zu einem Behandlungsplan existieren mehrere Pläne, welche die einzelnen enthaltenen Aktionen detailliert beschreiben.

Die Möglichkeit zur Modularisierung von Workflow-Definitionen in mehrere Teile bietet, wie in [JBS97] ausgeführt, verschiedene Vorteile:

- *Übersichtlichkeit:* komplexe Arbeitsabläufe mit vielen Teilaufgaben können übersichtlicher dargestellt werden, wenn sich mehrere Aktivitäten zu einer Aktivität (mit komplexer Binnenstruktur) zusammenfassen lassen
- *Wiederverwendbarkeit:* durch die Zusammenfassung von mehreren Aktivitäten zu einer komplexen Aktivität kann diese in verschiedenen Workflow-Definitionen verwendet werden, ohne jeweils die gesamte Binnenstruktur wiederholen zu müssen
- *Änderungsfähigkeit:* wird eine komplexe Aktivität in verschiedenen Workflow-Definitionen wiederverwendet, so müssen Änderungen an ihrer Binnenstruktur nur einmal durchgeführt werden

Im *AgentWork*-System wird die Modularisierung von Workflow-Definitionen durch *Sub-Workflow-Definitionen* bzw. *komplexe Aktivitätsdefinitionen* unterstützt.

Eine Sub-Workflow-Definition ist dabei nahezu identisch zu einer normalen Workflow-Definition, sie beinhaltet eine Anzahl von Aktivitäten. Die gesamte *Sub-Workflow-Definition* kann dann als eine Aktivität in einer übergeordneten Workflow-Definition verwendet werden. Dort tritt sie somit als *komplexe Aktivitätsdefinition* auf.

Eine Sub-Workflow-Definition kann ihrerseits auch wieder komplexe Aktivitätsdefinitionen enthalten, so daß eine mehrfache Verschachtelung möglich ist.

3.4 Realisierungskonzept

3.4.1 Verwendung von F-Logic

Die Regeln zur Adaptation von Workflow-Definitionen werden in einer Regelbasis abgelegt. Sie werden in einer formalen Sprache formuliert, welche die Auswertung der Regeln durch einen entsprechenden Regelprozessor ermöglicht.

Als formale Sprache für die Regeln wurde in [Mül00] F-Logic ausgewählt, die in [KLW95] vorgestellt wird. Diese Sprache vereint deduktive Aspekte mit einem objektorientierten Datenmodell. Die Vereinigung der beiden Paradigmen in einer Sprache wird als vielversprechend angesehen, da sie sich gut ergänzen.

3.4.2 Aktivitätsklassen

Um mittels der F-Logic-Regeln Adaptationen vornehmen zu können, muß es eine Möglichkeit geben, Aktivitäten in einem Workflow eindeutig zu klassifizieren, wie bereits in 3.3.2 dargestellt. Dazu wird ein System von Klassen in F-Logic entworfen, so daß jede Aktivität durch eine Klasse repräsentiert ist. Diese *Aktivitätsklasse* wird auch in der Workflow-Definition angegeben, um die Aktivität zu kennzeichnen.

Jede Aktivitätsklasse ist (direkt oder indirekt) von der Oberklasse *Activity* abgeleitet.

3.4.3 Datenklassen

Die automatische Adaptation von Workflow-Definitionen hat durch Änderungen der Aktivitäten auch zur Folge, daß der Datenfluß in der Workflow-Definition angepaßt werden muß. Um dies zu automatisieren, muß eine konzeptionelle Sichtweise auf die Daten und den Datenfluß gegeben sein. Dazu werden alle Daten in Objekten gekapselt, der Objektfluß wird als Zuweisungen auf diesen Objekten dargestellt. Als Sprache zur Modellierung der Objekte und des Objektflusses wird F-Logic verwendet.

Dazu werden in F-Logic auch die Klassen der Datenobjekte, also die *Datenklassen*, deklariert. Der Zugriff auf die Daten zur Runtime erfolgt über CORBA-Objekte, welche durch den CORBA Management Layer bereitgestellt werden. Dazu existiert zu jeder F-Logic-Datenklasse eine entsprechende CORBA-Klasse mit einer zugehörigen Factory, die in der

Lage ist, dynamisch Objekte der Klasse zu erzeugen. Dies kann mit Hilfe des Dynamic Invocation Interfaces (DII) von CORBA realisiert werden (s. [OMG99]).

3.4.4 Bedingungen und Queries

Bedingungen werden Transitionen zugeordnet, um zu erreichen, daß Aktivitäten in Abhängigkeit von bestimmten Objektwerten abgearbeitet werden.

Queries werden verwendet, um Daten aus externen Datenquellen abzurufen bzw. in diesen zu speichern.

Bedingungen und Queries gehören damit zum Objektfluß im weiteren Sinne. Entsprechend wird für ihre Modellierung ebenfalls F-Logic verwendet.

3.5 Wesentliche F-Logic-Konstrukte

Zur Erleichterung des Verständnisses der F-Logic-Verwendung sollen einige wesentliche Konstrukte der Sprache hier kurz vorgestellt werden. Die ausführliche Beschreibung von F-Logic-Syntax und Semantik ist in [KLW95] zu finden.

3.5.1.1 *is-a*-Aussage

Der Ausdruck

O : class-name

besagt, daß das Objekt mit dem Namen *O* aus der Klasse *class-name* ist.

Beispielsweise wird durch den Ausdruck

I : *imaging-order*

ein Objekt *I* aus der Klasse *imaging-order* bezeichnet.

3.5.1.2 Methoden

Methoden werden in F-Logic in der Form

$O[ScalarMethod@Q_1, ..., Q_k -> T]$

bzw.

$P[SetMethod@R_1, ..., R_l ->> \{S_1, ..., S_m\}]$

dargestellt. Die erste Variante stellt eine skalare Methode *ScalarMethod* des Objekts *O* mit den Parametern $Q_1, ..., Q_k$ und dem Ergebnis *T* dar.

Die zweite Variante beinhaltet eine mengenwertige Methode *SetMethod* des Objekts *P* mit den Parametern $R_1, ..., R_l$ und der Ergebnismenge $\{S_1, ..., S_m\}$.

Eine Besonderheit stellen Methoden ohne Parameter dar. Wie in [KLW95] dargestellt, wird für diese neben der Parameterliste auch das @ weggelassen. Effektiv sind diese dann Attribute des Objekts.

Beispielsweise wird mit dem Ausdruck

ct-examination[cs->Pt, focus->(X->focus)]

ein (implizites) Objekt der Aktivitätsklasse *ct-examination* angegeben, dessen Attribut *cs* das globale Patienten-Objekt *Pt* zugeordnet wird und dessen Attribut *focus* das *focus*-Attribut des Objekts *X* zugewiesen wird. In diesem Beispiel wird der Operator *->* jeweils als Zuweisungsoperator verwendet, dem linksseitigen Objekt *cs* wird das rechtsseitige Objekt *Pt* zugewiesen.

Der Operator *->* hat in F-Logic grundsätzlich verschiedene Bedeutungen. Die Bedeutung ist davon abhängig, welche der Objekte auf den beiden Seiten des Operators gegeben bzw. gesucht sind.

Ist ein Objekt auf einer Seite des Operators nicht gegeben bzw. gesucht, so kann er als *Zuweisungsoperator* betrachtet werden. Sind die Objekte auf beiden Seiten des Operators bereits gegeben, so ist er einem *Vergleichsoperator* gleichzusetzen. Dies ist im deduktiven Charakter von F-Logic begründet.

3.5.1.3 Bedingungen

Bedingungen sind syntaktisch den Methoden sehr ähnlich. Sie unterscheiden sich jedoch hinsichtlich der Auswertung. Der Ausdruck

E[cs->Pt, tumor-residuum->YES]

mit gegebenen Objekten *E* und *Pt* liefert beispielsweise einen Wahrheitswert zurück. Er sagt aus, ob das Objekt *E* der Bedingung genügt, daß *cs* dem globalen Patienten-Objekt *Pt* entspricht und das Attribut *tumor-residuum* den Wert *YES* besitzt. In diesem Fall wird der

Operator *−>* als Vergleichsoperator verwendet, der gesamte Ausdruck gibt einen Wahrheitswert zurück.

3.5.1.4 Queries

Queries zur Abfrage von Daten aus Datenquellen sind den Bedingungen ähnlich. Der Unterschied besteht darin, daß nicht ein gegebenes Objekt auf Erfüllung der Bedingungen geprüft wird, sondern alle Objekte einer gegebenen Klasse ermittelt werden, die der Query-Bedingung genügen.

Eine solche Query ist z.B.

?-M:tumor-marker ^ M[tumors-marked −>> (Pt −> diagnosis)]

Diese ermittelt alle Objekte *M* aus der Klasse *tumor-marker,* bei denen *Pt −> diagnosis* in der Menge *tumors-marked* enthalten ist, die also zur Diagnose des Patienten gehören.

4 Konstrukte des Workflow-Modells

Im Kapitel 3 wurde ein Überblick über das *AgentWork*-System und sein Workflow-Modell gegeben.

Im folgenden sollen nun die einzelnen Konstrukte des konzipierten Workflow-Modells aus Sicht des Anwenders vorgestellt werden.

Die eher modellierungs- und implementierungstechnischen Aspekte der Umsetzung der Konstrukte in ein UML-Modell werden dann im Kapitel 5 behandelt.

4.1 Kontrollfluß

4.1.1 Aktivitätsknoten

Aktivitätsknoten repräsentieren Aktivitäten, die manuell oder automatisch ausgeführt werden.

Jedem Aktivitätsknoten wird eine *Aktivitätsdefinition* zugeordnet, welche die durchzuführende Aktivität näher beschreibt. Zur Verdeutlichung der Zusammenhänge soll hier beispielhaft die Aktivitätsdefinition in Abbildung 7 dienen, eine ausführliche Beschreibung ist in 4.3 zu finden.

Input-Objects	Activity	Output-Objects
X: imaging-report	ct-examination [cs->Pt, focus->(X->focus)]	CT: ct-report

Abbildung 7: Beispiel einer Aktivitätsdefinition

Die dargestellte Aktivitätsdefinition beinhaltet die folgenden wesentlichen Bestandteile:

- Angabe der Eingabeobjekte (*X : imaging-report*)
- feingranuläre Definition der Aktivität (*ct-examination[...]*)
- Angaben über die Ausgabeobjekte (*CT : ct-report*)

- Angaben über auszuführende Applikationen (nicht dargestellt)

Die Zuordnung der Aktivitätsdefinitionen zu einem Aktivitätsknoten ermöglicht damit die Beschreibung der durchzuführenden Aktivität.

Außerdem wird jedem Aktivitätsknoten ein *Verantwortlicher* zugeordnet, der die Aktivität im Falle einer manuellen Aktivität ausführt bzw. bei einer automatischen Aktivität im Fehlerfall benachrichtigt wird (s. 4.4).

4.1.2 Kommunikationsknoten

Kommunikationsknoten werden verwendet, um Daten-Abhängigkeiten zwischen verschiedenen Workflow-Instanzen zu modellieren. Es werden zwei Arten von Kommunikationsknoten unterschieden:

- ausgehende Kommunikation ("Comm-Out"), die ein Ereignis darstellt, auf das andere Workflow-Instanzen warten können
- eingehende Kommunikation ("Comm-In"), die das Warten auf ein Ereignis in einem anderen Workflow definiert

Für jeden Kommunikationsknoten wird eine zeitliche Dauer (*Duration*) festgelegt, deren Durchschnittswert eine Abschätzung der Ausführungsdauer für den Adaptation-Agent angibt und deren Maximalwert eine maximale Wartezeit auf das Zustandekommen der Kommunikation spezifiziert.

Die Modellierung der Kommunikationsknoten ist im Rahmen des Projekts noch nicht vollständig abgeschlossen.

4.1.3 Kontrollknoten

Mittels Kontrollknoten kann der Ablauf des Kontrollflusses gesteuert werden.

4.1.3.1 Start- und End-Knoten

Start- und End-Knoten legen den Anfang und das Ende des Kontrollflusses fest, wie in Abbildung 8 dargestellt. Jede Workflow-Definition enthält je genau einen Start- und End-Knoten.

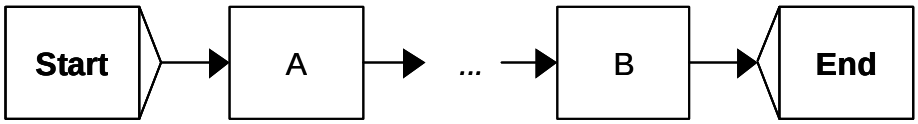


Abbildung 8: Start- und End-Knoten

4.1.3.2 Split- und Join-Knoten (Verzweigungen)

Split-Knoten ermöglichen die Verzweigung des Kontrollflusses in mehrere parallel ablaufende Pfade. Außerdem kann durch die Angabe von Bedingungen an den ausgehenden Transitionen des Split-Knotens festgelegt werden, unter welchen Bedingungen der folgende Pfad abgearbeitet werden soll.

Jeder Split-Knoten hat als Gegenstück genau einen Join-Knoten. Dieser vereinigt die parallel einlaufenden Kontrollpfade zu einem ausgehenden Kontrollpfad.

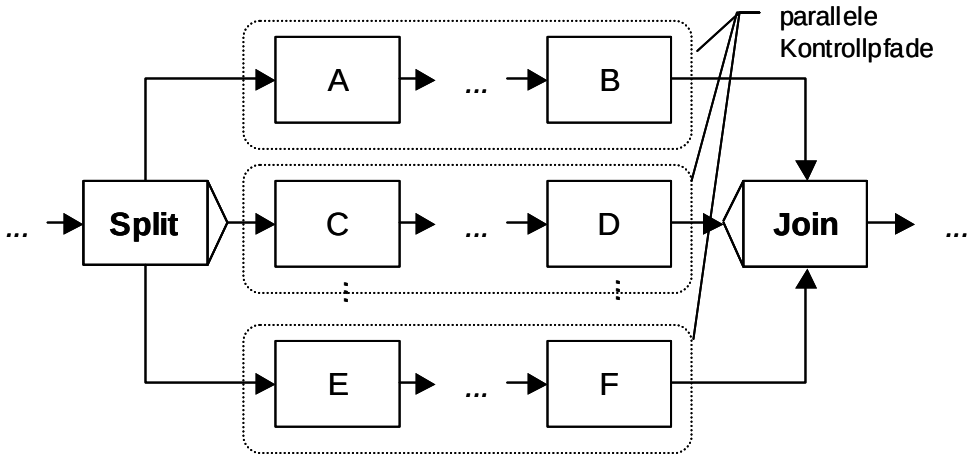


Abbildung 9: Split- und Join-Knoten

Es können zwei Arten von Split-Knoten verwendet werden:

- *And-Split*: Die ausgehenden Transitionen haben keine Bedingungen, es werden also immer alle ausgehenden Kontrollpfade abgearbeitet
- *Or-Split*: Die ausgehenden Transitionen haben (bis auf eine) Bedingungen, sie werden jeweils genau dann aktiviert, wenn die Bedingung erfüllt ist. Genau eine der ausgehenden Transitionen hat keine Bedingung, dies ist die sogenannte *Default-Transition*. Sie wird dann aktiviert, wenn keine der anderen Transitionen aktiviert wird. Damit ist sichergestellt, daß mindestens eine ausgehende Transition des Or-Splits aktiviert wird, es können aber auch mehrere aktiviert werden.

Für Join-Knoten stehen drei verschiedene Typen zur Verfügung:

- *All-Join*: Die ausgehende Transition des Join-Knotens wird aktiviert, wenn alle eingehenden Transitionen aktiviert sind. Wenn der zugehörige Split ein Or-Split ist, dann werden nur die eingehenden Transitionen berücksichtigt, deren Kontrollpfade tatsächlich abgearbeitet werden. Die Verwendung eines All-Joins ist beispielsweise sinnvoll, wenn in den eingehenden Kontrollpfaden Medikationen durchgeführt werden, die auf jeden Fall alle zu Ende geführt werden müssen, bevor die Fortsetzung des Behandlungsablaufes erfolgen kann.
- *One-Join-Complete*: Die ausgehende Transition des Join-Knotens wird aktiviert, wenn eine der eingehenden Transitionen aktiviert wird. Die Kontrollpfade aller anderen eingehenden Transitionen werden weiter abgearbeitet. Eine vorstellbare Verwendung ist die parallele Ausführung verschiedener Untersuchungen, wobei der Arbeitsablauf bereits nach Vorliegen der ersten Untersuchungsergebnisse fortgesetzt werden soll.
- *One-Join-Cancel*: Die ausgehende Transition des Join-Knotens wird aktiviert, wenn eine der eingehenden Transitionen aktiviert wird. Die Abarbeitung der Kontrollpfade aller anderen eingehenden Transitionen wird abgebrochen. Diese Art von Join-Knoten kann beispielsweise verwendet werden, wenn verschiedene Untersuchungen Ergebnisse liefern, die alternativ verwendet werden können. Das heißt, wenn Ergebnisse der zuerst beendeten Untersuchung vorliegen, können die anderen parallel liegenden Untersuchungen abgebrochen werden, da ihre Ergebnisse nicht mehr benötigt werden.

Der zugehörige Join-Knoten zu einem Or-Split-Knoten muß ein All-Join-Knoten sein. Der Grund dafür liegt in der Abschätzung der Adaptationsregionen: Wenn die Ausführungsdauer einer Split-Join-Region vorhergesagt werden soll, dann kann bei Kombination eines Or-Splits mit einem der One-Joins nicht vorhergesagt werden, welche Kontrollpfade überhaupt abgearbeitet werden. Damit kann auch nicht ermittelt werden, welcher zuerst beendet ist und damit zur Beendigung der Region führt und ihre gesamte Ausführungsdauer bestimmt. Die Abschätzung der Adaptationsregion würde demzufolge zu große Ungenauigkeiten beinhalten. Nähere Erläuterungen dazu sind in [Grei00] zu finden.

Bei einem And-Split ergibt sich dieses Problem nicht, es sind demzufolge alle Join-Typen zulässig.

4.1.3.3 Loop-Start und Loop-End-Knoten (Schleifen)

Mittels Loop-Start- und Loop-End-Knoten kann die wiederholte Ausführung bestimmter Abschnitte des Kontrollflusses festgelegt werden.

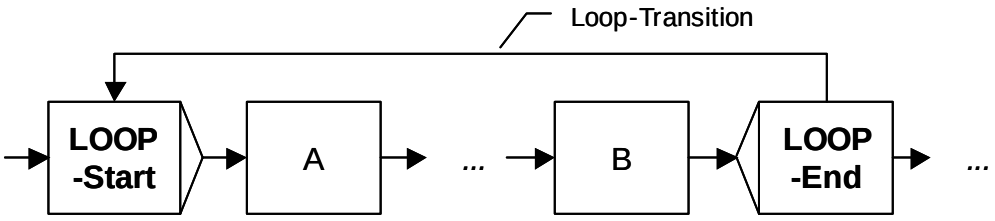


Abbildung 10: Schleife

Vom Loop-End-Knoten führt eine Transition zurück zum Loop-Start-Knoten, diese wird als *Loop-Transition* bezeichnet. Die Loop-Transition besitzt immer eine wertebasierte Bedingung. Solange diese Bedingung wahr ist, wird die Schleife erneut abgearbeitet. Erst wenn diese Bedingung falsch ist, wird die zweite ausgehende Transition des Loop-End-Knotens aktiviert und damit der folgende Kontrollfluß abgearbeitet.

Beim Loop-Start-Knoten wird eine Schätzung darüber angegeben, wie oft die Schleife minimal und maximal durchlaufen wird. Diese Angaben werden vom Adaptation-Agent zur Abschätzung der Adaptations-Region verwendet.

Zu Besonderheiten des Objektflusses bei Schleifen s. 4.2.3

4.1.4 Transitionen

Transitionen verbinden die Knoten und legen fest, in welcher Reihenfolge die Knoten durchlaufen werden.

Es werden zwei Arten von Transitionen unterschieden.

4.1.4.1 Normale Transitionen

Normale Transitionen legen fest, welche Knoten zu einem Kontrollpfad gehören. Ihnen können zwei Arten von Bedingungen zugeordnet werden:

- *temporale Bedingungen*: beinhalten zeitliche Bedingungen, wie eine minimale Wartezeit oder maximale Wartezeit zwischen zwei Aktivitäten
- *wertebasierte Bedingungen*: beinhalten Bedingungen, die sich auf die Werte von Objekten aus dem Objektfluß beziehen. Diese Art von Bedingungen ist ausschließlich bei

den von Or-Split-Knoten ausgehenden Transitionen und bei Loop-Transitionen zulässig

Eine Transition mit einer Bedingung wird nur dann aktiviert, wenn die Bedingung zu wahr evaluiert wird.

Die endgültige Modellierung der Transitions-Bedingungen, insbesondere auch der verschiedenen möglichen Kombinationen von temporalen und wertebasierten Bedingungen, ist im Rahmen des Projekts noch nicht abgeschlossen.

Im Moment wird folgendes Prinzip verwendet:

Wird eine temporale Bedingung allein verwendet, dann wird die *minimale Wartezeit* als Verzögerungszeit zwischen zwei Knoten verwendet. Wird die *maximale Wartezeit* vor Aktivierung des folgenden Knotens überschritten, weil beispielsweise eine manuelle Aktivität noch nicht begonnen wurde, dann erfolgt eine Fehlermeldung.

Wird eine temporale Bedingung zusammen mit einer wertebasierten Bedingung verwendet, so wird ebenfalls mindestens die angegebene *minimale Wartezeit* abgewartet. Dann wird die Abarbeitung fortgesetzt, falls die wertebasierte Bedingung erfüllt ist. Liegen die zur Auswertung der Bedingung nötigen Objekte nicht vor oder wird die folgende Aktivität aus anderen Gründen nicht rechtzeitig aktiviert, so wird nach Ablauf der *maximalen Wartezeit* eine Fehlermeldung ausgelöst.

In Verbindung mit einer wertebasierten Bedingung muß immer eine temporale Bedingung verwendet werden. Die stellt sicher, daß bei Überschreitung der maximalen Wartezeit eine Fehlermeldung ausgelöst wird, falls ein Fehler im Objektfluß vorliegt. Andererseits stellt dies keine Einschränkung dar, da die minimale Wartezeit auf Null gesetzt werden kann.

4.1.4.2 Synchronisations-Transitionen

Synchronisations-Transitionen legen fest, daß parallel liegende Knoten in einer bestimmten Reihenfolge abzuarbeiten sind. Dies ist in Abbildung 11 dargestellt.

Synchronisations-Transitionen sind nur zwischen parallel liegenden Knoten zulässig, d.h. wenn die Reihenfolge der Abarbeitung der Knoten nicht durch normale Transitionen eindeutig definiert ist, wie dies in Split-Join-Regionen der Fall ist. Außerdem darf weder der Quell- noch der Ziel-Knoten ein Start- oder End-Knoten sein und er darf nicht in einer Schleife liegen.

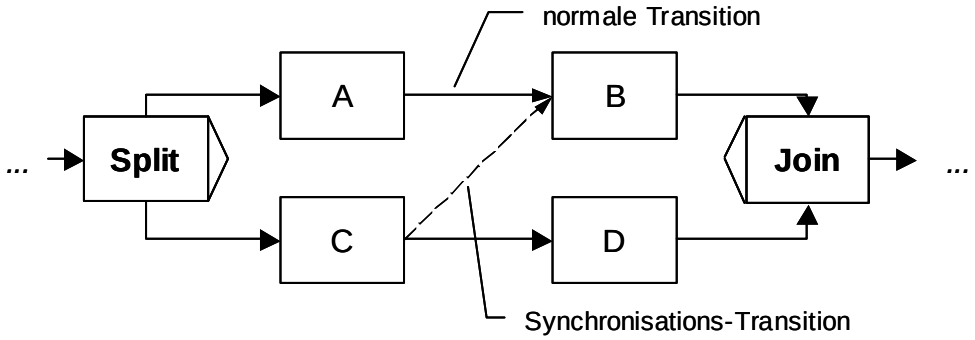


Abbildung 11: Transitionen

4.2 Objektfluß

Der im allgemeinen Workflow-Sprachgebrauch verwendete *Datenfluß* wird hier als *Objektfluß* bezeichnet, um zu verdeutlichen, daß die Modellierung vollständig objektorientiert ist.

Als Basis der Modellierung wird, wie bereits in 3.4.1 erwähnt, F-Logic verwendet.

4.2.1 Objekt-Spezifikationen

Objekt-Spezifikationen werden bei Aktivitätsdefinitionen, Applikationen und Bedingungen von externen Datenzugriffen und Transitionen verwendet. Sie sind im Zusammenhang mit den zugeordneten Objekt-Pfaden die Knoten im Objektfluß-Graphen.

Objekt-Spezifikationen spezifizieren die Namen und Klassen von Objekten, ihre Entsprechung in F-Logic sind einfache ID-Terme.

Die *Namen* der Objekte können frei vergeben werden, z.B. zur Bezeichnung der Semantik eines Objekts im gegebenen Kontext. Die *Klasse* des Objekts entspricht einer *F-Logic-Datenklasse.*

Das F-Logic-Konstrukt *X: imaging-report* ist beispielsweise eine Objekt-Spezifikation, die ein Objekt mit dem Namen *X* und der Klasse *imaging-report* spezifiziert.

Konstanten-Spezifikationen werden als eine besondere Art von Objekt-Spezifikation behandelt, bei denen neben Name und Klasse des Objekts zusätzlich ein Wert angegeben wird. Damit ist die Angabe einfacher konstanter Werte wie z.B. Zeichenketten und (Gleit-

komma-)Zahlen möglich. Wird eine Konstante ohne einen Wert spezifiziert, dann wird dies als Parameter angesehen, für den bei der Erzeugung der Workflow-Instanz ein Wert übergeben werden muß, dies kann z.B. interaktiv durch einen Benutzer erfolgen oder automatisch über entsprechende Aufrufe der Workflow-Engine.

Sowohl Objekt-Spezifikationen als auch Konstanten-Spezifikationen können als *global* innerhalb einer Workflow-Definition definiert werden, sie werden dann sofort bei Erstellung der Workflow-Instanz initialisiert. Bei globalen Konstanten-Spezifikationen werden dazu die gegebenen Werte verwendet. Globale Objekt-Spezifikationen können mittels externer Datenzugriffe initialisiert werden.

4.2.2 Objekt-Pfade

Mit Objekt-Pfaden kann auf einzelne Komponenten eines Objekts (d.h. Attribute, Methoden bzw. Unterobjekte[1]) verwiesen werden. Dies ist auch über mehrere Stufen möglich, so daß z.B. auf ein Attribut eines Unterobjekts verwiesen werden kann.

Ein Objekt-Pfad bezieht sich dabei immer auf eine Objekt-Spezifikation, die den Ausgangspunkt des Pfades darstellt.

In F-Logic sind zwei Varianten für Pfadausdrücke möglich. Als Beispiel soll hier der Zugriff auf das Attribut *street* des Unterobjekts *address* eines Objekts *P* der Klasse *Patient* dienen.

Die ursprüngliche F-Logic-Syntax, wie sie in [KLW95] beschrieben ist, sieht dafür einen Ausdruck der folgenden Form vor:

P: Patient[address->A[street->S]]

Darin wird das Zwischenobjekt *A* als Adresse verwendet, auf dessen Basis das eigentliche Ergebnisobjekt *S* ermittelt wird. Dieser Ausdruck liefert neben dem eigentlich interessierenden Objekt *S* zusätzlich noch das Objekt *A* zurück.

---

[1] In F-Logic werden Unterobjekte bzw. Attribute als Methoden bezeichnet, die keine Parameter haben, vgl. [KLW95]. Aus Gründen der Verständlichkeit sollen hier Methoden, Unterobjekte und Attribute als *Komponenten* bezeichnet werden.

In [MM99] wird eine alternative Syntax angeführt, bei der ein Pfadausdruck jeweils genau ein Objekt liefert, nämlich das, welches am Ende des Ausdrucks steht. Für das Beispiel sieht der Pfadausdruck dann folgendermaßen aus:

(P:Patient).address.street

Die Objekt-Pfade des Workflow-Modells können diese Form repräsentieren. Dazu wird der Ausdruck zerlegt in eine Objekt-Spezifikation *P:Patient* und einen Objekt-Pfad, der sich wiederum aus den beiden Komponenten *address* und *street* zusammensetzt.

Ein Objekt-Pfad kann auch leer sein, d.h. keine Komponenten enthalten, er verweist damit auf das in der Objekt-Spezifikation angegebene Objekt als ganzes, also beispielsweise *P:Patient*.

Den Methoden in einem Objekt-Pfad können auch Parameter übergeben werden. Beispielsweise könnte die Klasse *Patient* eine Methode *administered-drug-amount* beinhalten, welcher der Name eines Medikaments übergeben wird und die dann die Menge zurückgibt, die dem Patienten von diesem Medikament verabreicht wurde.

Sei *D:Drug* die Objekt-Spezifikation für das Medikament, dann hat der gesamte Pfadausdruck die Form

(*P:Patient).administered-drug-amount(D:Drug.name),*

wobei *D:Drug.name* der Pfadausdruck ist, der den Namen des Medikaments liefert, er wird der Methode *administered-drug-amount* als Parameter übergeben.

In Form von Objekt-Pfaden werden außerdem die Bedingungen abgelegt, die bei externen Datenzugriffen und Transitionen spezifiziert werden können (s. 5.4.2a)).

4.2.3 Komponenten-Zuweisungen

Komponenten-Zuweisungen stellen die Kanten im Objektfluß-Graphen dar. Eine Komponenten-Zuweisung verweist auf zwei Objekt-Pfade: einen *Quell-Pfad* und einen *Ziel-Pfad*. Zur Runtime werden die Daten vom Quell-Pfad zum Ziel-Pfad übergeben, sobald das Quell-Objekt verfügbar ist.

Durch die Komponenten-Zuweisungen wird es einerseits ermöglicht, Objekte direkt zu übergeben. Dies erfolgt dadurch, daß Quell-Pfad und Ziel-Pfad leer sind und damit nicht eine Unterkomponente des zugehörigen Objekts referenzieren, sondern das Objekt selbst.

Beispielweise ist in Abbildung 12 eine Komponenten-Zuweisung *CT : X* dargestellt, die das Objekt *CT: ct-report* dem Objekt *X: imaging-report* zuweist.

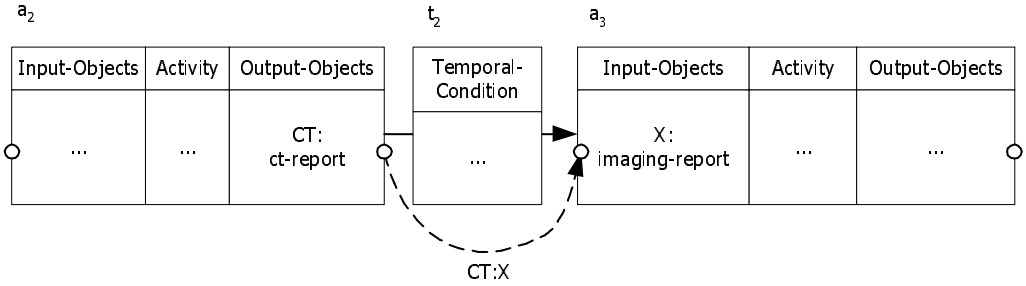


Abbildung 12: Beispiel einer Komponenten-Zuweisung

Es ist aber andererseits auch möglich, ein Objekt aus verschiedenen anderen Objekten bzw. Komponenten zusammenzusetzen, bzw. verschiedene Komponenten eines Objekts an verschiedenen Stellen weiterzuverwenden.

Die Zuweisung kann auf zwei verschiedene Arten erfolgen:

- *Assignment*: Der Ziel-Pfad erhält eine direkte Referenz auf das Objekt des Quell-Pfades
- *Shallow Copy*: Das Objekt des Quell-Pfades wird kopiert, für Unterobjekte werden keine Kopien erzeugt, es werden stattdessen Referenzen auf die ursprünglichen Unterobjekte eingefügt

Bei beiden Arten der Zuweisung müssen die Objekte, die durch Quell- und Zielpfad angegeben sind, typkompatibel sein, d.h. entweder sind ihre Klassen identisch oder die Klasse des Zielobjekts ist eine Oberklasse der Klasse des Quellobjekts.

Nicht unterstützt wird derzeit das *Deep Copy*. Dabei würde das Objekt des Quell-Pfades kopiert, seine Unterobjekte, deren Unterobjekte usw. Problematisch dabei ist, daß zyklische Abhängigkeiten in der Komposition der einzelnen Klassen zu unendlich langen Kopiervorgängen führen würden. Der Grund dafür liegt darin, daß in F-Logic (wie auch in anderen Objektsprachen, z.B. OQL) keine Unterscheidung von Komposition und Assoziation möglich ist. Dieses Problem ließe sich nur durch eine syntaktische Erweiterung von F-Logic beheben.

Beim Einsatz von Komponenten-Zuweisungen ist zu berücksichtigen ist, daß diese wie Synchronisations-Transitionen wirken, da eine Aktivität erst gestartet werden kann, wenn alle benötigten Eingabeobjekte vorliegen und diese wiederum von Ausgabeobjekten ande-

rer Aktivitäten stammen können, die mittels Komponenten-Zuweisungen übergeben werden.

Eine Besonderheit ergibt sich bei der Verwendung von Komponenten-Zuweisungen im Zusammenhang mit Schleifen. Innerhalb von Schleifen sollte es möglich sein, Objekte bei einem Schleifendurchlauf zu ändern und beim folgenden Schleifendurchlauf auf das geänderte Objekt zuzugreifen. Um dies zu ermöglichen, werden rückführende Komponenten-Zuweisungen verwendet.

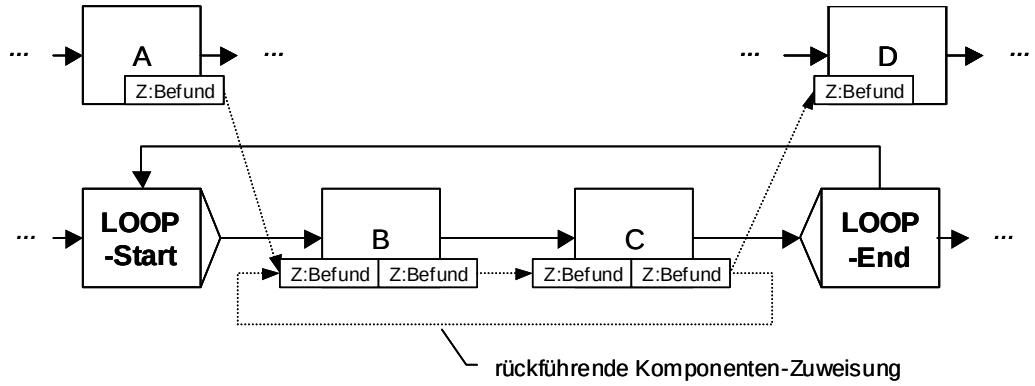


Abbildung 13: Objektfluß in Schleifen

Dabei ist das Eingabeobjekt einer Aktivität zweifach mit Komponenten-Zuweisungen zu versehen. Beim ersten Schleifendurchlauf wird die Komponenten-Zuweisung verwendet, die vom Aktivitätsknoten außerhalb der Schleife in die Schleife hinein führt, bei allen weiteren Schleifendurchläufen wird dagegen die Komponenten-Zuweisung innerhalb der Schleife verwendet.

Analog dazu können die Ausgabeobjekte innerhalb der Schleife doppelt mit Komponenten-Zuweisungen versehen werden: Einmal, um während der Schleifendurchläufe die Rückgabe der Objekte an andere Knoten innerhalb der Schleife zu ermöglichen, und zum anderen, um die Objekte nach dem letzten Schleifendurchlauf an Aktivitätsknoten außerhalb der Schleife weiterzugeben.

4.2.4 Komponenten-Bedingung

Um einfache Vorbedingungen an die Eingabeobjekte einer Aktivitätsdefinition zu stellen, werden Komponenten-Bedingungen verwendet. Diese ermöglichen es, Komponenten zweier Objekte, die über Objekt-Pfade angegeben werden, mittels einfacher Operatoren,

wie <, >, ≤, ≥, -> zu vergleichen. Wenn dieser Vergleich zur Laufzeit nicht zu einem positiven Ergebnis führt, dann wird eine Fehlermeldung generiert.

Der Vergleichsoperator -> ist für alle Klassen von Objekten zugelassen. Alle anderen Vergleichsoperatoren sind nur dann zulässig, wenn sie für die Klassen der zu vergleichenden Objekte definiert sind.

4.2.5 Externe Datenzugriffe

Externe Datenzugriffe ermöglichen es, aus Workflow-Instanzen heraus Daten aus verschiedenen Datenbeständen abzurufen und Änderungen an den Datenbeständen vorzunehmen. Die Daten können sich dabei in verschiedenen Datenbeständen befinden, wie z.B. in Sammlungen von Dateien, relationalen, objektrelationalen und objektorientierten Datenbanksystemen u. a. Ein Ziel des Workflow-Modells ist dabei, eine Sicht auf diese Datenbestände zur Verfügung zu stellen, die von ihrer Heterogenität abstrahiert.

Externe Datenzugriffe treten in zwei Formen auf:

- um bestimmte Objekte aus externen Datensammlungen abzurufen, *Retrieval-Query* genannt
- um neue oder geänderte Objekte in externen Datensammlungen abzuspeichern, bzw. Objekte daraus zu entfernen, *Manipulation-Query* genannt

Im Abschnitt 5.4.2a) werden erste Ansätze für die Modellierung beider Formen im UML-Buildtime-Modell dargestellt.

4.3 Aktivitätsdefinitionen

Aktivitätsdefinitionen werden den Aktivitätsknoten zugeordnet und beinhalten nähere Angaben darüber, welche Aktionen im Rahmen der Aktivität durchzuführen sind. Eine einmal spezifizierte Aktivitätsdefinition kann mehrfach verwendet werden, in dem sie verschiedenen Aktivitätsknoten, auch in verschiedenen Workflow-Definitionen, zugeordnet wird.

Je nach ihrer *inneren Struktur* werden einfache und komplexe Aktivitätsdefinitionen unterschieden.

Dagegen ist ihre *äußere Struktur*, die bei der Verwendung der Aktivitätsdefinition bei einem Aktivitätsknoten sichtbar ist, einheitlich. Sie beinhaltet immer den Namen der Aktivitätsdefinition sowie ihre Ein- und Ausgabeobjekte als *Objekt-Spezifikationen*.

4.3.1 Einfache Aktivitätsdefinitionen

Einfache Aktivitätsdefinitionen beschreiben manuelle oder automatische Aktivitäten.

Jeder Aktivitätsdefinition wird als *Definition* mittels einer Objekt-Spezifikation eine Klasse aus der Menge der F-Logic-Klassen zugeordnet, wobei diese eine Unterklasse der Klasse *Activity* sein muß.

Die F-Logic-Repräsentation der Aktivität wird mittels Objekt-Pfaden und Komponenten-Zuweisungen zwischen den Eingabeobjekten der Aktivitätsdefinition und der *Definition* modelliert.

Input-Objects	Activity	Output-Objects
X: imaging-report	ct-examination [cs->Pt, focus->(X->focus)]	CT: ct-report

Abbildung 14: Beispiel einer einfachen Aktivitätsdefinition

In F-Logic könnte eine Aktivität beispielsweise beschrieben werden durch

$$ct\text{-}examination[cs->Pt,\ focus->(X[focus])]$$

ct-examination ist die F-Logic-Klasse der Aktivität, sie wird in der Objekt-Spezifikation *Definition* abgelegt, mittels zweier Objekt-Pfade und Komponenten-Zuweisungen werden die benötigten Daten übergeben:

- *cs* erhält das globale Patienten-Objekt *Pt*
- *focus* erhält das Unterobjekt *focus* des Eingabeobjekts *X*

Anhand dieser F-Logic-Darstellung kann der Adaptation-Agent die Aktivität identifizieren und entsprechende Modifikationsregeln anwenden. In diesem Beispiel wäre die Anwendung einer Anpassungsregel der Art "Entferne *ct-examination* des Patienten *Bob*, des Bereichs *Head*" denkbar. Im Gegensatz zu anderen Aktivitätsdefinitionen, wie die der WfMC, die nur den Namen einer Aktivität beinhaltet, sind damit wesentlich feingranulärere Definitionen von Aktivitäten möglich.

Aktivitätsdefinitionen können *Applikationen* (s. 4.3.3) zugeordnet werden, die bei Ausführung der Aktivität gestartet werden sollen.

Außerdem wird in ihnen der *Objektfluß* von den Eingabeobjekten der Aktivitätsdefinition zu den Eingabeobjekten der Applikationen und von den Ausgabeobjekten der Applikationen zu den Ausgabeobjekten der Aktivitätsdefinition mittels Komponenten-Zuweisungen spezifiziert.

Für die *Ausführungsdauer* können drei Abschätzungen angegeben werden: die minimale, durchschnittliche und maximale Ausführungsdauer. Dies können Schätzwerte sein oder Meßwerte der tatsächlichen Ausführungsdauer aus der Runtime-Umgebung

Mit Hilfe von *Komponenten-Bedingungen* (s. 4.2.4) können zusätzlich Bedingungen auf den Eingabeobjekten zur Runtime geprüft werden.

4.3.2 Komplexe Aktivitätsdefinitionen, Sub-Workflows

Eine komplexe Aktivitätsdefinition beinhaltet eine komplette (Sub-)*Workflow-Definition*, die ausgeführt wird, wenn der zugehörige Aktivitätsknoten aktiviert wird.

Die Eingabe- bzw. Ausgabeobjekte der Aktivitätsdefinition werden in den Objektfluß der Sub-Workflow-Definition einbezogen, in der aus ihnen Daten abgerufen werden können, bzw. die Ausgabeobjekte mit Daten aufgefüllt werden können.

Nachdem die Sub-Workflow-Definition vollständig abgearbeitet ist, wird auch der zur Aktivitätsdefinition gehörige Aktivitätsknoten als beendet angesehen.

Die Sub-Workflow-Definition kann ihrerseits wieder Aktivitätsknoten mit komplexen Aktivitätsdefinitionen enthalten, so daß eine mehrfache Verschachtelung möglich ist.

4.3.3 Applikationen

Applikationen werden jeweils einer einfachen Aktivitätsdefinition zugeordnet. Diese Applikationen werden zur Runtime dann durch die Workflow-Engine über eine CORBA-Schnittstelle gestartet und ggf. mit ihren Eingabe-Objekten versorgt. Die Workflow-Engine nimmt dann auch die Ausgabeobjekte der Applikationen wieder entgegen. Nach der Beendigung aller Applikationen, die zu einer Aktivität gehören, wird diese als abgeschlossen angesehen.

Eine speziell für die Verwendung mit dem Workflow-System konzipierte Applikation kann die dafür benötigten CORBA-Schnittstellen direkt implementieren, bereits vorhandene Applikationen können über einen CORBA-Wrapper diese Schnittstellen zur Verfügung stellen.

Für jede Applikation wird ein eindeutiger *Name* vergeben, anhand dessen die Workflow-Engine zur Runtime das zu aktivierende CORBA-Objekt identifizieren kann.

Die Ein- und Ausgabeobjekte der Applikation werden mittels *Objekt-Spezifikationen* angegeben. Zur Runtime verwaltet der CORBA-Management-Layer die entsprechenden CORBA-Objekte.

Weiterhin wird für jede Applikation angegeben, ob sie Informationen über ihren *Status* zur Verfügung stellt und ob sie *transaktionsfähig* ist, d.h. an einem 2-Phasen-Commit-Protokoll teilnehmen kann.

Die Eigenschaft *automatic* gibt an, ob die Applikation Interaktionen mit dem Benutzer durchführt, oder vollständig automatisch abläuft.

Schließlich wird jede Applikation entsprechend ihrer Interaktionsmöglichkeiten mit der Workflow-Engine einem von drei *Typen* zugeordnet. Nähere Informationen darüber sind in [Die00] zu finden.

4.4 Organisationsmodell

Das Organisationsmodell ist relativ einfach gehalten. Es soll nur Schnittstellen in Form von Klassen bereitstellen, die in einer konkreten IT-Umgebung verwendet werden können, um z.B. einen bereits vorhandenen Directory Service einzubinden, der dann auch einen entsprechenden Autorisierungsmechanismus bereitstellen kann.

Das Organisationsmodell beruht auf Personen und Rollen. Jedem Aktivitätsknoten wird zur Buildtime eine Rolle zugewiesen, die für ihre Ausführung verantwortlich ist. Zur Laufzeit erfolgt über die Rollen die konkrete Zuordnung von Verantwortlichkeiten für die Aktivitäten zu den Personen.

Dabei kann berücksichtigt werden, daß nicht alle Personen jederzeit verfügbar, d.h. am System angemeldet, sind. So kann z.B. die Rolle "Stationsarzt" zu verschiedenen Tageszeiten durch verschiedene Personen ausgeübt werden.

4.4.1 Rolle

Eine Rolle spezifiziert eine Funktion innerhalb der Organisation, die durch ihren *Namen* beschrieben wird.

Ihr können eine oder mehrere *Personen* mit einer Priorität zugeordnet werden, welche die Rolle ausüben.

Zur Runtime wird die Zuordnung der konkreten Verantwortlichkeit zunächst nach aufsteigender Priorität der Personen vorgenommen. Sobald eine verfügbare Person gefunden ist, wird ihr die Verantwortlichkeit für die entsprechende Aktivität übertragen.

Außerdem kann jeder Rolle eine *Stellvertreter-Rolle* zugeordnet werden. Diese wird dann verwendet, wenn keine der zugeordneten Personen verfügbar ist.

4.4.2 Person

Eine Person verkörpert einen Benutzer, der sich am System anmelden kann. Für sie wird als *Name* ihr Benutzername angegeben.

4.5 Workflow-Definition

Eine Workflow-Definition enthält die Spezifikation von Kontrollfluß und Objektfluß für die Durchführung eines Geschäftsprozesses. Dies beinhaltet alle Aktivitätsknoten, Kommunikationsknoten und Kontrollknoten, Transitionen, Objekt-Spezifikationen, Komponenten-Zuweisungen und externen Datenzugriffe.

Der *Name* der Workflow-Definition beschreibt den zugehörigen realen Arbeitsablauf.

Abbildung 15 enthält ein Beispiel für eine Workflow-Definition. Ausgangspunkt ist die Erteilung einer *imaging-order* über eine Benutzerschnittstelle. Als erste Aktivität wird dann eine *x-ray-examination* durchgeführt. Das Ergebnis der Untersuchung (*XRR: x-ray-report*) wird in einer externen Datenquelle (*PDB*) abgelegt. Je nach Ergebnis unterscheidet sich der weitere Arbeitsablauf vom OR-Split an.

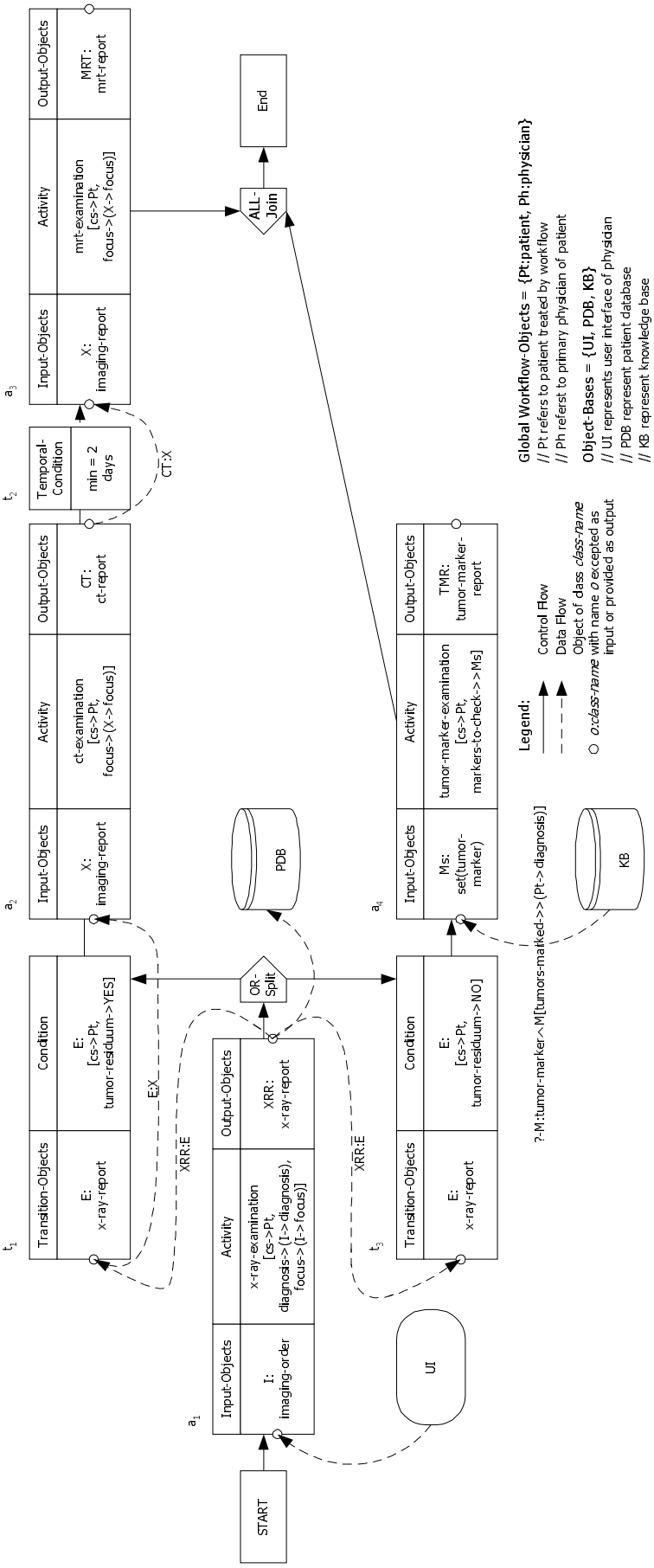


Abbildung 15: Beispiel-Workflow (aus Projektbestand)

Ist die Transitionsbedingung *E:[cs->Pt, tumor-residuum->YES]* erfüllt, so wird eine *ct-examination* vorgenommen und nach einer Wartezeit von mindestens 2 Tagen außerdem eine *mrt-examination*.

Ist jedoch die Transitionsbedingung *E:[cs->Pt, tumor-residuum->NO]* erfüllt, so werden aus einer externen Datenquelle Informationen über *tumor-marker* des Patienten abgerufen. Unter Verwendung dieser wird dann eine *tumor-marker-examination* durchgeführt.

Die Verzweigung des Arbeitsablaufes endet mit dem All-Join, womit dann auch das Ende des Arbeitsablaufs erreicht ist.

5 UML-Modellierung des Workflow-Modells

Nachdem in Kapitel 4 die Konstrukte des Workflow-Modells aus Sicht des Anwenders erläutert wurden, soll nun ihre konkrete objektorientierte Modellierung mittels UML als Grundlage für die Implementierung beschrieben werden.

5.1 Hilfsklassen

Die Hilfsklassen stellen Datenstrukturen bereit, die nicht Workflow-spezifisch sind und in verschiedenen Kontexten verwendet werden.

a) Duration

Die Klasse Duration repräsentiert eine zeitliche Dauer. Die Attribute *min*, *average* und *max* geben die minimale, durchschnittliche und maximale Dauer an. *unit* ist die Einheit, in der die Angaben gemacht werden, möglich sind: *second, minute, hour* und *day*.

b) Point

Die Klasse Point spezifiziert einen Punkt anhand einer *x*- und einer *y*-Koordinate. Sie wird im Zusammenhang mit der grafischen Darstellung von Workflows verwendet.

c) Point-Array

Die Klasse Point-Array wird für die grafische Darstellung von Objekten verwendet, die mehrere Positionsangaben benötigen, wie zum Beispiel Polygonzüge. Sie wird bei der Darstellung von Workflow-Transitions und Component-Mappings verwendet.

5.2 Basisklassen

a) Activity-Definition

Die Klasse Activity-Definition enthält die Aktivitätsdefinitionen.

Activity-Definition ist die (rein virtuelle[2]) Oberklasse der Klassen Basic-Activity-Definition und Complex-Activity-Definition.

Das Attribut *name* enthält einen Namen, der die Aktivität kurz beschreibt. Er dient insbesondere in der grafischen Darstellung als Unterscheidungsmerkmal für die Aktivitäten.

Die Assoziation *defines* gibt an, bei welchen Activity-Nodes die Activity-Definition verwendet wurde.

Die Assoziation *inputs* gibt an, welche Object-Specifications die für die Aktivität benötigten Eingabeobjekte spezifizieren.

Analog dazu geben die über *outputs* assoziierten Object-Specifications an, welche Ausgabeobjekte die Aktivität bereitstellt.

b) Basic-Activity-Definition

Die Klasse Basic-Activity-Definition ist von Activity-Definition abgeleitet. Sie repräsentiert einfache Aktivitätsdefinitionen.

Die über *definition* assoziierte Object-Specification enthält die F-Logic-Klasse, welche die Aktivität repräsentiert.

Über die Assoziation *has-applications* werden der Aktivität Applications zugeordnet, die ausgeführt werden sollen, wenn die Aktivität aktiviert wird.

Die über *estimated-duration* zugeordnete Duration gibt eine geschätzte Ausführungsdauer für die Aktivität an. Diese Angabe wird vom Adaptation-Agent benötigt, um das Ausmaß einer Adaptationsregion abzuschätzen.

c) Application

Die Klasse Application repräsentiert eine Anwendung.

Über *input* kann eine Object-Specification assoziiert werden, die spezifiziert, von welcher Klasse das erwartete Eingabeobjekt der Anwendung ist.

Analog dazu kann über *output* angegeben werden, von welcher Klasse das bereitgestellte Ausgabeobjekt der Anwendung ist.

---

[2] Das bedeutet, daß von dieser Klasse keine Objektinstanzen erzeugt werden. Sie stellt lediglich eine Beschreibung für die gemeinsamen Eigenschaften der abgeleiteten Klassen dar.

Das Attribut *name* spezifiziert den Namen der CORBA-Schnittstelle der Anwendung.

Das Attribut *type* gibt an, um welche Art von Applikation es sich handelt.

Mittels der Attribute *state-available* und *transactionable* wird die Verfügbarkeit von Statusinformationen und die Transaktionsfähigkeit angegeben, das Attribut *automatic* gibt an, ob Benutzerinteraktionen erforderlich sind oder die Applikation automatisch abläuft.

d) Complex-Activity-Definition

Die Klasse Complex-Activity-Definition ist von Activity-Definition abgeleitet.

Über *defined-by* wird die Workflow-Definition angegeben, die ausgeführt werden soll, wenn die komplexe Aktivität aktiviert wird. Die über die Assoziationen *inputs* bzw. *outputs* der Basisklasse angegebenen Object-Specifications stehen innerhalb der zugeordneten Workflow-Definition bereit, um mittels Component-Mappings ihre Daten abzurufen bzw. ihnen Daten zuzuweisen.

e) Role

Die Klasse Role repräsentiert eine Rolle. Das Attribut *name* gibt die Bezeichnung für die Funktion an.

Die attributierte Assoziation *performed-by* gibt die Personen an, welche die Rolle ausüben. Mittels des Attributs *priority* wird eine Reihenfolge für die Verantwortlichkeit festgelegt.

Die Assoziation *substituted-by* gibt eine stellvertretende Role an, umgekehrt werden über *substitute-for* die Rollen assoziiert, für welche die Rolle Stellvertreter ist.

f) Person

Die Klasse Person repräsentiert eine Person.

Das Attribut *name* enthält den Benutzernamen der Person.

Die Assoziation *has-roles* gibt an, welche Roles der Person zugeordnet sind.

5.3 Workflow-Definition und Kontrollfluß-Klassen

Abbildung 16 enthält das UML-Schema der allgemeinen Klassen und Kontrollflußklassen. Im folgenden werden diese erläutert.

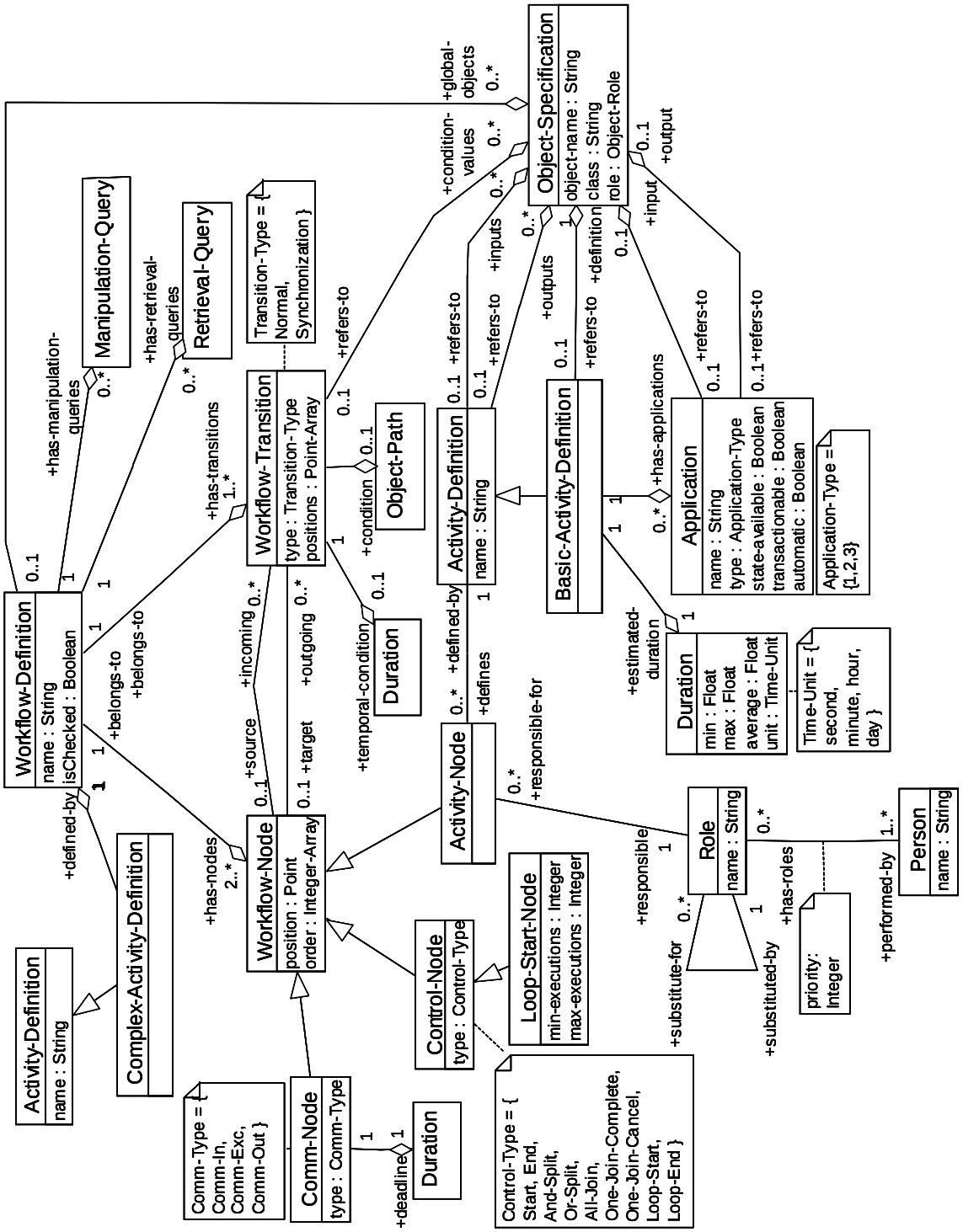


Abbildung 16: UML-Schema (1) – allgemeine und Kontrollfluß-Klassen

a) Workflow-Definition

Das Attribut *name* enthält die Bezeichnung der Workflow-Definition.

Über *is-checked* wird angegeben, ob die Workflow-Definition mittels der Verifikations-Algorithmen erfolgreich geprüft wurde. Nur wenn dieses Attribut auf *true* gesetzt ist, können Instanzen zu dieser Workflow-Definition erzeugt werden.

Die Assoziationen *has-nodes* bzw. *has-transitions* geben die in der Workflow-Definition enthaltenen Workflow-Nodes bzw. Workflow-Transitions an.

Die Assoziationen *has-manipulation-queries* und *has-retrieval-queries* geben die entsprechenden enthaltenen Manipulation- bzw. Retrieval-Queries an.

Über *global-objects* werden Object-Specifications assoziiert, die in der Workflow-Definition global verfügbar sein sollen.

b) Workflow-Transition

Die Klasse Workflow-Transition beinhaltet die Transitionen.

Das Attribut *type* gibt an, um welchen Type von Transition es sich handelt. Mögliche Werte sind *Normal* und *Synchronization.* Das Attribut *positions* wird zur grafischen Darstellung der Transition benutzt.

Die über *temporal-condition* assoziierte Duration gibt eine Mindestwartezeit (Attribut *min*) bzw. Höchstwartezeit (Attribut *max*) an.

Über *condition* kann ein Object-Path assoziiert werden, der eine wertebasierte Bedingung enthält, die zugehörigen Object-Specifications, auf denen die Bedingung formuliert ist, werden über *condition-values* assoziiert. Die wertebasierte Bedingung wird in F-Logic formuliert und im Buildtime-Modell strukturiert abgelegt.

Syntaktisch sind die wertebasierten Bedingungen, die Transitionen zugeordnet werden können, den Bedingungen bei *Retrieval-Queries* sehr ähnlich, siehe dazu auch 5.4.2a) .

Ein Unterschied besteht in ihrer Auswertung: Bei einer Query werden alle diejenigen Objekte aus einer gegebenen Menge ausgewählt, welche die Bedingung erfüllen. Ergebnis ist die Menge der ausgewählten Objekte.

Bei einer Transitionsbedingung hingegen sind die Objekte gegeben. Es ist zu ermitteln, ob diese die Bedingung erfüllen, Ergebnis ist demzufolge ein Wahrheitswert.

Daraus ergibt sich, daß die wertebasierten Transitionsbedingungen analog zu den Bedingungen bei *Retrieval-Queries* als *Object-Paths* im Buildtime-Modell abgelegt werden können. Der Unterschied besteht lediglich darin, daß eine Transitionsbedingung keine Ausgabeobjekte *(outputs*) besitzt.

c) Workflow-Node

Die Klasse Workflow-Node ist die (rein virtuelle) Oberklasse von Activity-Node, Control-Node und Comm-Node.

Das Attribut *position* wird bei der grafischen Darstellung verwendet.

Im Attribut *order* wird eine Ordnungsnummer abgelegt, welche die direkte Ermittlung der Lage zweier gegebener Knoten zueinander ermöglicht, siehe dazu 6.2.3.4.

d) Comm-Node

Die Klasse Comm-Node repräsentiert Kommunikationsknoten.

Das Attribut *type* gibt an, um welche Art von Kommunikationsknoten es sich handelt. Mögliche Werte sind *Comm-In*, *Comm-Exc* und *Comm-Out*.

Die über *deadline* assoziierte Duration gibt mittels des *max*-Attributs an, wie lange höchstens auf eine erfolgreiche Kommunikation gewartet werden soll. Wird diese Wartezeit überschritten, dann wird eine Fehlermeldung ausgelöst. Im *average*-Attribut wird die durchschnittliche Ausführungsdauer festgelegt, die der Adaptation-Agent verwendet.

e) Control-Node

Die Klasse Control-Node repräsentiert einen Kontrollknoten. Das Attribut *type* gibt an, von welchem Typ der Kontrollknoten ist. Mögliche Typen sind: *Start, End, And-Split, Or-Split, All-Join, One-Join-Complete, One-Join-Cancel, Loop-Start, Loop-End*. Falls der Typ *Loop-Start* ist, dann wird die Unterklasse Loop-Start-Node verwendet.

f) Loop-Start-Node

Die Klasse Loop-Start-Node ist Unterklasse von Control-Node. Sie enthält die für Control-Nodes vom Typ *Loop-Start-Node* zusätzlich nötigen Attribute.

Die Attribute *min-executions* bzw. *max-executions* geben eine Schätzung darüber an, wie oft die Schleife mindestens bzw. höchstens durchlaufen wird.

g) Activity-Node

Die Klasse Activity-Node repräsentiert einen Knoten, der eine Aktivität ausführt.

Über *defined-by* wird eine Activity-Definition assoziiert, welche die durchzuführende Aktivität beschreibt.

Die Assoziation *responsible* verweist auf die für die Aktivität verantwortliche Role.

Über die Assoziation *has-paths* werden die zugehörigen Object-Paths angegeben (s. 5.4.1c)).

5.4 Objektfluß-Klassen

Abbildung 17 enthält das UML-Schema der Objekflußklassen.

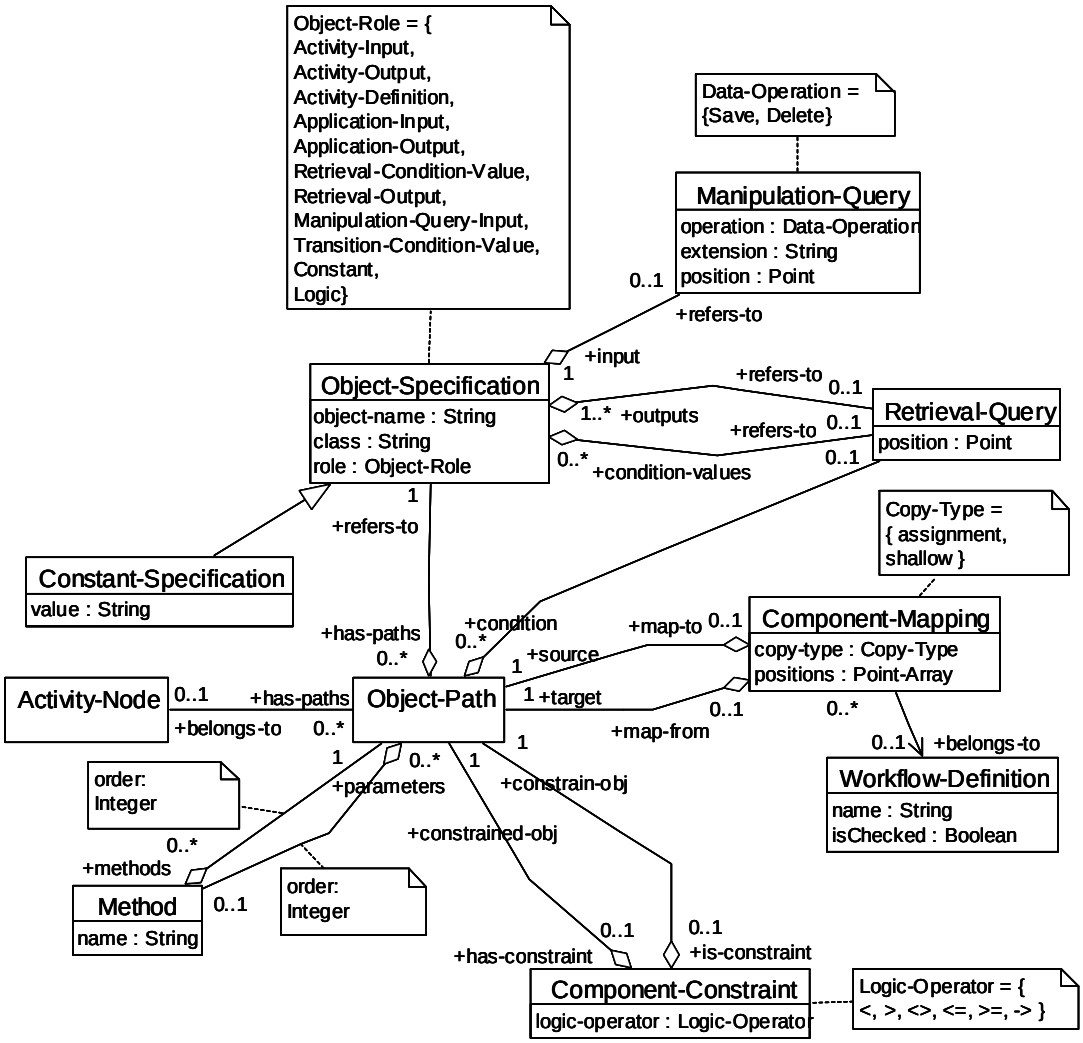


Abbildung 17: UML-Schema (2) - Objektflußklassen

Im folgenden werden zunächst die Klassen des internen Objektflusses erläutert, daran anschließend diejenigen des externen Objektflusses.

5.4.1 Interner Objektfluß

a) Object-Specification

Die Klasse Object-Specification enthält die Objekt-Spezifikationen.

Im Attribut *object-name* wird der Name des spezifizierten Objekts abgelegt, das Attribut *class* enthält dessen Klasse.

Das Attribut *role* enthält eine Angabe darüber, in welchem Kontext die Objekt-Spezifikation verwendet wird. Es wird beim Erstellen der Object-Specification automatisch gesetzt und während der Verifikation des Objektflusses verwendet. Mögliche Werte sind: *Activity-Input, Actitity-Output, Activity-Definition, Application-Input, Application-Output, Retrieval-Condition-Value, Retrieval-Output, Manipulation-Query-Input, Transition-Condition-Value* und *Logic*. Für Konstanten *(role* = C*onstant*) wird die spezielle Klasse *Constant-Specification* verwendet.

Über die Assoziation *refers-to* wird das Objekt angegeben, in dessen Kontext die Object-Specification verwendet wird. Der Typ des assoziierten Objekts kann aus der *role* abgeleitet werden. Von allen möglichen *refers-to*-Assoziationen einer Object-Specification wird jeweils höchstens eine belegt.

Über *has-paths* werden die Object-Paths assoziiert, die auf das spezifizierte Objekt Bezug nehmen.

b) Constant-Specification

Constant-Specification spezifiziert eine Konstante. Die Klasse ist von Object-Specification abgeleitet.

Im Attribut *value* kann ein Wert angegeben werden. Wird kein Wert angegeben, so wird dieser bei Erzeugung der Workflow-Instanz ermittelt.

c) Object-Path

Die Klasse Object-Path enthält Objekt-Pfade.

Über die attributierte Assoziation *methods* werden die Komponenten angegeben, aus denen sich der Object-Path zusammensetzt. Das Attribut *order* gibt dabei die Reihenfolge der Komponenten im Objekt-Pfad an.

Die Assoziation *refers-to* gibt die Object-Specification an, auf die sich der Object-Path bezieht.

Eine der beiden Assoziationen *map-to* und *map-from* kann auf ein Component-Mapping verweisen, bei dem der Object-Path verwendet wird.

Analog kann eine der beiden Assoziationen *has-constraint* und *is-constraint* auf ein Component-Constraint verweisen, das den Object-Path verwendet.

Die Assoziation *belongs-to* gibt an, zu welchem Activity-Node ein Object-Path gehört. Diese Assoziation ist aus folgendem Grund nötig:

Eine Activity-Definition kann bei verschiedenen Activity-Nodes mehrfach verwendet werden. Damit werden implizit auch die zur Activity-Definition gehörigen Object-Specifications mehrfach verwendet. Dann ist aber die eindeutige Zuordnung eines Object-Paths zu einer Activity-Node zunächst nicht möglich. Zur Runtime muß aber bei der Ausführung einer Activity-Node klar sein, welche Object-Paths und damit verbundene Component-Mappings im Kontext eines Activity-Nodes verwendet werden müssen. Die Assoziation *belongs-to* ermöglicht diese Zuordnung (s. a. 5.5.2).

d) Method

Die Klasse Method beinhaltet einzelne Elemente eines Object-Paths.

Das Attribut *name* enthält den Namen der Methode bzw. Komponente.

Die attributierte Assoziation *parameters* verweist auf Object-Paths, welche die Daten für den Methodenaufruf bereitstellen. Das Attribut order gibt die Reihenfolge der Parameter in der Parameterliste an.

Methoden ohne Parameter sind ebenfalls möglich, diese stellen Attribute dar.

e) Component-Mapping

Component-Mappings enthalten die Komponenten-Zuweisungen.

Das Attribut *copy-type* gibt die Art der Zuweisung an. Mögliche Arten sind: *assignment* und *shallow.*

Die Assoziationen *source* bzw. *target* geben die Object-Paths an, die das Quell-Objekt bzw. Ziel-Objekt der Zuweisung spezifizieren.

Über *belongs-to* wird ggf. die Workflow-Definition assoziiert, zu deren Objektfluß das Component-Mapping gehört.

In *positions* werden die grafischen Positionen für die Darstellung abgelegt.

f) Component-Constraint

Ein Component-Constraint spezifiziert eine Komponenten-Bedingung.

Die Assoziationen *constrain-obj* bzw. *constrained-obj* geben die Object-Paths an, welche die zu vergleichenden Objekte liefern.

Der *logic-operator* gibt den Vergleichsoperator an, der die Werte <,>, <>, <=, >=, –> haben kann.

Die Reihenfolge der beiden zu vergleichenden Objekte wird wie folgt festgelegt: *constrained-obj logic-operator constrain-obj*.

5.4.2 Externer Objektfluß

Der externe Objektfluß ermöglicht das Abrufen von Daten aus externen Datenquellen durch *Retrieval-Queries* und das Ablegen von Daten in diesen durch *Manipulation-Queries.*

a) Retrieval-Query

Bei der Modellierung der *Retrieval-Queries* besteht eine wesentliche Aufgabe darin, eine Abfragesprache festzulegen, welche die Heterogenität der Datenbestände verbirgt. Da die Basis für die Workflow-Objektmodellierung ein F-Logic-Klassenmodell ist, sollte die Abfragesprache für die externen Datenabfragen auf der Syntax der F-Logic-Queries basieren. Dazu wurde eine Auswahl aus den syntaktischen Konstrukten von F-Logic getroffen, so daß eine ausreichende Ausdrucksmächtigkeit für die Auswahl von Objekten erreicht wird. Diese wird in Anhang A erläutert.

Für diese syntaktischen Konstrukte wurde eine entsprechende Umsetzung und strukturierte Repräsentation in mehreren Klassen erstellt, die sich in das übrige UML-Buildtime-Modell einfügt. Die Umsetzung ist in Anhang B dargestellt, hier soll nur ein Überblick gegeben werden.

Die *Object-Specifications* der Assoziation *outputs* spezifizieren die Objekte, die als Ergebnis des Queries zurückgegeben werden. Die in den *Object-Specifications* angegebenen *classes* spezifizieren dabei die Mengen der Objekte, aus denen die Objekte ausgewählt werden. Dies entspricht etwa der FROM-Klausel eines SQL-Statements.

Der *Object-Path* der Assoziation *condition* enthält die gesamte Bedingung der Query. Die als *outputs* und *condition-values* der *Retrieval-Query* angegebenen *Object-Specifications* werden in ihm referenziert, er verknüpft diese Bestandteile zum Bedingungsteil der Query und entspricht damit dem WHERE-Teil eines SQL-Statements.

Die *Object-Specifications* der Assoziation *condition-values* spezifizieren die Objekte, deren Daten als Bedingungen in der Query verwendet werden. Dies entspricht der Verwendung von Variablen im WHERE-teil eines SQL-Statements, deren Werte erst zur Laufzeit zur Vervollständigung des Statements eingesetzt werden.

Ein dem SELECT-Teil eines SQL-Statements entsprechender Teil ist bei der Retrieval-Query nicht direkt enthalten. Sie gibt immer ganze Objekte zurück. Welche Komponenten von diesen dann verwendet werden, wird mit Hilfe von Component-Mappings in der Workflow-Definition festgelegt.

Im Attribut *position* wird die grafische Position innerhalb der Workflow-Definition abgelegt.

b) Manipulation-Query

Manipulation-Queries ermöglichen das Einfügen, Ändern oder Löschen von Objekten in externen Datenquellen.

Ein Objekt der Klasse *Manipulation-Query* repräsentiert die Änderung eines Objekts im Datenbestand.

Die *Object-Specification* der Assoziation *inputs* gibt das betreffende Objekt an.

Die Eigenschaft *operation* gibt an, welche Änderungsoperation durchzuführen ist:

- *Save* fügt ein neues Objekt in den Datenbestand ein (Insert) oder ändert ein bestehendes Objekt (Update). Welche Operation auszuführen ist, hängt davon ab, auf welche Art das zu speichernde Objekt vom CML angelegt wurde. Wenn das Objekt ursprünglich Ergebnis einer Retrieval-Query war, wird die Update-Operation ausgeführt. Wurde das Objekt ursprünglich als leeres Objekt angelegt und anderweitig mit Daten ge-

füllt (z. B. durch Applikationen), dann wird die Insert-Operation durchgeführt. Die Information, wie ein Objekt angelegt wurde, muß dazu vom CML verwaltet werden.

- *Delete* löscht ein Objekt aus dem Datenbestand

Die Eigenschaft *extension* gibt an, in welcher Objektausdehnung in den physischen Datenquellen das Objekt gespeichert werden soll, falls eine Insert-Operation durchgeführt wird. Im Falle einer Update- oder Delete-Operation wird diese Angabe vom CML ignoriert und die Änderungen werden in der Extension durchgeführt, in der sich das Objekt schon befindet.

5.5 Beispiele

Die folgenden Beispiele sollen die Zusammenhänge verschiedener Klassen des Buildtime-Modells verdeutlichen.

5.5.1 Activity-Definitions und Workflow-Definition

In diesem Beispiel werden die Zusammenhänge von Basic-Activity-Definition, Complex-Activity-Definition und Workflow-Definition dargestellt.

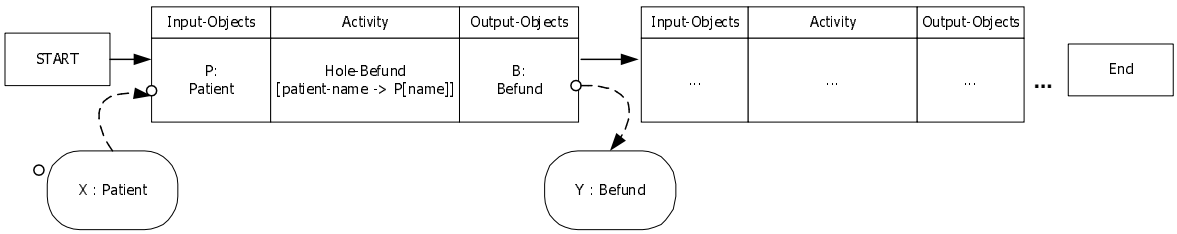


Abbildung 18: Workflow-Definition "Befund auswerten"

In Abbildung 18 ist ein vereinfachtes Beispiel einer Workflow-Definition dargestellt. Diese Workflow-Definition gehört zu einer Complex-Activity-Definiton "Befund auswerten". Das Input-Objekt *X:Patient* der Complex-Activity-Definition wird in der Workflow-Definition verwendet und dem Input-Objekt *P:Patient* der Activity "Hole-Befund" zugewiesen. Analog dazu wird das Output-Objekt *B:Befund* in der Workflow-Definition dem Output-Objekt *Y : Befund* der Complex-Activity-Definition zugewiesen.

In Abbildung 19 ist dieses Beispiel aus einer anderen Sichtweise dargestellt. Sie enthält die *Objektinstanzen* der Klassen, in denen die Daten der Objekte aus Abbildung 18 abgelegt sind.

Die grau eingerahmte Teilstruktur enthält die Elemente, die zu einer einfachen Aktivitäts-definition mit dem Namen "Hole-Befund" gehören, deren F-Logic-orientierte Darstellung wie folgt ist:

- inputs: *P : Patient*
- definition: *H : Hole-Befund[patient-name –>P[name]]*
- outputs: *B : Befund*

Die zugehörigen Application-Instanzen sind nicht dargestellt.

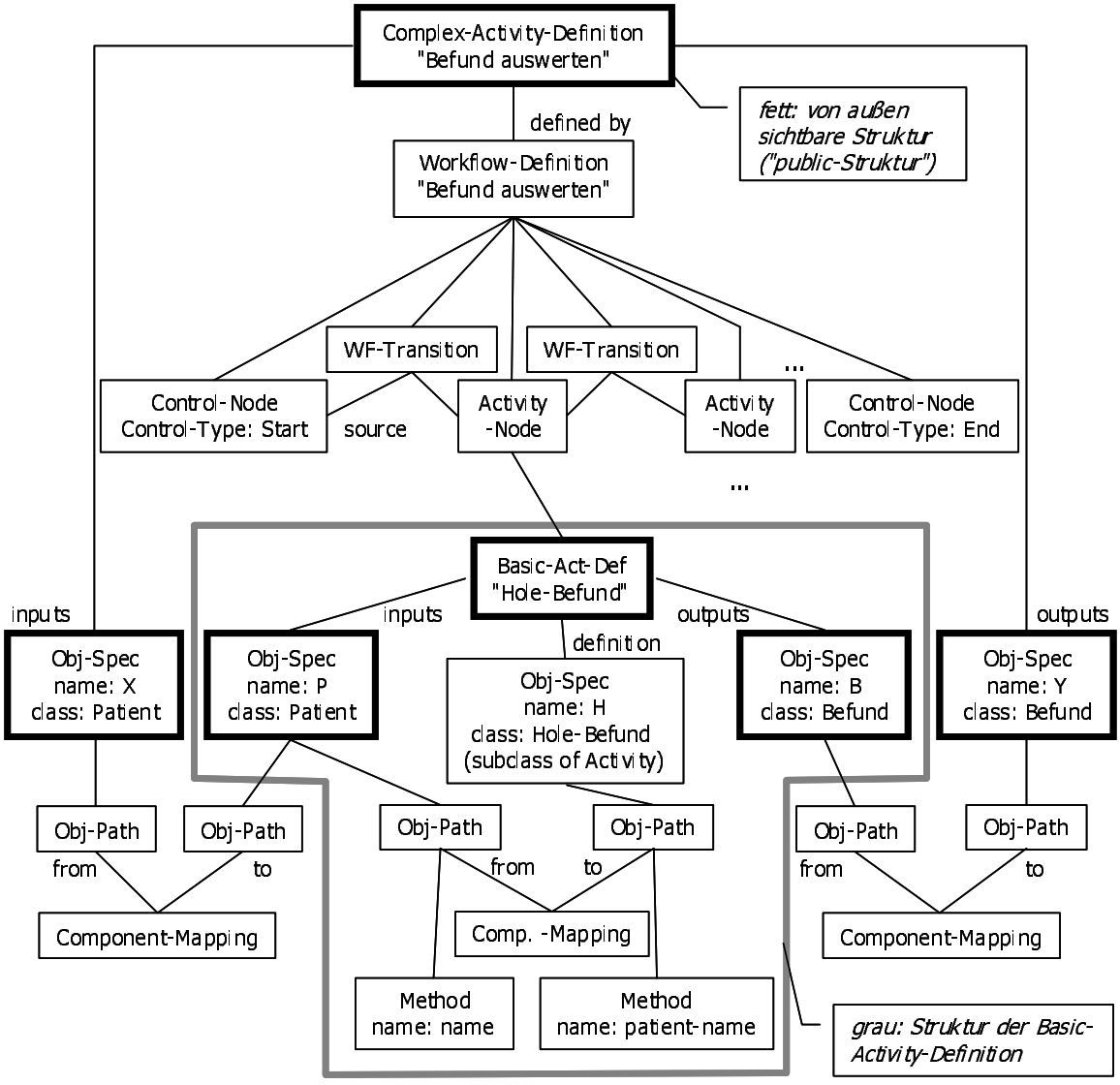


Abbildung 19: Beispiel von Activity-Definitions und Workflow-Definition

Diese Basic-Activity-Definition hat dabei folgende Schnittstellen zu ihrer Verwendung in einer Workflow-Definition: Über die Assoziation der *Basic-Activity-Definition* selbst zu

einer Activity-Node wird sie in den *Kontrollfluß* einer Workflow-Definition eingebunden. Die beiden Object-Specifications *P : Patient* und *B : Befund* werden über Object-Paths in den *Objektfluß* der Workflow-Definition eingebunden.

Diese ist wiederum einer komplexen Aktivitätsdefinition "Befund auswerten" zugeordnet. Diese besitzt ein Input-Objekt *X : Patient* und ein Output-Objekt *Y : Befund*. Die beiden Object-Specifications sind am Objektfluß der Workflow-Definition beteiligt, in dem sie mit Hilfe von Object-Paths und Component-Mappings mit den Object-Specifications der Basic-Activity-Definition verbunden werden.

Diese gesamte Complex-Activity-Definition könnte nun wie im nächsten Beispiel gezeigt in verschiedenen Workflow-Definitionen verwendet werden.

5.5.2 Mehrfache Verwendung einer Activity-Definition

An dem Beispiel in Abbildung 20 soll insbesondere die Zugehörigkeit von Object-Paths zu übergeordneten Objekten verdeutlicht werden.

Dargestellt ist eine Complex-Activity-Definition "Befund auswerten", sowie ausschnittsweise zwei Workflow-Definitionen, in denen diese verwendet wird.

Die Activity-Definition besitzt ein Input-Objekt *X : Patient* und ein Output-Objekt *Y : Befund*. Diese sind jeweils über Object-Paths und Component-Mappings in den Objektfluß der Workflow-Definitionen eingebunden. Die Object-Paths sind dabei jeweils den Workflow-Definitionen zugehörig, in denen sie über Component-Mappings verwendet werden.

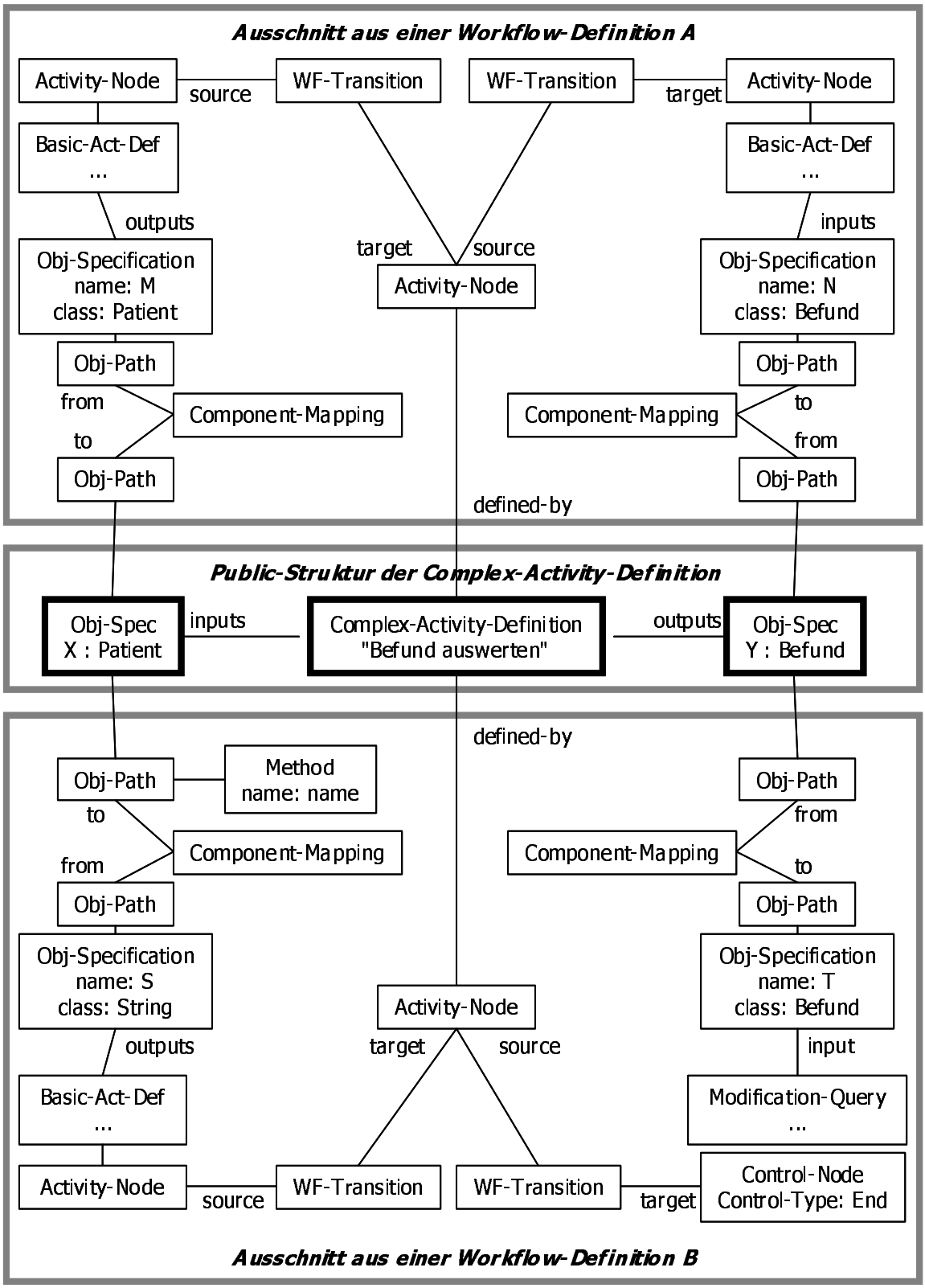


Abbildung 20: Beispiel für mehrfache Verwendung einer Activity-Definition

6 Workflow-Editor

6.1 Anforderungen

6.1.1 Funktionalität

Der Workflow-Editor soll zur graphischen Erstellung und Bearbeitung von Workflow-Definitionen und den damit zusammenhängenden Objekten des UML-Buildtime-Modells, wie sie im vorangegangenen Kapitel vorgestellt wurden, dienen.

Die erstellten Workflow-Definitionen müssen gewissen Korrektheitskriterien genügen, um die Ausführbarkeit durch die Workflow-Engine sicherzustellen. Dabei ist es wünschenswert, die Korrektheit möglichst weitgehend sicherzustellen, da der Aufwand zur Behandlung eines Fehlers zur Runtime wesentlich höher ist als zur Buildtime. So ist beispielsweise fehlende Typkompatibilität einer Zuweisungsoperation zur Buildtime durch die vorliegenden Typinformationen leicht feststellbar, würde jedoch zur Runtime zu einem Fehler in der Workflow-Ausführung bis hin zum Abbruch des gesamten Workflows führen.

Die Workflow-Definitionen müssen persistent gespeichert werden und den anderen Modulen des Systems zur Verfügung stehen. Dabei muß es aber auch möglich sein, nicht korrekte Workflow-Definitionen zu speichern, um diese zu einem späteren Zeitpunkt weiterzubearbeiten.

Außerdem soll der Export der grafischen Workflow-Darstellung in einem verbreiteten Vektorgrafik-Format zu Dokumentationszwecken möglich sein.

6.1.2 Ergonomie

Die Aufgabe der Erstellung von Workflow-Definitionen ist für den Benutzer grundsätzlich relativ komplex. Dies hängt mit der großen Anzahl von Parametern zusammen, die angegeben werden müssen, um einen Workflow vollständig zu beschreiben.

Daher ist es wünschenswert, daß die Benutzeroberfläche des Workflow-Editors möglichst weitreichende Unterstützung bietet und dem Benutzer eine intuitive Arbeitsweise ermöglicht.

6.1.3 Zusätzliche Anforderungen

In [LR00] werden verschiedene Anforderungen an Workflow-Systeme aufgestellt, einige davon sind auch für den Workflow-Editor relevant:

- *Sicherheit:* Workflow-Definitionen sind sicherheitstechnisch sensible Informationen einer Organisation. Da zu ihrer Erstellung ein hoher Aufwand nötig ist, stellen sie Informationen von nicht unerheblichem Wert dar. Außerdem können inkorrekte Veränderungen von Workflow-Definitionen eine große negative Auswirkung auf die Arbeitsabläufe in einer Organisation haben. Workflow-Definitionen müssen deshalb vor unberechtigtem Zugriff und Veränderung geschützt werden.
- *Zuverlässigkeit:* Auch im Falle eines Fehlers müssen die Daten im Workflow-System immer konsistent bleiben, für den Workflow-Editor bedeutet dies, daß die Änderungen an Workflow-Definitionen und anderen Daten immer durch Transaktionen abgesichert werden müssen.
- *Kapazität:* Die Kapazität des Workflow-Systems muß ausreichen, um die gegebene Anzahl der Benutzer und die damit verbundenen Aktivitäten zu unterstützen. Für den Workflow-Editor würde dies idealerweise bedeuten, daß beliebig viele Benutzer zur gleichen Zeit Workflow-Modellierungen vornehmen können. Probleme ergeben sich daraus, wenn mehrere Benutzer zur gleichen Zeit die gleichen Daten ändern, was zu Inkonsistenzen führen würde. Diese könnten durch den Einsatz konventioneller ACID-Transaktionen verhindert werden. Allerdings ist ihr Einsatz problematisch, da sie auf Grund der langen Bearbeitungsvorgänge und stark vernetzten Objektstrukturen des UML-Buildtime-Modells zu langfristigen und weitreichenden Sperren von Datenbeständen führen würden. Ein möglicher Lösungsansatz für diese Problematik ist der CONCORD-Ansatz [Rit97]. In ihm wird deshalb eine workflow-ähnliche Vorgehensweise für solche Entwurfsprozesse vorgeschlagen, die den Gesamt-Entwurfsprozeß in einzelne Teilprozesse zerlegt, die sich wiederum aus einzelnen Entwurfsaktionen zusammensetzen.

6.2 Konzept

Die Benutzeroberfläche des Workflow-Editors in dieser Arbeit ermöglicht die interaktive Bearbeitung aller für die Workflow-Definitionen relevanten Objekte.

Grundsätzlich werden immer folgende Informationen dargestellt (in Abbildung 21 links):

- Liste der vorhandenen Workflow-Definitionen, gegliedert nach normalen Workflows und Sub-Workflows
- Liste der vorhandenen Aktivitäts-Definitionen, gegliedert nach einfachen und komplexen Aktivitäten
- Liste der vorhandenen Personen und Rollen

Für diese Listen sind jeweils Funktionen zur Erstellung von neuen Objekten, Änderung der vorhandenen Objekte bzw. deren Löschung vorgesehen.

Insbesondere können Workflow-Definitionen aus der Liste heraus geöffnet werden, um sie grafisch zu bearbeiten. (Abbildung 21 rechts)

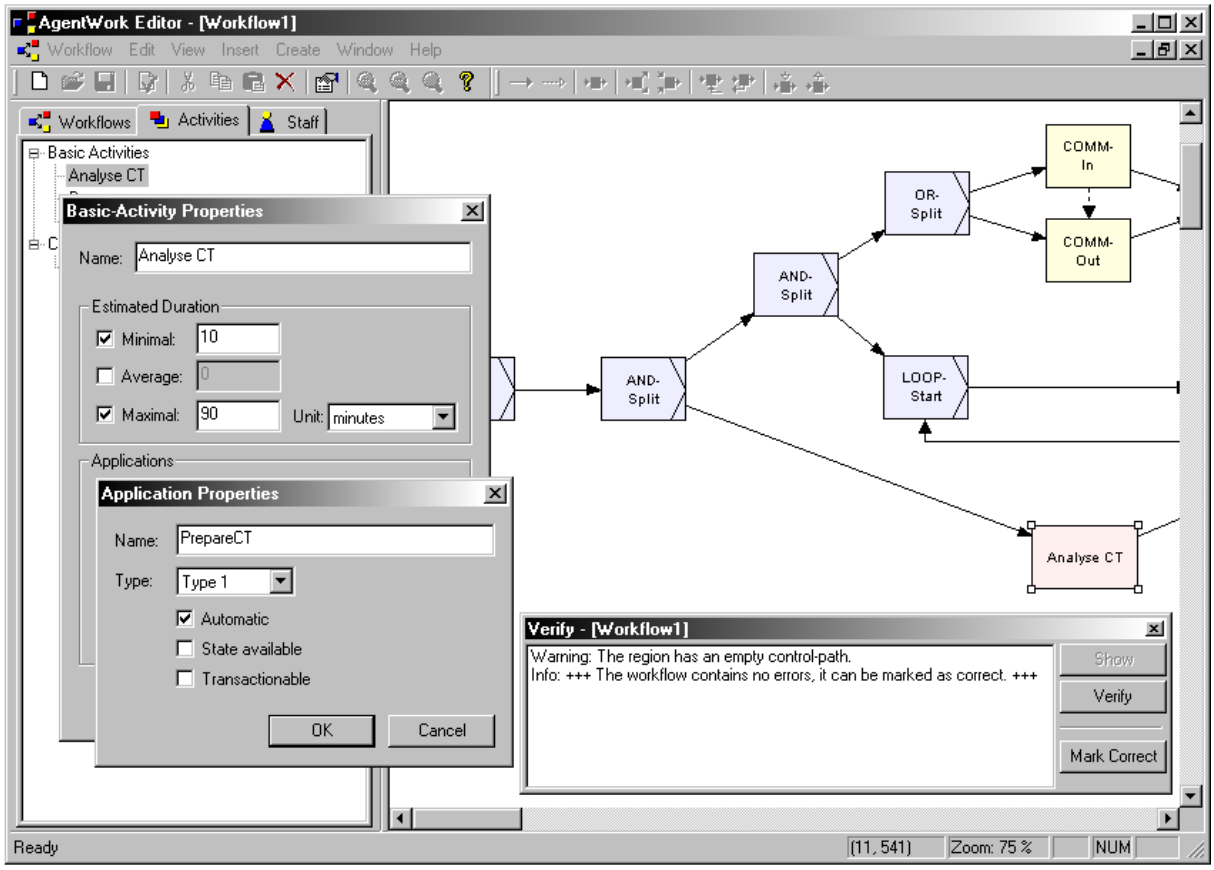


Abbildung 21: Benutzeroberfläche des Workflow-Editors

Innerhalb von Workflow-Definitionen werden außerdem folgende Funktionsgruppen angeboten:

- Einfügen neuer Objekte
- Markierung von einzelnen Objekten oder Objektgruppen
- Verschieben oder Löschen der markierten Objekte
- Bearbeitung der Eigenschaften markierter Objekte

In jedem Fall werden nur die Funktionen auf der Benutzeroberfläche aktiviert, die mit den aktuell markierten Objekten durchführbar sind.

Im Folgenden soll ein Überblick über die zur Verfügung stehenden Funktionen gegeben werden.

6.2.1 Bearbeitung von Basisobjekten

Basisobjekte sind die Objekte, die im Workflow-Editor global zur Verfügung stehen und gleichzeitig in verschiedenen Workflow-Definitionen verwendet werden können.

Funktion	**Erläuterung**
Basic-Activity-Definition anlegen / ändern	Erstellt bzw. ändert eine vorhandene Basic-Activity-Definition.
Basic-Activity-Definition löschen	Löscht eine vorhandene Basic-Activity-Definition.
Applikation zuordnen / ändern	Die Applikation ist jeweils einer Basic-Activity-Definition zugeordnet.
Applikation löschen	Löscht eine vorhandene Applikation.
Complex-Activity-Definition anlegen / ändern	Erstellt bzw. ändert eine Complex-Activity-Definition und die zugehörige gleichnamige Workflow-Definition.
Complex-Activity-Definition löschen	Löscht eine vorhandene Complex-Activity-Definition.
Rolle anlegen / ändern	Erstellt bzw. ändert eine Rolle.
Rolle löschen	Löscht eine Rolle.
Person anlegen / ändern	Erstellt bzw. ändert eine Person.
Person löschen	Löscht eine Person.

Tabelle 1: Funktionen zur Bearbeitung von Basisobjekten

6.2.2 Bearbeitung von Workflow-Definitionen

Ziel der Bearbeitung von Workflow-Definitionen ist es, Workflow-Definitionen zu erstellen, die im Sinne der zulässigen Konstrukte des Workflow-Modells korrekt sind. Die zulässigen Basis-Konstrukte wurden im Kapitel 4 vorgestellt.

Um der Anforderung korrekter Workflow-Definitionen zu genügen, standen zwei verschiedene Modelle für die Benutzeroberfläche zur Diskussion.

Das Modell der *operatorbasierten Bearbeitung* läßt bei der Bearbeitung der Workflow-Definitionen nur solche Änderungen zu, die eine bestehende korrekte Kontrollflußstruktur in eine andere, wiederum korrekte Kontrollflußstruktur zulassen. Damit ist die grundlegende Korrektheit des Kontrollflusses sichergestellt.

Das Modell der *freien Bearbeitung* ermöglicht dagegen nahezu beliebige Kombinationen der Basis-Konstrukte in der Workflow-Definition, d.h. die Korrektheit der Workflow-Definitionen muß durch eine syntaktische Verifikation des Kontrollflusses und eine Verifikation des Objektflusses sichergestellt werden.

Im folgenden werden die beiden Modelle vorgestellt und abschließend verglichen.

6.2.2.1 Operatorbasierte Bearbeitung

a) Kontrollflußoperatoren

Die Kontrollflußoperatoren überführen unter Berücksichtigung der Bedingungen einen korrekten Kontrollfluß in einen anderen korrekten Kontrollfluß. Dabei führt ein Operator zumeist mehrere einzelne Änderungsoperationen aus.

Operator	**Erläuterung**	**Bedingungen**
Workflow-Definition anlegen	Erster Operator jeder Workflow-Definition, erstellt Start- und End-Knoten mit einer verbindenden Transition	keine
Workflow-Definition löschen		keine
Aktivitätsknoten einfügen	Knoten wird in eine bestehende Transition eingefügt, dadurch wird die bestehende Transition in zwei Transitionen aufgesplittet Problem: Zuordnung einer ggf. vorhandenen Transitionsbedingung zu einer der beiden Transitionen	keine

Operator	**Erläuterung**	**Bedingungen**
Aktivitätsknoten ändern	Zuweisung einer anderen Aktivitätsdefinition	keine
Kommunikationsknoten einfügen	analog zu *Aktivitätsknoten einfügen*	keine
Kommunikationsknoten ändern	Änderung der definierten Ereignisse	keine
Aktivitätsknoten / Kommunikationsknoten löschen	Knoten wird entfernt, ein- und ausgehende Transition werden verschmolzen Problem: Auswahl der verbleibenden Transitionsbedingung	keine
Transitionsbedingung einfügen / ändern	Transitionen werden durch Operatoren zum einfügen von Knoten erstellt, bis auf Synchronisations-Transitionen	wertebasierte Bedingung nur bei von OR-Split ausgehenden Transitionen und Loop-Transitionen einfügbar
Transitionsbedingung löschen		keine
Split/Join-Region einfügen	es werden ein Split- und ein Join-Knoten jeweils in eine gegebene Transition eingefügt, zusätzlich werden zwei Kontrollpfade eingefügt (Bedingungen werden gesondert durch Operator *Transitionsbedingung einfügen* spezifiziert)	die Transitionen, in die eingefügt wird, müssen auf derselben Verschachtelungsebene von Split/Join- und Loop-Regionen liegen ("korrekte Klammerung")
Split-Typ ändern	setzt jeweils anderen Split-Typ ein	nur möglich, wenn zugehöriger Join "All-Join" ist, Wechsel auf "And-Split" nur, wenn alle ausgehenden Kanten ohne wertebasierte Bedingung
Join-Typ ändern		nur möglich, wenn zugehöriger Split „And-Split“ ist
Split-/Join-Region löschen	löscht Split- und zugehörigen Join-Knoten, Kontrollpfad zwischen beiden bleibt erhalten Problem: Auswahl der verbleibenden Transitionsbedingungen (bei Split- und Join-Knoten)	es existiert nur ein Kontrollpfad zwischen Split- und Join-Knoten
Kontrollpfad hinzufügen	fügt eine Transition zwischen dem Split-Knoten und dem zugehörigen Join-Knoten ein	keine
Kontrollpfad entfernen	löscht gesamten Kontrollpfad vom Split- bis zum Join-Knoten	letzter Kontrollpfad in Split-Join-Region kann nicht gelöscht werden

Operator	Erläuterung	Bedingungen
Schleife einfügen	fügt Loop-Start und Loop-End mit Transition Loop-End -> Loop-Start ein (Bedingungen werden gesondert durch Aufruf von *Transitionsbedingung einfügen* spezifiziert)	keine
Schleife Löschen		keine
Synchronisations-Transition einfügen		Vorgänger- und Nachfolger-Knoten liegen in derselben Split/Join-Region, in derselben Verschachtelungstiefe und nicht direkt in einer Loop-Region
Synchronisations-Transition löschen		keine

Tabelle 2: Operatoren zur Kontrollflußbearbeitung

b) Bearbeitung des Objektflusses

Die Korrektheit des Objektflusses hängt von der Kontrollflußstruktur ab, da beispielsweise ein Eingabeobjekt einer Aktivität vorliegen muß, um diese aktivieren zu können. Wird dieses Eingabeobjekt von einem Ausgabeobjekt einer anderen Aktivität mit Daten versorgt, so muß diese deshalb im Kontrollfluß vorher oder parallel liegen.

Wegen dieser Abhängigkeit der Korrektheit des Objektflusses vom Kontrollfluß können keine Operatoren definiert werden, welche die Korrektheit schon bei der Bearbeitung sicherstellt. Es werden daher einfache Funktionen zur Bearbeitung angeboten, der Objektfluß muß abschließend im Zusammenhang mit dem Kontrollfluß verifiziert werden.

Operator	Erläuterung	Bedingungen
Retrieval-Query hinzufügen / ändern	erstellt bzw. ändert einen Retrieval-Query	keine
Manipulation-Query hinzufügen / ändern		keine
Query löschen		keine
Object-Specification anlegen / ändern	wird bei Application, Activity-Definition, Retrieval-Query und Manipulation-Query verwendet	keine
Object-Specification löschen	wird bei Application, Activity-Definition, Retrieval-Query und Manipulation-Query verwendet	keine

Operator	Erläuterung	Bedingungen
Component-Mapping hinzufügen / ändern	ermöglicht auch die Bearbeitung der zugehörigen Objekt-Pfade	die Kombination der Object-Roles von Quell- und Ziel-Objekt muss zulässig sein, Component-Mappings müssen korrekt sein (Typprüfung bei Zuweisungen und Methodenparametern)
Component-Mapping löschen		keine

Tabelle 3: Funktionen zur Bearbeitung des Objektflusses

6.2.2.2 Freie Bearbeitung

a) Bearbeitung des Kontrollflusses

Die Bearbeitung des Kontrollflusses erfolgt mit Funktionen, die jeweils nur einfache Änderungen an den betreffenden Objekten durchführen.

Operator	Erläuterung	Bedingungen
Workflow-Definition anlegen	erstellt Workflow-Definition mit Start- und End-Knoten	keine
Workflow-Definition löschen		Workflow-Definition gehört nicht zu einer Complex-Activity-Definition
Aktivitätsknoten einfügen / ändern	Zuordnung einer Aktivitätsdefinition und einer verantwortlichen Rolle	keine
Kontrollknoten einfügen / ändern	je nach Knotentyp evtl. zusätzliche Parameter	keine
Knoten löschen	löscht Knoten beliebigen Typs	keine
Transition einfügen / ändern	Festlegung des Vorgänger- und Nachfolgerknoten, im Fall "einfügen" wird eine leere Bedingung erzeugt und der Synchronisationstyp auf "normale Transition" gesetzt	keine
Transitionseigenschaften bearbeiten	Bearbeitung der Transitionsbedingung und des Transitionstyps	keine
Transition löschen		keine

Tabelle 4: Einfache Funktionen zur Kontrollflußbearbeitung

b) Bearbeitung des Objektflusses

wie unter 6.2.2.1b)

6.2.2.3 Vergleich der Bearbeitungsmodelle

Die beiden wesentlichen Kriterien für den Vergleich der Bearbeitungsmodelle sind die benötigte Funktionalität in der Implementierung und die Benutzerfreundlichkeit.

a) Korrektheit der Workflow-Definitionen

Die Definition syntaktisch korrekter Kontrollflußkonstrukte ist bei der operatorbasierten Bearbeitung in den Operatoren enthalten und bei der freien Bearbeitung im Verifikationsalgorithmus.

Die Definition der Korrektheit des Objektflusses befindet sich in beiden Fälle im Verifikationsalgorithmus.

b) Implementierung

Operatorbasierte Bearbeitung	**Freie Bearbeitung**
- komplexe Operatoren mit Bedingungsprüfung	- einfache Operatoren
- keine Kontrollflußverifikation erforderlich	- Kontrollflußverifikation erforderlich
- Objektflußverifikation erforderlich	- Objektflußverifikation erforderlich

Tabelle 5: Vergleich der Bearbeitungsmodelle hinsichtlich Implementierung

Insgesamt sind keine wesentliche Unterschiede hinsichtlich des Aufwandes der Implementierung erkennbar.

c) Ergonomie

Die Benutzeroberfläche sollte den Benutzer bei seiner Aufgabe möglichst gut unterstützen und sich an bekannten Konzepten orientieren, um bei der ohnehin komplexen Aufgabe der Definition von Workflows keine zusätzlichen Erschwernisse zu verursachen.

Der Ansatz der *operatorbasierten Bearbeitung* wird diesem Anspruch nur bedingt gerecht. Besonders nachteilig wirkt sich dabei aus, daß der Benutzer seine intuitive Vorstellung des zu modellierenden Workflows zunächst in eine Reihe von Operatoraufrufen umformulieren muß, um die entsprechende Workflow-Definition zu erstellen.

Bei Änderungen an einer Workflow-Definition erweist sich der Ansatz als etwas unflexibel, da u. U. mehrere Operatoren verwendet werden müssen, um eine einfache Änderung auszuführen. Würden dagegen für alle denkbaren Änderungen spezielle Operatoren bereitgestellt, so würde die ohnehin schon hohe Anzahl der Operatoren weiter erhöht und damit die Auswahl eines geeigneten Operators zur Durchführung einer bestimmten Änderung für den Benutzer sehr unübersichtlich.

Das Modell der *freien Bearbeitung* wird dagegen als besser angesehen. Es vermeidet die oben angeführten Nachteile, indem es nur einfache Funktionen zur Verfügung stellt, die für den Benutzer sowohl von ihrer Anzahl als auch von ihren Auswirkungen her leichter überschaubar sind.

Die Funktionen der grafischen Bearbeitung orientieren sich eng an den einzelnen grafischen Objekten und sollten damit auch besser den allgemein bekannten Konzepten von grafischen Darstellungswerkzeugen entsprechen.

d) Auswahl des Modells

Der Ansatz der *freien Bearbeitung* verspricht einen Vorteil bei der Benutzerfreundlichkeit und wird deshalb als Grundlage für die Implementierung gewählt. Der mit der Auswahl diese Ansatzes notwendig gewordene Verifikationsalgorithmus wird im folgenden Abschnitt behandelt.

6.2.3 Verifikation von Workflow-Definitionen

Im folgenden wird beschrieben, wie die Korrektheit von Workflow-Definitionen mit Hilfe von Verifikationsalgorithmen geprüft wird. Zunächst werden einige Definitionen aufgestellt, auf deren Basis dann die Korrektheitskriterien formuliert werden.

Die Korrektheitskriterien stellen sicher, daß keine syntaktischen Fehler in der Workflow-Definition vorliegen. Wenn eine Workflow-Definition alle Korrektheitskriterien erfüllt, dann wird das Attribut *isChecked* auf true gesetzt. Dann können Workflow-Instanzen erzeugt werden, die auf der Workflow-Definition basieren und durch die Workflow-Engine zur Abarbeitung gebracht werden.

Allerdings kann mittels der Korrektheitskriterien nicht sichergestellt werden, daß keinerlei Fehler während der Abarbeitung des Workflows auftreten. Einerseits können verschiedene

Formen von technischen Fehlern zur Runtime auftreten, die entsprechend behandelt werden müssen.

Andererseits kann aber auch das Vorliegen semantischer Fehler in den Workflow-Definitionen nicht erkannt werden, da dazu im Allgemeinen das benötigte Wissen fehlt.

In [HOR98] werden weitergehende Aussagen über die Verifikation von Workflow-Definitionen getroffen, wobei insbesondere auch die Komplexität und Entscheidbarkeit einiger Verifikationsprobleme untersucht wird.

Somit können die Verifikationsalgorithmen den Workflow-Modellierer lediglich bei der Erstellung korrekter Workflow-Definitionen unterstützen, ihm aber nicht die Verantwortung für die Korrektheit der Workflow-Definition abnehmen.

6.2.3.1 Verifikation des Kontrollflusses

a) Definitionen

einfache Knoten: Aktivitätsknoten, Kommunikationsknoten

Kontrollknoten: Start-, End-, Split-, Join-, Loop-Start-, Loop-End-Knoten

Region-Anfangs-Knoten: Start-, Split-, Loop-Start-Knoten

Region-Ende-Knoten: End-, Join-, Loop-End-Knoten

Region: Paar aus einem Region-Anfangs-Knoten und einem zugehörigen Region-Ende-Knoten, also (Start, End), (Or-Split, All-Join), (And-Split, beliebiger Join), (Loop-Start, Loop-End) mit den Kontrollpfaden, die vom Region-Anfangs-Knoten zum Region-Ende-Knoten führen

globale Region: ist eine Start-End-Region

lokale Region: ist Split-Join-Region oder Loop-Region

Kontrollpfad: ist die Sequenz von einfachen Knoten und lokalen Regionen, die durch Transitionen verbunden sind. Er beginnt mit der ausgehenden Transition eines Region-Anfangs-Knotens und endet mit der eingehenden Transition des zugehörigen Region-Ende-Knotens.

Knoten liegt in einem Kontrollpfad: wenn er entweder selbst Bestandteil des Kontrollpfades ist, oder *in einer Region liegt*, die Bestandteil der Sequenz des Kontrollpfades ist[3]

Knoten liegt in einer Region: wenn er in einem Kontrollpfad liegt, der vom Region-Anfangs-Knoten der Region ausgeht[3]

parallel liegende Knoten: zwei Knoten liegen parallel, wenn sie nicht in demselben Kontrollpfad liegen

Loop-Transition: Transition, die vom Loop-End-Knoten zum Loop-Start-Knoten führt

Default-Transition: (einzige) ausgehende Transition eines Or-Splits ohne wertebasierte Bedingung, sie wird aktiviert, falls keine andere von diesem Or-Split ausgehende Transition aktiviert wird

normale Transition: Transition, die von einem Quell- zu eine, Ziel-Knoten führt und damit die Bestandteile eines Kontrollpfades definiert

Synchronisations-Transition: Transition, die eine bestimmte Reihenfolge in der Abarbeitung von Knoten in parallelen Kontrollpfaden festlegt

b) Verifikation

Die Verifikation des Kontrollflusses ist eine wechselseitige rekursive Verschachtelung der Prüfung von Regionen bzw. einfachen Knoten einerseits und sequenziellen Kontrollpfaden andererseits.

- ein **einfacher Knoten** ist korrekt, wenn
  - er genau eine eingehende und genau eine ausgehende normale Transition hat,
  - die ausgehende Transition keine wertebasierte Bedingung hat
- ein **Kontrollpfad** ist korrekt, wenn er entweder
  - aus einer Sequenz von korrekten einfachen Knoten besteht oder
  - aus einer Sequenz von korrekten einfachen Knoten und korrekten lokalen Regionen besteht

---

[3] Die rekursive Verschachtelung der Definitionen *Knoten liegt in einem Kontrollpfad* und *Knoten liegt in einer Region* terminiert, wenn ein Kontrollpfad erreicht ist, der keine weiteren (verschachtelten) lokalen Regionen enthält.

- eine **lokale Region** ist korrekt, wenn sie genau eine eingehende und genau eine ausgehende normale Transition hat und entsprechend ihres Typs korrekt ist
- eine **Loop-Region** ist korrekt, wenn
  - genau ein Kontrollpfad vom Loop-Start-Knoten zu einem Loop-End-Knoten führt,
  - die Loop-Transition von diesem Loop-End- zum Loop-Start-Knoten der Region führt und eine wertebasierte Bedingung besitzt
- eine **Split-Join-Region** ist korrekt, wenn mindestens zwei Kontrollpfade vom Split-Knoten ausgehen, alle Kontrollpfade an demselben Join-Knoten enden und
  - im Falle eines **Or-Splits**:
    - alle ausgehenden Transitionen, bis auf genau eine (Default-Transition), eine wertebasierte Bedingung haben
    - der zugehörige Region-Ende-Knoten ein All-Join ist
  - im Falle eines **And-Splits**:
    - keine der ausgehenden Transitionen eine wertebasierte Bedingung hat
    - der zugehörige Region-Ende-Knoten vom Typ All-Join, One-Join-Finish oder One-Join-Cancel ist
- eine **globale Region** ist korrekt, wenn sie keine eingehenden und keine ausgehenden Transitionen hat und die innere Struktur genau ein korrekter Kontrollpfad ist
- eine **Synchronisations-Transition** ist korrekt, wenn
  - sie keine wertebasierte Bedingung hat und
  - die zugehörigen Knoten *parallel liegen*, keine Start- oder End-Knoten sind, und nicht in Loop-Regionen liegen

6.2.3.2 Verifikation des Objektflusses

a) Definitionen

Typkompatibilität: Wird ein Objekt der Klasse *X* erwartet, und ein Objekt der Klasse *Y* geliefert, so ist die Typkompatibilität gewährleistet, wenn entweder *Y* identisch mit *X* ist, oder *Y* eine Subklasse von *X* ist

b) Korrektheitskriterien

Die verschiedenen Korrektheitskriterien für den Objektfluß sollen im folgenden kurz dargestellt werden.

- Component-Mappings zwischen zwei Object-Paths bzw. den zugehörigen Object-Specifications müssen jeweils ein Objekt, welches Daten bereitstellt und ein Objekt, das Daten benötigt, miteinander verbinden. Damit sind Component-Mappings nur bei folgenden Kombinationen von Roles der beteiligten Object-Specifications zulässig:

Role des Target-Objects / Role des Source-Objects	Activity-Input	Transition-Condition-Value	Retrieval-Condition-Value	Manipulation-Query-Input	Activity-Input-Objekte der Complex-Activity-Definition[4]	Activity-Definition	Application-Input	Activity-Output
Kontext: Workflow-Definition								
Activity-Output	x	x	x	x	x			
Retrieval-Output	x	x	x					
Constant	x	x	x	x	x			
Activity-Input-Objekte der Complex-Activity-Definition[4]	x	x	x	x	x			
Kontext: Basic-Activity-Definition								
Activity-Input						x	x	x
Application-Output								x

Tabelle 6: Zulässige Source- und Target-Object-Roles bei Component-Mappings

[4] Falls die (Sub-)Workflow-Definition zu einer Complex-Activity-Definition gehört, dann werden die Activity-Input- und Activity-Output-Objekte der Complex-Activity-Definition mit in den Objektfluß der Sub-Workflow-Definition einbezogen

- Bei Component-Mappings und den Parametern von Methods in Object-Paths muß die *Typkompatibilität* gewährleistet sein.
- Die in Object-Paths verwendeten Komponenten müssen in der Definition der jeweiligen Klasse enthalten sein. Wird beispielsweise ein Object-Path der Form *(P:Patient).name* verwendet, so muß *name* ein Attribut der Klasse *Patient* sein.
- Alle Objekte, die Daten bereitstellen (z.B. Ausgabeobjekte einer Aktivität), sollten weiterverwendet werden. Dazu muß entweder das Objekt selbst oder alle direkten Unterkomponenten mittels Component-Mappings anderen Objekten zugewiesen werden.
- Alle Objekte, die Daten benötigen (z.B. Eingabeobjekte einer Aktivität), sollten korrekt mit Daten versorgt werden. Dazu muß entweder das Objekt als ganzes oder alle direkten Unterkomponenten mittels Component-Mappings mit Daten versorgt werden. Ist dies nicht der Fall, dann wird bei der Verifikation eine Warnung[5] ausgegeben.
- Alle Objekte, die Daten benötigen, dürfen nicht mehrfach mittels Component-Mappings mit Daten versorgt werden. In Schleifen gilt die Ausnahme, daß ein zweites Component-Mapping zulässig ist, dabei muß eins ein *rückführendes Component-Mapping* sein (s. 4.2.3).

6.2.3.3 Verifikation komplexer Konstrukte

- eine **Basic-Activity-Definition** ist korrekt, wenn der zugehörige innere Objektfluß korrekt ist
- eine **Complex-Activity-Definiton** ist korrekt, wenn die zugehörige (Sub-) Workflow-Definition korrekt ist (unter Einbeziehung der Activity-Input- und -Output-Objekte der Complex-Activity-Definition)
- eine **Workflow-Definition** ist korrekt, wenn
  - sie eine korrekte globale Region enthält,
  - alle Knoten und Transitionen in der globalen Region liegen[6],

---

[5] d.h. der Workflow-Modellierer wird gewarnt, daß an dieser Stelle u.U. ein Fehler vorliegt und die Möglichkeit zur Änderung gegeben, die Workflow-Definition kann jedoch trotzdem als korrekt markiert werden

[6] zur Erkennung von Knoten und Transitionen, die nicht mit dem Kontrollfluß verbunden sind und deshalb bei der Verifikation der globalen Region unbemerkt bleiben

- alle Synchronisations-Transitionen korrekt sind,
- der Graph bestehend aus Knoten und Transitionen (bis auf Loop-Transitionen) zyklenfrei ist, dabei werden auch alle Component-Mappings (*rückführende Component-Mappings* in Schleifen) als Synchronisations-Transitionen zwischen den zugehörigen Knoten betrachtet,
- alle verwendeten Basic- und Complex-Activity-Definitions korrekt sind,
- allen Activity-Nodes eine verantwortliche Role zugeordnet ist
- der Objektfluß der Workflow-Definition korrekt ist

6.2.3.4 Basisobjekt-Änderungen und Korrektheit von Workflow-Definitionen

Wie aus der Definition der Korrektheit einer Workflow-Definition erkennbar ist, hängt diese auch von den verwendeten Basisobjekten ab. Das führt dazu, daß Änderungen an den Basisobjekten die Korrektheit von Workflow-Definitions beeinflussen können. Dies betrifft konkret die Klassen *Role* und *Activity-Definition*.

a) Abhängigkeit von Roles

Die Korrektheit einer Workflow-Definition hängt davon ab, daß die als Verantwortliche angegebenen Roles auch tatsächlich existieren.

Dies wird dadurch sichergestellt, daß bei Angabe des Verantwortlichen nur existierende Roles ausgewählt werden können. Außerdem wird verhindert, daß eine Role gelöscht werden kann, die in einer Workflow-Definition referenziert wird.

b) Abhängigkeit von Activity-Definitions

Das Löschen einer Activity-Definition würde dazu führen, daß alle Workflow-Definitionen, in denen diese verwendet wird, nicht mehr korrekt sind.

Dagegen haben Änderungen an einer Activity-Definition, bei der keine Input- oder Output-Objekte hinzugefügt oder entfernt werden, keine Auswirkungen auf die entsprechenden Workflow-Definitionen.

Eine mögliche Lösung dieser Problematik besteht darin zu verhindern, daß verwendete Activity-Definitions gelöscht werden. Änderungen an Input- oder Output-Objekten sollten erst nach Rückfrage erlaubt werden, wobei dann alle betroffenen Workflow-Definitionen neu verifiziert werden müssen.

6.3 Implementierung

6.3.1 Grundlagen

Als Programmiersprache für die Implementierung des Workflow-Editors wurde C++ gewählt, da sie der Aufgabenstellung angemessen erschien.

Die Implementierung erfolgt auf der Plattform Microsoft Windows NT/2000 mit der Entwicklungsumgebung Microsoft Visual C++. Diese bietet mit der Klassenbibliothek Microsoft Foundation Classes (MFC) eine für die Belange des Workflow-Editors ausreichende Unterstützung an, insbesondere was die ergonomische Gestaltung der Benutzeroberfläche entsprechend den Windows-Standards und die Datenbankschnittstelle betrifft.

Wünschenswert wäre gewesen, die grafische Bearbeitung der Workflow-Definitionen basierend auf bereits vorhandenen Klassenbibliotheken zu realisieren. Allerdings konnte keine solche gefunden werden, die eine ausreichende Flexibilität und Erweiterbarkeit geboten hätte und damit mit vertretbarem Aufwand wiederverwendbar gewesen wäre.

Somit wurden die Funktionen zur grafischen Darstellung und Bearbeitung innerhalb der Klassen des Workflow-Editors implementiert.

6.3.2 Objektmodell

Das Objektmodell des Workflow-Editors ist eng an das UML-Buildtime-Modell, wie in Kapitel 5 beschrieben, angelehnt. Die zusätzliche Funktionalität, die für die grafische Bearbeitung und Darstellung, sowie für die Verfikation benötigt wird, wird in spezialisierten Klassen implementiert, die von den Workflow-Buildtime-Klassen der Datenbank-Zugriffsschicht (s. nächster Abschnitt) abgeleitet sind.

Zusätzliche Klassen werden für die Dialoge zur Bearbeitung der verschiedenen Objekte angelegt, diese sind von den entsprechenden MFC-Klassen abgeleitet.

Das Objektmodell verwendet die allgemein übliche Dokument-Ansicht-Struktur, bei der die Daten und Funktionen des Dokuments - hier konkret der Workflow-Definition - getrennt von den Funktionen der grafischen Ansicht bzw. Benutzeroberfläche in zwei Klassen implementiert werden. Diese Architektur erhöht durch die stärkere Modularisierung die Übersichtlichkeit und bietet Vorteile im Hinblick auf Erweiterungen, zum Beispiel der

denkbaren Hinzufügung einer alternativen Darstellungsform der Dokumentdaten, wie in [Kru96] ausgeführt.

Außerdem wird als Basis für die Darstellung der Workflow-Definitionen ein Windows-übliches *Multiple Document Interface* (MDI) verwendet, so daß mehrere Workflow-Definitionen gleichzeitig zur Bearbeitung geöffnet sein können.

Eine Übersicht über die Klassen des Workflow-Editors ist im Anhang zu finden.

6.3.3 Speicherung der Objekte

Die den Workflow-Definitionen zugrundeliegenden Daten liegen stark strukturiert in Objekten vor. Diese Datenstrukturen müssen persistent gespeichert und den anderen Komponenten des Systems zugängig gemacht werden.

Für die Speicherung der Daten kommen einfache Dateien oder Datenbanksysteme in Frage. Die Speicherung in Datenbanksystemen bietet vielfältige Vorteile, insbesondere im Hinblick auf Zuverlässigkeit und Sicherheit. Der Autorisierungsmechanismus des Datenbanksystems kann genutzt werden, um Änderungen an Workflow-Definitionen nur durch berechtigte Personen zuzulassen. Mit Hilfe von Transaktionen kann die Konsistenz der gespeicherten Daten sichergestellt werden.

Für die Speicherung der Objekte würde sich ein objektorientiertes Datenbanksystem anbieten. Allerdings stand im Rahmen des Projekts kein OODBS zur Verfügung, bei dem insbesondere auch im Hinblick auf die Langfristigkeit des Projekts eine ausreichende Unterstützung seitens des Herstellers gewährleistet schien.

Die Speicherung von Objekten in relationalen Datenbanksystemen ist ein schon seit längerer Zeit verwendetes Verfahren. Aus den zur Verfügung stehenden relationalen Datenbanksystemen wurde IBM DB/2 ausgewählt, das hinsichtlich Performance, Stabilität und langfristiger Unterstützung durch den Hersteller als geeignet erscheint.

Für die Anbindung des Editors an die Datenbank erschienen prinzipiell zwei Wege möglich:

- die direkte Abfrage der Daten aus der Datenbank in dem Moment, wenn sie benötigt werden und die unmittelbare Ausführung von Änderungsoperationen in der Datenbank
- die Einführung einer Datenbank-Zugriffsschicht, die
  - aus der Datenbank gelesene Daten zwischenspeichert

- Änderungen an den Daten zwischenspeichert und auf explizite Anweisung zurückschreibt
- die Datenbankzugriffe kapselt und
- nach außen hin eine objektorientierte Sicht auf die Daten bereitstellt

Im Hinblick auf die interaktive Oberfläche des Workflow-Editors erschien der erste Ansatz aus Performancegründen sehr problematisch, da insbesondere die grafische Darstellung ständige Abfragen der Daten benötigt. Daher wurde der zweite Ansatz gewählt.

Allerdings ergeben sich in solch einer Zugriffschicht, die eine relationale Sicht auf die Daten in eine objektorientierte Sicht transformiert, und umgekehrt die in Objekten vorliegenden Daten wieder in relationalen Tabellen speichert, eine gewisse Anzahl von Problemklassen, wie sie in [Kel98] umrissen sind. Diese verursachen einen erheblichen Konzeptions- und Implementierungsaufwand.

Da die Funktionalität der Datenbank-Zugriffsschicht auch von anderen Komponenten, wie der Workflow-Engine und dem Adaptation-Agent, benötigt wird, wurde sie als gemeinsam zu verwendendes Modul identifiziert und im Rahmen von [Die00] konzipiert und implementiert. Dabei wurde allerdings auf Grund der hohen Komplexität zunächst nur der Kontrollfluß-Teil des UML-Modells umgesetzt.

6.3.4 Algorithmus zur Kontrollfluß-Verifikation

Der Algorithmus zur Kontrollfluß-Verifikation dient zur Prüfung einer Workflow-Definition auf Erfüllung der unter 6.2.3.1 aufgeführten Korrektheitsbedingungen für den Kontrollfluß.

Diese Aufgabe wird in drei Teilalgorithmen gegliedert, die sich gegenseitig aufrufen können:

- Prüfung von Regionen
- Prüfung von Kontrollpfaden
- Prüfung einzelner Knoten

Die Zusammenhänge zwischen diesen sind in Abbildung 22 dargestellt, ihre ausführliche Darstellung befindet sich im Anhang.

Jeder Teilalgorithmus arbeitet den entsprechenden Teil der Workflow-Definition sequentiell ab, analog der Abarbeitungsreihenfolge zur Runtime.

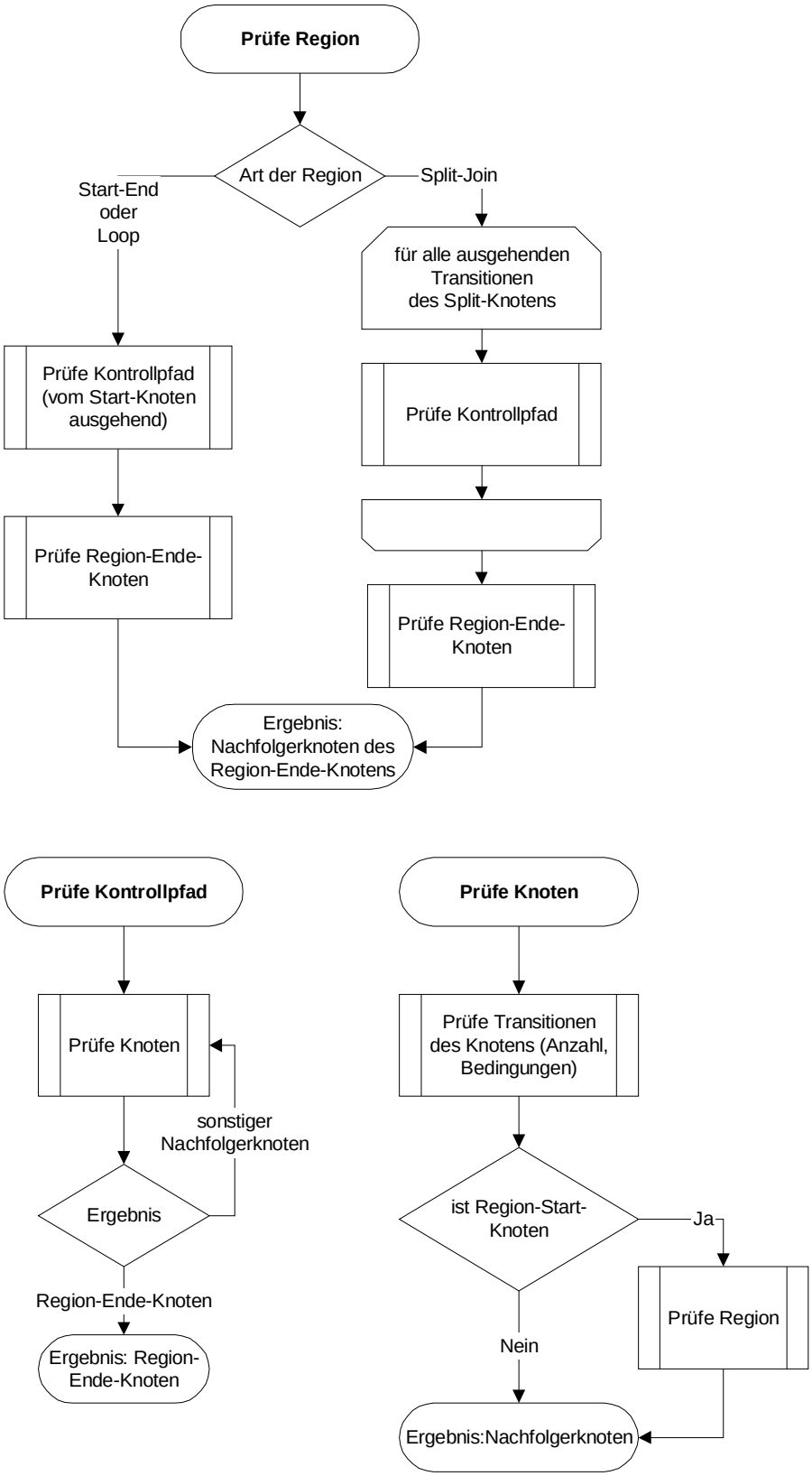


Abbildung 22: Struktur des Algorithmus zur Kontrollfluß-Verifikation

Die Verifikation einer Workflow-Definition beginnt immer mit der Prüfung der enthaltenen Start-End-Region. Innerhalb der Prüfung dieser *Region* wird der vom Start-Knoten

ausgehende *Kontrollpfad* verifiziert. Die Verifikation des Kontrollpfades beinhaltet dann wiederum die Prüfung der einzelnen *Knoten*. Ist ein solcher zu prüfender Knoten eine Region-Start-Knoten, dann wird an der Stelle die Prüfung dieser Region durchgeführt.

Auf diese Weise wird über eine rekursive Verschachtelung der Prüfungen von Regionen, Kontrollpfaden und Knoten die entsprechend verschachtelte Workflow-Definition geprüft.

6.3.5 Numerierung der Knoten

An verschiedenen Stellen im Workflow-System ist es nötig festzustellen, wie die Lage zweier gegebener Knoten zueinander ist, d.h. ob sie parallel liegen bzw. in welcher Reihenfolge sie im Kontrollfluß liegen. Diese Aussage zu treffen, ist normalerweise relativ aufwendig, da jedes Mal der Kontrollflußgraph vom Startknoten aus abgearbeitet werden müsste, bis die Positionen der beiden Knoten ermittelt sind.

Um diesen Aufwand zu verringern, wird während der Verifikation des Kontrollflusses jedem Knoten eine Ordnungsnummer zugeteilt, die derart zusammengesetzt ist, daß durch den direkten Vergleich von zwei Ordnungsnummern festgestellt werden kann, wie die Lage der zugehörigen Knoten zueinander ist. Um diese Aussage zu ermöglichen, müssen in den Ordnungsnummern Informationen einerseits über die Reihenfolge der Knoten und andererseits über die Zugehörigkeit zu ggf. parallelen Kontrollpfaden abgelegt sein.

6.3.5.1 Zusammensetzung der Ordnungsnummern

Jede Ordnungsnummer besteht aus einem Array von Integer-Werten. Die einzelnen Einträge im Array sind von 0 an aufsteigend numeriert. In jedem ungeradzahligen Eintrag wird die Nummer des Knotens innerhalb des Kontrollpfades abgelegt, in jedem geradzahligen Eintrag die Nummer des Kontrollpfades selbst.

In den nachfolgenden Abbildungen ist die Vergabe der Ordnungsnummern für die verschiedenen Kontrollfluß-Konstrukte dargestellt. Die Ordnungsnummern sind im Format x_0 . x_1 . ... . x_n dargestellt, wobei x_i die Einträge des Arrays sind, i ist die Nummer des Eintrags.

Die Numerierung beginnt immer beim Startknoten mit der Ordnungsnummer 0.0. Die Knoten zwischen Start- und End-Knoten werden entsprechend der folgenden Grafiken numeriert, die Ordnungsnummer des End-Knotens ergibt sich dann aus der Ordnungsnummer seines Vorgängerknotens.

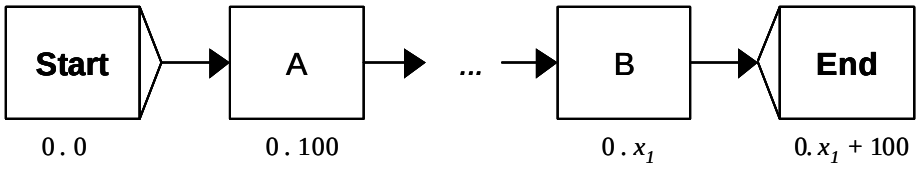


Abbildung 23: Ordnungsnummern bei Start-End-Regionen

In einem Kontrollpfad werden die Knoten mit einem 100-er Abstand numeriert. Das heißt, hat ein Knoten die Ordnungsnummer x_0 . ... . x_n, dann erhält der Nachfolgerknoten in dem Kontrollpfad die Ordnungsnummer x_0 . ... . x_n + 100. Durch den Abstand bei der Numerierung wird sichergestellt, daß beim Einfügen von Knoten durch den Adaptation-Agenten nicht alle nachfolgenden Knoten neu numeriert werden müssen. Dabei wird davon ausgegangen, daß zwischen zwei bestehenden Knoten nicht mehr als 99 Knoten eingefügt werden.

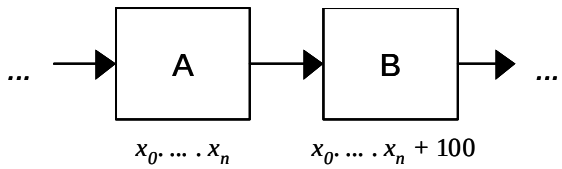


Abbildung 24: Ordnungsnummern in einem Kontrollpfad

Bei Split-Join-Regionen, wie in Abbildung 25, wird von der Nummer des Split-Knotens der Region ausgehend zunächst ein Numerierungseintrag hinzugefügt, welcher die ausgehenden Kontrollpfade des Split-Knotens unterscheidet (im Bild: 0 ... *m*).

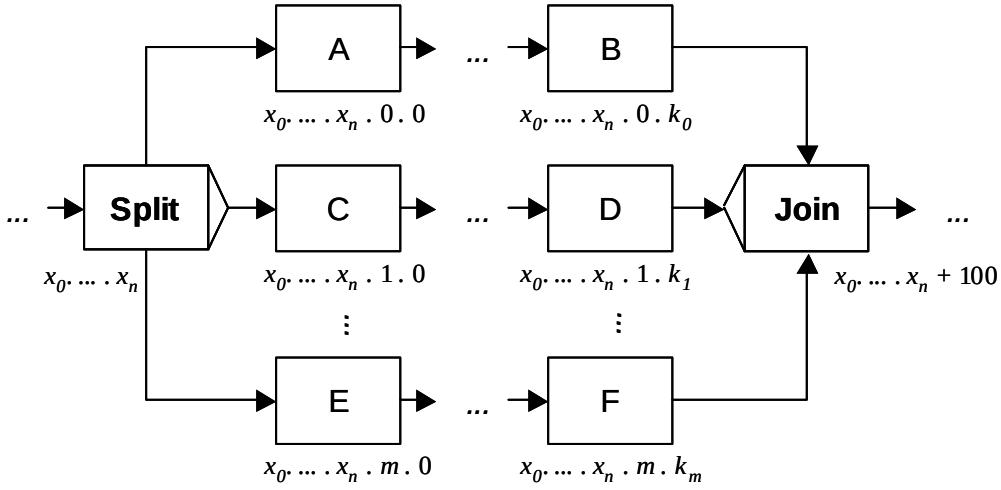


Abbildung 25: Ordnungsnummern bei Split-Join-Regionen

Innerhalb der einzelnen Kontrollpfade werden die Knoten dann in einem weiteren Numerierungseintrag wiederum in ihrer Reihenfolge numeriert (im Bild: x_0 . ... . x_n . i . 0 bis x_0 . ... . x_n . i . k_i , wobei i die Nummer des Kontrollpfades ist).

Die Ordnungsnummer des Join-Knotens ergibt sich direkt aus der des Split-Knotens.

Bei der in Abbildung 26 dargestellten Loop-Region erfolgt die Numerierung analog zu Split-Join-Regionen, um eine konsistente Bearbeitung der verschiedenen Region-Arten durch den Numerierungsalgorithmus zu erreichen. Die Ordnungsnummer des Loop-End-Knotens ergibt sich wiederum direkt aus der des Loop-Start-Knotens

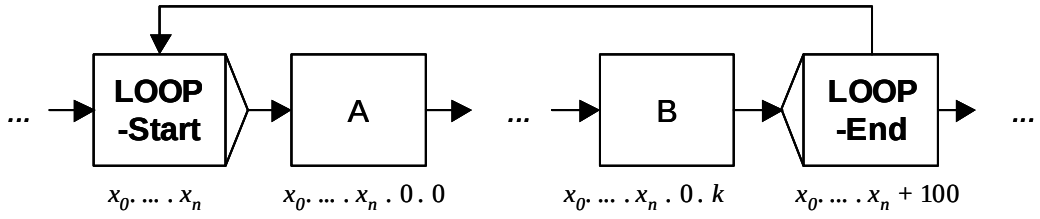


Abbildung 26: Ordnungsnummern bei Loop-Regionen

Aus diesem Numerierungsschema ist ersichtlich, daß für jede Verschachtelungsebene von Regionen zwei Einträge zu der Ordnungsnummer hinzugefügt werden, je ein Eintrag für die Nummer des Kontrollpfades und für die Nummer des Knotens im Kontrollfpad.

6.3.5.2 Verwendung von Ordnungsnummern

a) Vergleich von Ordnungsnummern

Der Vergleich von zwei Knoten x, y anhand ihrer Ordnungsnummern x_0 . ... . x_m bzw. y_0 . ... . y_n erfolgt, um festzustellen welcher der drei möglichen Fälle vorliegt:

- Knoten x liegt vor Knoten y
- Knoten y liegt vor Knoten x
- die Knoten x und y sind nebenläufig

Der Algorithmus zum Vergleich der Ordnungsnummern ist in Abbildung 27 dargestellt.

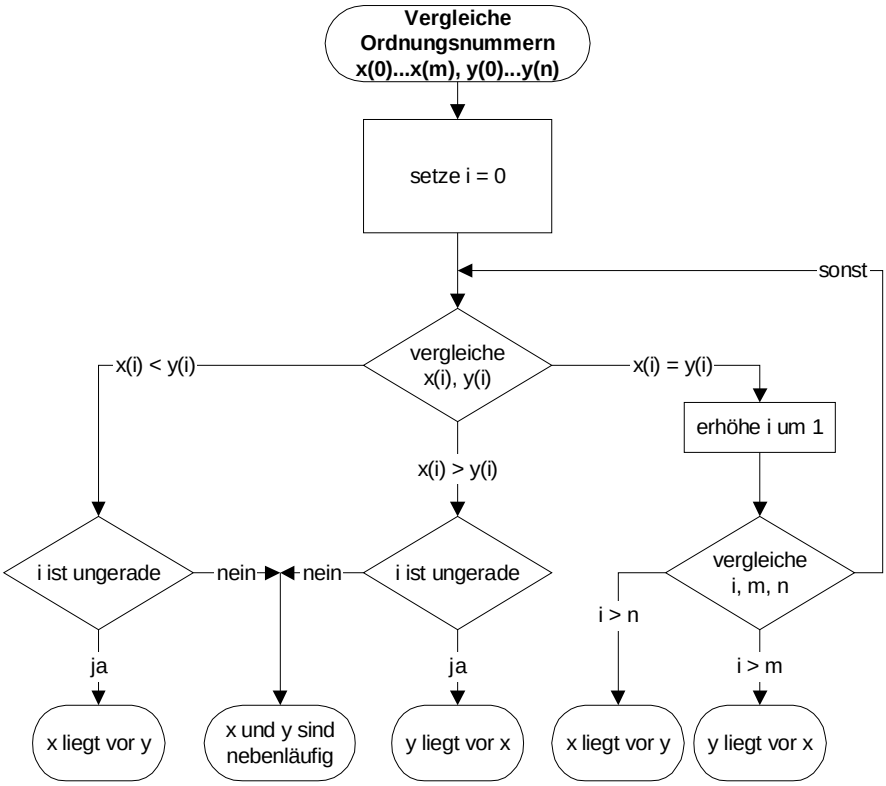


Abbildung 27: Vergleich von Ordnungsnummern

b) Suchen von zugehörigen Knoten

Eine weitere Möglichkeit der Verwendung von Ordnungsnummern ist das Auffinden von Knoten, die in einer bestimmten Beziehung zu einem gegebenen Knoten stehen. Mögliche Beziehungen sind z.B.:

(1) der Vorgängerknoten zum gegebenen Knoten

(2) der Nachfolgerknoten zum gegebenen Knoten

(3) zu einem gegebenen Split- bzw. Loop-Start-Knoten zugehöriger Join- bzw. Loop-End-Knoten

(4) zu einem gegebenen Join- bzw. Loop-End-Knoten zugehöriger Split- bzw. Loop-Start-Knoten

Beispielhaft soll hier der Algorithmus für die Beziehungen (2) und (3) angeführt werden.

Gegeben sei der Knoten x mit der Ordnungsnummer $x_0 \,.\, \ldots \,.\, x_n$. Gesucht ist der Knoten y.

Sei M die Menge aller Ordnungsnummern $y_0 \,.\, \ldots \,.\, y_n$ (mit genau n Einträgen), für die gilt $y_n < x_n$. Die Ordnungsnummer des gesuchten Knotens ist dann diejenige aus der Menge M, bei der y_n maximal ist. Damit ist der Knoten y ermittelt.

Analog zu dem Beispiel sind die Knoten mit den Beziehungen (1) und (4) ermittelbar.

Der Vorteil der Verwendung der Ordnungsnummern besteht wiederum darin, daß nicht der gesamte Kontrollflußgraph durchlaufen werden muß, sondern die Knoten durch Auffinden der entsprechenden Ordnungsnummern ermittelt werden können.

6.3.5.3 Einschränkungen der Ordnungsnummern

Die Vergabe der Ordnungsnummern orientiert sich ausschließlich an den spezifizierten normalen Transitionen. Unberücksichtigt bleiben dabei Abhängigkeiten in der Abarbeitungsreihenfolge, die durch Synchronisations-Transitionen bzw. Objektflußkanten (d.h. Komponenten-Zuweisungen) hervorgerufen werden.

Das heißt, wenn der Vergleich zweier Ordnungsnummern ergibt, daß die Knoten parallel liegen, so sind dabei diese Abhängigkeiten nicht berücksichtigt.

6.3.6 Export der grafischen Workflow-Darstellung

Der Export der grafischen Workflow-Darstellung erfolgt über die Windows-eigene Zwischenablage. Als Format wird der Windows-Standard "Enhanced Metafile" verwendet, der zur Darstellung von Vektorgrafiken dient.

Damit ist es möglich, die grafische Darstellung von Workflow-Definitionen direkt über die Zwischenablage in verschiedene Windows-Programme zu übernehmen und dort weiter zu bearbeiten oder auszudrucken.

6.3.7 Realisierter Umfang der Implementierung

Die Implementierung des Workflow-Editors umfasst die Bearbeitung von Basisobjekten, die grafische Bearbeitung des Kontrollflusses der Workflow-Definitionen und die Kontrollfluß-Verifikation.

Die Implementierung der Objektfluß-Modellierung wurde nicht vorgenommen, da einerseits die Konzeption des Objektflusses, insbesondere bezüglich externer Datenzugriffe erst im Rahmen des CORBA Management Layers vervollständigt werden soll, andererseits aber auch die Implementierung der Objektfluß-Klassen in der Datenbank-Zugriffsschicht noch nicht vorgenommen wurde, welche zur Speicherung der Daten aber notwendig ist. Die Integration der Objektfluß-Modellierung in den Workflow-Editor wurde aber in dessen Objektmodell bereits vorgesehen.

7 Zusammenfassung

7.1 Ergebnisse der Arbeit

Zunächst wurde in Zusammenarbeit mit anderen Mitarbeitern des *AgentWork*-Projekts ein Workflow-Modell entwickelt, welches Ansätze aus dem $Adept_{flex}$-Modell aus [RD98] beinhaltet. Dabei wurden die speziellen Anforderungen an dynamisch zu adaptierende Workflow-Definitionen berücksichtigt.

Darauf aufbauend wurden im Rahmen dieser Arbeit Details weiter ausgearbeitet, insbesondere spezielle Probleme des Objektflusses betreffend, wie den Zugriff auf externe Datenquellen. Dabei entstand die Erkenntnis, daß der Objektfluß, besonders der Zugriff auf externe Datenquellen, bereits in der Modellierung relativ komplex ist. Diese Problematik soll deshalb im Rahmen des CORBA Management Layers getrennt fortgeführt werden.

Anschließend erfolgte die Konzeption der Benutzeroberfläche des Workflow-Editors, wobei sich aus dem gewählten Ansatz die Notwendigkeit ergab, eine umfassende Verifikation der Workflow-Definitionen durchzuführen.

Als Grundlage für die Verifikation wurde die Korrektheit einer Workflow-Definition definiert. Darauf aufbauend entstand ein Algorithmus zur Kontrollfluß-Verifikation.

Als Ergebnis der Arbeit wurde ein Workflow-Editor prototypisch implementiert. Dieser ermöglicht die Modellierung der Workflow-Basisobjekte sowie die grafische Bearbeitung des Kontrollflusses von Workflow-Definitionen.

7.2 Ausblick

Die objektorientierte Modellierung und Implementierung des Workflow-Editors kann als Ausgangspunkt für eine Vervollständigung der Funktionalität dienen.

Dabei ist sicherlich in erster Linie die Implementierung der Objektflußmodellierung und -verifikation wünschenswert.

Der bestehende Verifikationsalgorithmus für den Kontrollfluß könnte um einen semantischen Teil erweitert werden, der die Aktivitätsklassen und Regeln aus der Regelbasis verwendet, um einige semantische Fehler bereits zur Buildtime zu erkennen.

Der Export und Import von Workflow-Definitionen und Basisobjekten mit Dateien wäre eine andere denkbare Erweiterung. Damit könnten diese zwischen verschiedenen Installationen des Workflow-Systems ausgetauscht werden, was bisher nur indirekt über Datenbank-Dumps möglich ist. Ob diese Austausch-Möglichkeit auch für die Interoperabilität mit anderen Workflow-Systemen genutzt werden kann, ist allerdings fragwürdig, da im *AgentWork*-System zur Verwirklichung der dynamischen Adaptation spezielle Anforderungen an die Struktur der Workflow-Definitionen gestellt werden.

Wünschenswert wäre auch eine Versionsverwaltung von Workflow-Definitionen und Basisobjekten. Damit könnten geänderte Versionen von Workflow-Definitionen erstellt werden, während noch Instanzen von der bisherigen Workflow-Definition erzeugt werden können. Außerdem würden Änderungen an Aktivitätsdefinitionen neue Versionen der Workflow-Definitionen erzeugen, in denen diese verwendet werden, statt diese als "zu verifizierend" zu kennzeichnen.

Außerdem wäre die Weiterverwendung der grafischen Workflow-Darstellung möglich, um einerseits den aktuellen Status einer Workflow-Instanz, d. h. die bereits durchlaufenen und die aktiven Knoten und Transitionen, und andererseits auch die durch dynamischen Adaptationen veränderten Workflows darzustellen.

Literaturverzeichnis

[AAH98] Adam, Nabil R.; Atluri, Vijayalakshmi; Huang, Wei-Kuang: *Modeling and Analysis of Workflows Using Petri Nets*, Journal of Intelligent Information Systems, Vol. 10, No. 2: 131-158, Kluwer Academic Publishers, Boston 1998

[Die00] Dietzsch, Alexander: *Konzeption und Implementierung einer Workflow-Engine für dynamische Workflow-Adaptationen*, Diplomarbeit, Universität Leipzig, 2000

[Grei00] Greiner, Ulrike: *Ein wissensbasierter Agent zur ereignisorientierten Adaptation von Workflows*, Diplomarbeit, Universität Leipzig, 2000

[Har87] Harel, D.: *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, Vol.8 S. 231-274, 1987

[HOR98] ter Hofstede, A.H.M.; Orlowska, M. E.; Rajapakse, J.: *Verification problems in conceptual workflow specifications*. Data & Knowledge Engineering 362: 239-256, Elsevier Science B.V.,1998

[HR99] Härder, Theo; Rahm, Erhard: *Datenbanksysteme. Konzepte und Techniken der Implementierung*, Springer, Berlin, 1999

[JBS97] Jablonski, Stefan; Böhm, Markus; Schulze, Wolfgang: *Workflow-Management. Entwicklung von Anwendungen und Systemen*, 1. Auflage, dpunkt-Verlag, Heidelberg, 1997.

[Jöd97] Jödecke, Enrico: *Konzeption und Implementierung eines datenbankgestützten Dokumentenstruktur-Editors und -Generators*, Universität Leipzig, 1997

[Kel98] Keller, Wolfgang: *Object-Relational Access Layers. A Roadmap, Missing Links and More Patterns*, EA Generali, 1998 http://ourworld.compuserve.com/homepages/WofgangWKeller

[KLW95] Kifer, Michael; Lausen, Georg; Wu, James*: Logical Foundations of Object-Oriented and Frame-Based Languages*. Journal of the ACM, 42(4):741-843, 1995

[Kru96] Kruglinski, David: Inside Visual C++, deutsche Ausgabe, Microsoft Press Deutschland, Unterschleißheim, 1996

[LP94] Löffler, M.; Pfreundschuh, M.: *Integratives Konzept zur Behandlung hochmaligner Non-Hodgkin-Lymphome.* Studienprotokoll der Studie B, Leipzig, 1994

[LR00] Leymann, Frank; Roller, Dieter: *Production Workflow. Concepts and Techniques,* Prentice-Hall, New Jersey, 2000

[Mei00] Produktinformation über das Medical Control Center® der Meierhofer AG, *http://www.meierhofer.de/fs_p1.htm*, 2000

[MM99] May, Wolfgang; Marrón, Pedro José: *Florid. How to Write F-Logic Programs in Florid*, Institut für Informatik, Universität Freiburg, 1999

[Mül00] Müller, Robert: *Event-Oriented Dynamic Adaptation of Workflows. Model, Architecture and Implementation.* Dissertation, Institut für Informatik. Universität Leipzig, 2000

[OMG98] *CORBAservices: Common Object Services Specification*, OMG, 1998

[OMG99] *The Common Object Request Broker: Architecture and Specification*, Rev. 2.3.1, OMG, 1999

[OMG00] *Workflow Management Facility Specification*, V1.2, OMG, 2000

[RD98] Reichert, Manfed; Dadam, Peter: *ADEPT$_{flex}$ - Supporting Dynamic Changes of Workflows Without Loosing Control*, University of Ulm, Dept. Databases and Information Systems, 1998

[Rit97] Ritter, N.: *DB-gestützte Kooperationsdienste für technische Entwurfsanwendungen*, Universität Kaiserslautern, Dissertation DISBIS 33, infix-Verlag, 1997

[UML99] *Unified Modeling Language V1.3*, OMG, 1999

[VB96] Vossen, Gottfried; Becker, Jörg*: Geschäftsprozeßmodellierung und Workflow-Management. Modelle, Methoden, Werkzeuge*, 1. Auflage, Internat. Thomson Publ., Bonn, 1996

[WRM95] Hollingsworth, David: *Workflow Management Coalition - The Workflow Reference Model*, 1995

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A: F-Logic-Syntax

Im folgenden werden aus [KLW95] ausgewählte F-Logic-Konstrukte aufgeführt und kurz erläutert. Die Auswahl berücksichtigt dabei insbesondere die für Queries benötigten Konstrukte. Die Modellierung von Retrieval-Queries auf dieser Basis wird im Anhang B beschrieben.

a) Molekulare Formeln

Molekulare Formeln sind die einfachste Form von Formeln in F-Logic.

Objektbezeichner sind Objektvariablen oder zusammengesetzte Ausdrücke aus Funktionssymbolen und Objektvariablen.

Aus den Konstrukten für *molekulare Formeln* wurden die folgenden ausgewählt:

- eine *is-a*-Behauptung der Form *O:C*, wobei *O* eine Objektvariable ist und *C* der Name einer Klasse
- ein *Objekt-Molekül* der Form *O[Methoden-Ausdruck*[7]*]*, wobei O ein Objektbezeichner ist. Ein *Methoden-Ausdruck* kann die folgenden Formen haben:
  - skalarer Ausdruck ($k \geq 0$): $ScalarMethod@Q_1, ..., Q_k -> T$
  - mengenwertiger Ausdruck ($l, m \geq 0$): $SetMethod@R_1, ..., R_l ->> \{S_1, ..., S_m\}$

O:C sagt aus, daß das Objekt *O* aus der Klasse *C* oder aus einer der Unterklassen stammt.

ScalarMethod und *SetMethod* sind Bezeichner für Methoden. *T* und S_i sind Objektbezeichner für die Ausgabe-Objekte der Methoden, wenn diese auf dem Objekt *O* mit den Parametern $Q_1, ..., Q_k$ bzw. $R_1, ..., R_l$ aufgerufen werden.

---

[7] Konstrukte der Form *O[Methoden-Ausdruck1; Methoden-Ausdruck2; ...]* können, wie in [KLW95] S.14 aufgeführt, äquivalent durch die Konjunktion $O[Methoden\text{-}Ausruck1] \wedge O[Methoden\text{-}Ausruck2] \wedge ...$ ausgedrückt werden und sind deshalb an dieser Stelle nicht explizit berücksichtigt

Ein Sonderfall sind Methoden ohne Parameter (*k = 0* bzw. *l = 0*), für diese wird "@" weggelassen, so daß z.B. *P[Method->Val]* statt *P[Method@->Val]* geschrieben wird. Diese Methoden ohne Parameter stellen effektiv Attribute dar.

b) Komplexe Formeln

F-Formeln werden aus einfacheren F-Formeln unter Verwendung von logischen Verknüpfungen zusammengesetzt:

- molekulare Formeln sind F-Formeln
- $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg \varphi$ sind F-Formeln, falls φ und ψ F-Formeln sind

c) Queries

Queries werden in F-Logic in der Form *? - Q* ausgedrückt, wobei Q eine komplexe Formel ist.

Anhang B: Retrieval-Queries mit F-Logic

In folgenden wird erläutert, wie die in Anhang A beschriebenen F-Logic-Konstrukte syntaktisch umgeformt werden können, um sie in den Klassen des UML-Buildtime-Modells zu speichern und damit die Verwendung bei Retrieval-Queries zu ermöglichen.

a) Grundprinzip der Repräsentierung von F-Logic-Queries im Buildtime-Modell

Die zentrale Klasse für die Umsetzung ist die Klasse *Retrieval-Query*. Eine Instanz dieser Klasse definiert, zusammen mit den assoziierten Objekten, eine externe Datenabfrage.

Das Grundprinzip der syntaktischen Umsetzung der F-Logic-Konstrukte ist die Ersetzung der Infix-Notationen (wie $A -> B$, $C \vee D$, $E \wedge F$) durch Prefix-Notationen (wie *is(A, B), or(C, D), and(E, F)*).

Nach Durchführung dieser syntaktischen Ersetzung sind F-Logic-Queries syntaktisch als verschachtelte Methodenaufrufe und Objektpfade dargestellt, so daß sie als Methoden in *Object-Paths* gespeichert werden können.

Es wird eine F-Logic-Klasse *logic* definiert, welche die oben angeführten Prefix-Notationen als Methoden beinhaltet. Es existiert dazu im gesamten Workflow-System genau eine *Object-Specification* mit dem *object-name* "L" und der *class* "logic".

Zur Darstellung der strukturierten Daten der UML-Buildtime-Klassen *Object-Path, Object-Specification* und *Method* im Text soll die folgende Notation dienen:

- Für eine Methode mit dem *name* "Method" und den Parametern Q_1, ... Q_n usw. wird *.Method(*Q_1*, ...,* Q_n*)* geschrieben, Q_1 ... Q_n sind dabei *Object-Paths*. Bei parameterlosen Methoden (Attributen) werden die Klammern weggelassen.
- Für einen Object-Path, ausgehend von einer Object-Specification mit dem *object-name* "O" und der *class* "C" wird zunächst *O:C* geschrieben. Dann werden ggf. die entspre-

chenden Methoden in ihrer Reihenfolge angehängt, also beispielsweise *O:C.MethodA(P_1, ..., P_n).MethodB(Q_1, ..., Q_m)*

b) Repräsentierung von Objektbezeichnern

Das Konzept der Objektbezeichner wird mittels der Klasse *Object-Path* umgesetzt. Ein Objektbezeichner besteht, wie oben angeführt, aus einer Objektvariablen oder aus einem zusammengesetzten Ausdruck aus Funktionssymbolen (Methoden) und Objektvariablen.

Objektvariablen werden mittels *Object-Specifications* repräsentiert. Der Name der Variablen wird im Attribut *name* abgelegt, die Klasse des enthaltenen Objekts im Attribut *class*. Eine Besonderheit ist dabei, daß für jede verwendete Objektvariable auch die Klasse des enthaltenen Objekts angegeben werden muß, d.h. jede Objektvariable ist gleichzeitig eine *is-a*-Behauptung (s.u.).

Methoden werden mittels der UML-Buildtime-Klasse *Method* repräsentiert. Der Name der Methode wird im Attribut *name* gespeichert, die Parameter werden mittels der Assoziation *parameters* als *Object-Paths* angegeben.

c) Repräsentierung einer is-a-Aussage

Die *is-a*-Behauptung *O:C* wird mittels einer *Object-Specification* dargestellt. Der Objektname *O* wird dabei im Attribut *object-name* abgelegt, der Klassenname *C* im Attribut *class*.

d) Repräsentierung von Objekt-Molekülen

d1) Skalare Methoden

Objekt-Moleküle, die Methoden der Form

O[ScalarMethod@Q_1, ..., Q_k -> T]

Beinhalten, werden syntaktisch umgeformt mittels einer Methode "is" der Klasse *logic*. Der Methodenname "is" verdeutlicht die Semantik des Vergleichsoperators ->, den die Methode ersetzt.

Die entsprechende Struktur im Buildtime-Modell ist

L:logic.is(O:C.ScalarMethod(Q_1, ..., Q_k), T).

Da sie syntaktisch einem F-Logic-Methodenaufruf entspricht, kann sie in einem Objekt der Klasse *Object-Path* gespeichert werden.

Die Methode *is* hat die Signatur *logicResult is(Object, Object)*. Der Ergebnistyp *logicResult* ist dabei eine spezieller Typ, der ausschließlich von den Methoden der Klassen *logic* und *objectSet* (s.u.) verarbeitet werden kann. Damit wird die korrekte Konstruktion von Queries erzwungen, d.h. es wird vermieden, daß Werte, die nur innerhalb der Query-Evaluierungs-Engine bereitstehen, als Parameter von Methoden der Objekte angegeben werden können.[8]

d2) Mengenwertige Methoden

In F-Logic existiert zu jeder Klasse *C* implizit eine Mengenklasse, deren Instanzen jeweils eine Menge von Objekten der Klasse *C* enthalten können. Im folgenden soll ein erster Ansatz für die Unterstützung von Mengen im Buildtime-Modell vorgestellt werden.

Es wird eine Klasse *objectSet* definiert, welche die Funktionalität enthält, die für Mengen von Objekten benötigt wird. Dazu gehören:

- Zugriff auf einzelne Objekte in der Menge (z.B. über einen Iterator)
- Bildung von Untermengen von Objekten, wobei die Objekte in der Untermenge anhand einer gegebenen Bedingung ausgewählt werden

Diese Funktionalität kann mit Hilfe von Methoden der Klasse *objectSet* angeboten werden, diese dient als generische Oberklasse aller Mengenklassen.

Ist die Verwendung von Mengen von Objekten einer Klasse *C* erwünscht, so muß eine Mengenklasse *CSet* existieren, die Unterklasse der Klasse *objectSet* ist. Durch das explizite Anlegen der Klasse wird eine zusätzliche Flexibilität beim Umgang mit Objektmengen in den CORBA-Repräsentationen der F-Logic-Klassen ermöglicht. Zu jeder F-Logic-Mengenklasse kann dann eine spezielle CORBA-Klasse erstellt werden, welche die Mengenoperationen entsprechend den zugrundeliegenden Datenquellen implementiert.

---

[8] Das ist deshalb nötig, da die Auswertung von Querys im allgemeinen Fall in externen Engines stattfindet, wie z.B. in einer SQL-Datenbank. Es ist dann nicht möglich, auf die temporären Zwischenwerte der Auswertung zuzugreifen.

Damit entspricht ein Objekt-Molekül der Form

O[SetMethod@R_1, ..., R_l ->> {S_1, ..., S_m}]

einem Objekt-Molekül der Form

O[ScalarMethod@R_1, ..., R_k -> S],

wobei S ein Objektbezeichner ist, der auf ein Objekt einer Mengenklasse verweist. Die Umsetzung solcher Objekt-Moleküle mit skalaren Methoden ist oben beschrieben.

e) Komplexe Formeln

Wie oben angeführt, bestehen komplexe Formeln aus molekularen Formeln bzw. molekularen Formeln, die mit logischen Verknüpfungen oder Quantifikatoren verbunden sind.

Dazu werden, entsprechend den möglichen F-Logic-Konstrukten, folgende Methoden der Hilfsklasse *logic* definiert:

F-Logic-Konstrukt	**Methode von *logic*** (Signatur)
$P_1 \wedge ... \wedge P_n$ [9]	logicResult L:logic.and(P_1:logicResult, ..., P_n:logicResult)
$P_1 \vee ... \vee P_n$	logicResult L:logic.or(P_1:logicResult, ..., P_n:logicResult)
$\neg$ P	logicResult L:logic.not(P:logicResult)

Tabelle 7: Syntaktische Umsetzung von F-Logic-Konstrukten in Methoden

f) Retrieval-Queries

Ein gesamter Query wird durch ein Objekt der Klasse *Retrieval-Query* und die assoziierten Objekte repräsentiert.

Der *Object-Path* der Assoziation *condition* enthält die gesamte Bedingung der Query. Er besteht aus den oben beschriebenen Methoden. Die als *outputs* und *condition-values* der *Retrieval-Query* angegebenen *Object-Specifications* werden neben dem Hilfsobjekt *logic* in ihm referenziert, er verknüpft diese Bestandteile zum Bedingungsteil der Query.

Die *Object-Specifications* der Assoziation *condition-values* spezifizieren die Objekte, deren Daten als Bedingungen in der Query verwendet werden.

---

[9] Die Konstrukte $P_1 \wedge P_2$ bzw. $P_1 \vee .P_2$ aus 7.2b) sind als Sonderfall enthalten

Die *Object-Specifications* der Assoziation *outputs* spezifizieren die Objekte, die als Ergebnis der Query zurückgegeben werden. Die in den *Object-Specifications* angegebenen *classes* spezifizieren dabei die Mengen der Objekte, aus denen die Objekte ausgewählt werden, die der *condition* genügen.

g) Erweiterungen des Ansatzes

Die Modellierung der externen Datenzugriffe mit Hilfe von F-Logic und der UML-Umsetzung stellen - wie bereits erwähnt - nur einen ersten Ansatz dar. Speziell die Behandlung von Objekt-Mengen verlangt noch eine genauere Spezifikation. Diese steht allerdings im engen Zusammenhang mit der Konzeption und Realisierung der CORBA-Management-Layer.

Prinzipiell erscheint der Ansatz jedoch erweiterungsfähig, so daß auch weitere F-Logic-Konstrukte, wie z.B. Prädikate, in die strukturierte UML-Darstellung eingebunden werden können.

Anhang C: Klassen des Workflow-Editors

Im folgenden sollen die Klassen, die im Zusammenhang mit der Workflow-Editor-Implementierung stehen, übersichtsartig dargestellt werden. Nach der Bezeichnung der Klassen werden folgende Gruppen unterschieden:

Präfix	**Bedeutung**
WE...	Klassen des Workflow-Editors
CWF...	Klassen des Database Access Layers (Implementierung im Rahmen von [Die00])
C...	MFC-Klassen, bzw. allgemeine Erweiterungen dieser

Tabelle 8: Bedeutung der Klassenpräfixe

In den folgenden Grafiken ist jeweils nur die Vererbungshierarchie dargestellt, Assoziationen wurden weggelassen.

In Abbildung 28 sind die Klassen dargestellt, welche die Daten der Workflow-Definitionen und der assoziierten Objekte enthalten. Sie entsprechen weitgehend den Klassen des UML-Buildtime-Modells.

Im Zusammenhang mit der grafischen Bearbeitung und Darstellung wurden einige zusätzliche Klassen eingeführt:

Klasse	**Beschreibung**
WEObject	Oberklasse aller im Workflow-Editor bearbeitbaren Objekte, stellt Informationen über auf das Objekt anwendbare Funktionen zur Verfügung
WEGraphObject	enthält Eigenschaften, die alle in einer Workflow-Definition grafisch zu bearbeitenden Objekte besitzen: Selektierbarkeit, Verschiebbarkeit, Größe, Position
WERectObject	spezialisiert WEGraphObject für rechteckige Objekte (Knoten im Graphen)

Klasse	**Beschreibung**
WEPathObject	spezialisiert WEGraphObject für pfadförmige Objekte (Kanten im Graphen)
CRectEx	erweitert MFC-Klasse CRect um einige Methoden

Tabelle 9: Basisklassen zur grafischen Darstellung und Bearbeitung

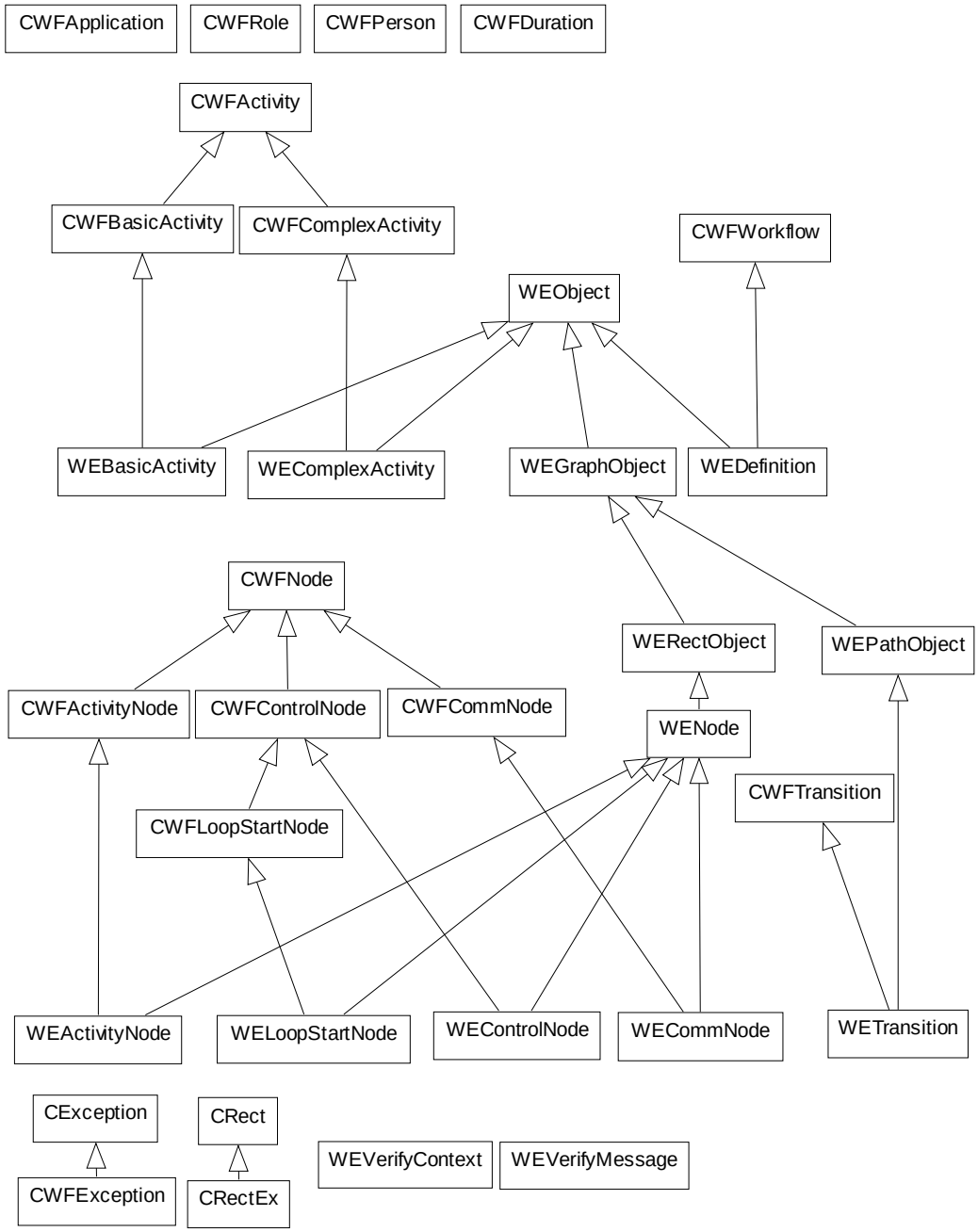


Abbildung 28: Workflow-Klassen

In Abbildung 29 sind die Klassen der Dialoge aufgeführt, die zur Bearbeitung der einzelnen Workflow-Objekte dienen, sowie Dialoge zur Anmeldung und zur Information über den Workflow-Editor.

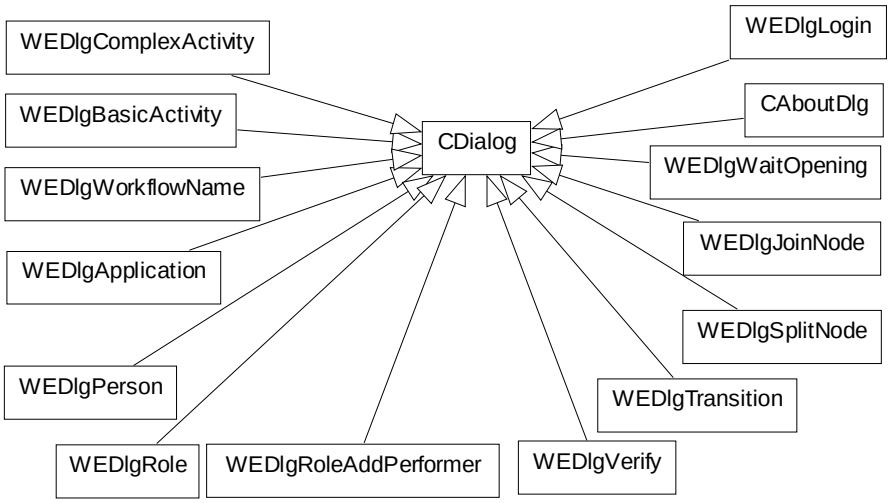


Abbildung 29: Dialogklassen

Abbildung 30 enthält die Klassen, die sich aus der Verwendung des MFC-Dokument-Ansicht-Frameworks ergeben und weitere Oberflächenklassen.

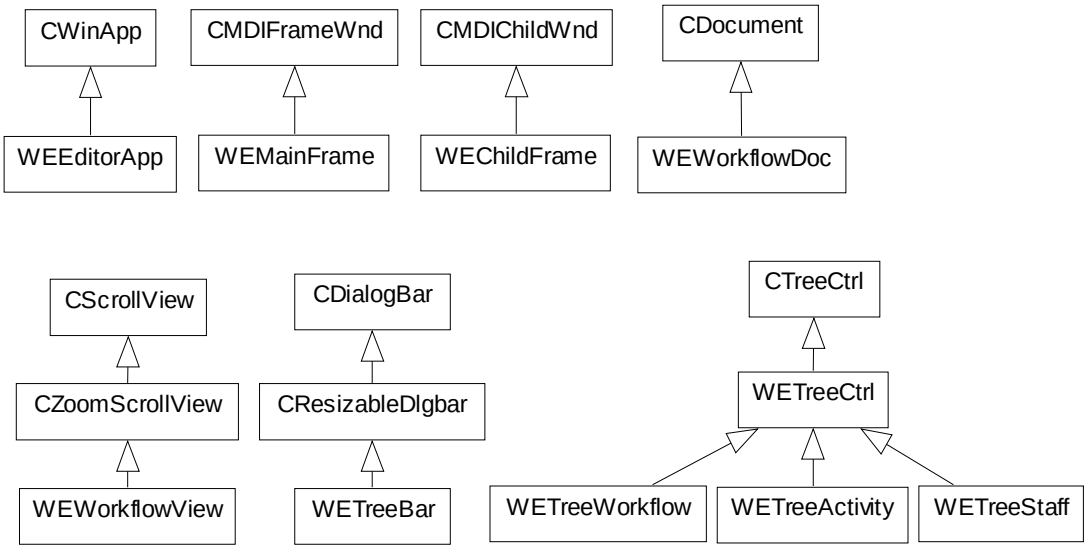


Abbildung 30: Framework- und Oberflächenklassen

Klasse	**Beschreibung**
WEEditorApp	Klasse des Workflow-Editors
WEMainFrame	äußeres Fenster des Workflow-Editors
WEChildFrame	Fenster zur Darstellung einer Workflow-Definition
WEWorkflowDoc	enthält Dokumentdaten, also Daten der Workflow-Definition
CZoomScrollView	erweitert MFC-CScrollView um Funktionen zum Vergrößern/Verkleinern der Darstellung
WEWorkflowView	stellt Workflow-Definitionen grafisch dar, ermöglicht Bearbeitung (markieren, verschieben, Aufruf von Eigenschaftsdialogen, Export der grafischen Darstellung)
WETreeBar	enthält Register zur Darstellung der Baumansichten der Basisobjekte
WETreeCtrl	erweitert MFC-CTreeView um spezielle Funktionen zur Baumdarstellung der Workflow-Basisobjekte
WETreeWorkflow	Baumansicht der Workflow-Definitions und Sub-Workflow-Definitions
WETreeActivity	Baumansicht der Basic- und Complex-Activity-Definitions
WETreeStaff	Baumansicht der Persons und Roles

Tabelle 10: Dokument-Ansicht-Klassen und weitere Klassen

Anhang D: Algorithmus zur Kontrollfluß-Verifikation

In den folgenden Flußdiagrammen ist die ausführliche Darstellung des Algorithmus zur Kontrollfluß-Verifikation enthalten.

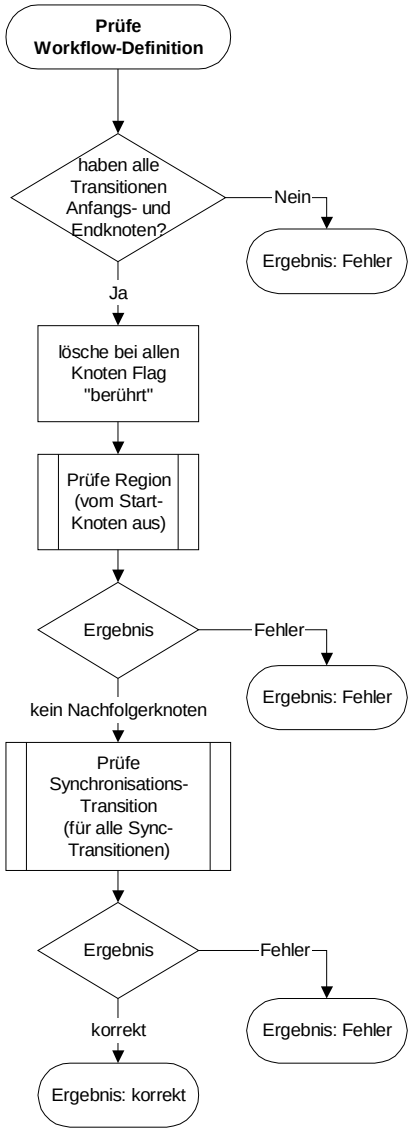


Abbildung 31: Verifikation der Workflow-Definition

Abbildung 32: Verifikation einer Region

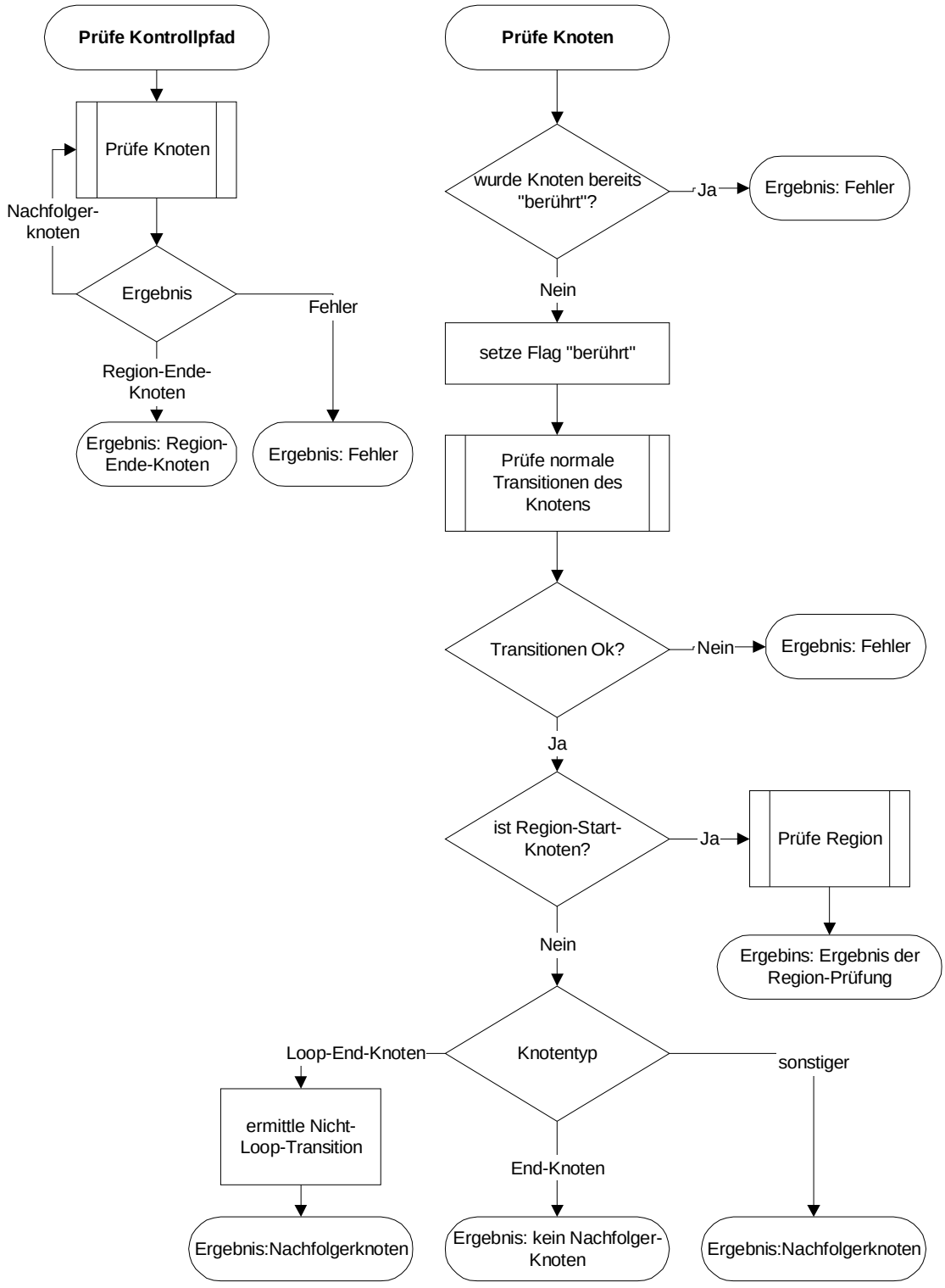


Abbildung 33: Verifikation von Kontrollpfad und Knoten

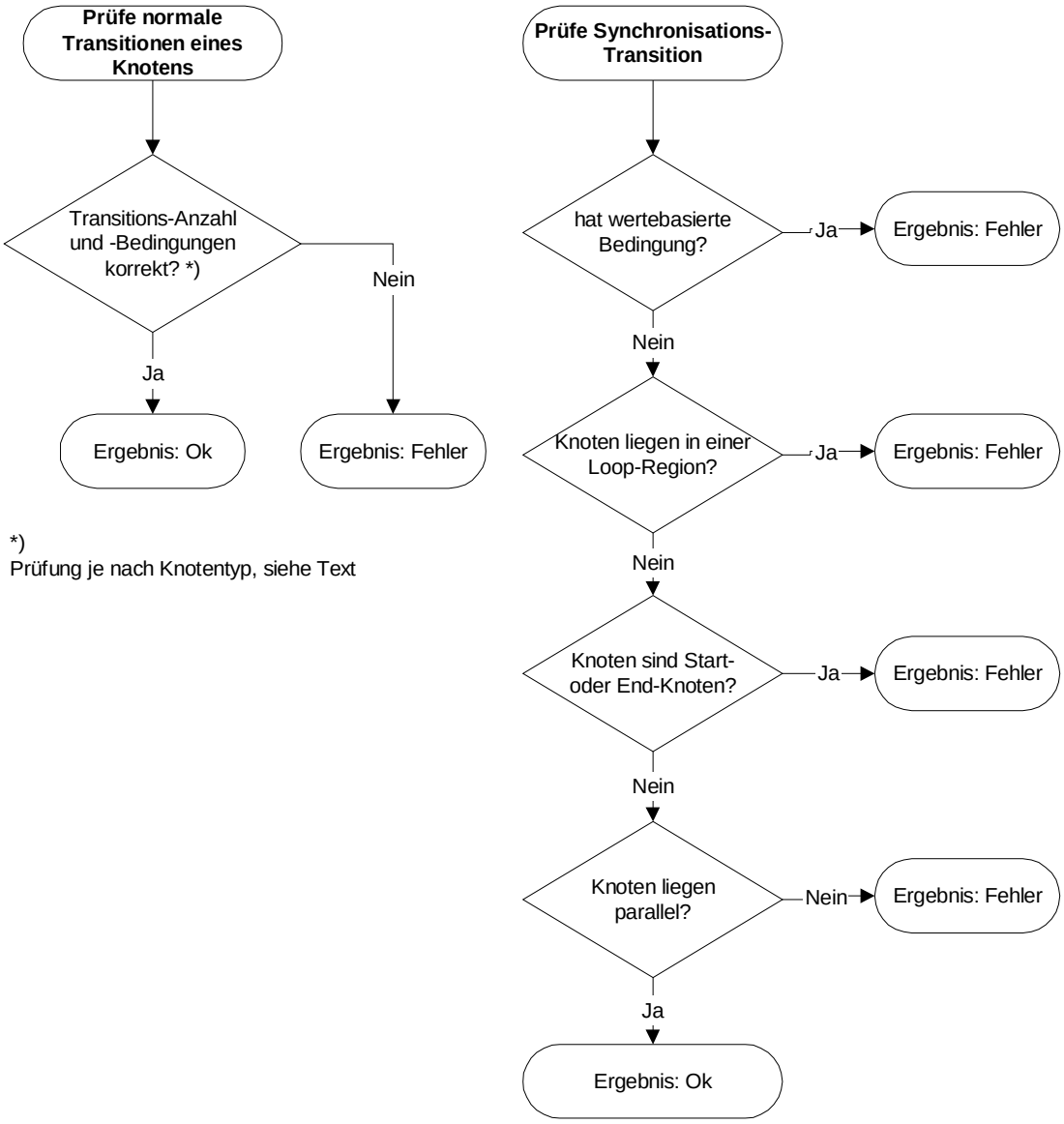


Abbildung 34: Verifikation von Transitionen

Die Bedingungen bei der Prüfung der Transitions-Anzahl und Transitions-Bedingungen sind je nach Knotentyp wie folgt:

Knotentyp	**Anzahl der ausgehenden Transitionen**	
	gesamt	**davon mit wertebasierter Bedingung**
Aktivitätsknoten	1	0
Kommunikationsknoten	1	0
And-Split	n > 1	0
Or-Split	n > 1	n - 1
Join	1	0
Loop-Start	1	0

Knotentyp	**Anzahl der ausgehenden Transitionen**	
	gesamt	**davon mit wertebasierter Bedingung**
Loop-End	2	1
Start	1	0
End	0	0

Tabelle 11: Zulässige Anzahl ausgehender Transitionen

Beim Start-Knoten ist außerdem Bedingung, daß dieser keine eingehende Transition hat.

Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, Juli 2000