

Metamodellbasierte und hierarchieorientierte Workflowmodellierung

Dissertation

zur

Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

der Fakultät für Informatik und Elektrotechnik

der Universität Rostock

vorgelegt von

Jens Brüning, geboren am 15. Februar 1979 in Bremen

aus Rostock

Rostock, 12. Juli 2013

Gutachter:

Prof. Dr. Peter Forbrig, Universität Rostock
Prof. Dr. Martin Gogolla, Universität Bremen
Prof. Dr. Kurt Sandkuhl, Universität Rostock

Datum der Einreichung: 10.12.2012

Datum der Verteidigung: 14.05.2013

Zusammenfassung

In dieser Arbeit werden Metamodelle eingesetzt, um Workflow- bzw. Geschäftsprozessmodellierungssprachen und ihre operationale Semantik zu definieren. Mit einer deklarativen und einer hierarchischen Sprache werden zwei Modellierungsweisen verfolgt, die im Bereich der Geschäftsprozessmodellierung nicht weit verbreitet sind. Der Hauptvorteil beim deklarativen Ansatz liegt in einer höheren Flexibilität und bei der hierarchischen Sprache in einer besseren Verständlichkeit der Modelle. Die Arbeit ist damit in zwei Teile gegliedert, die jeweils mit zwei Kapiteln die Design- und Runtime der entsprechenden Herangehensweise betrachtet.

Der erste Teil präsentiert einen deklarativen, metamodellbasierten Ansatz zur Geschäftsprozessmodellierung, -analyse und -ausführung. Mit dem deklarativen Hintergrund, der standardmäßig alle Ablaufreihenfolgen erlaubt und dann über Constraints einschränkt, wird eine flexiblere Modellierungsweise im Kontrast zu konventionellen, imperativen Sprachen verfolgt. Es werden unterschiedliche Modelle zur Daten-, Organisations- und Geschäftsprozessmodellierung durch ein Metamodell bereitgestellt. Die Modelle werden in einem UML-Spezifikationswerkzeug erstellt während statische Konsistenzeigenschaften über OCL-Invarianten im Entwicklungsprozess überprüft werden. Daraufhin sind sie mit Hilfe einer UML-Action Language im selben Tool ausführbar. Ein Plugin wurde für das UML-Werkzeug erstellt, das die im Ablauf befindlichen Prozesse darstellt. Dem Nutzer wird dadurch eine Interaktionsschnittstelle zur Validierung der dynamischen Modelleigenschaften gegeben. Die Modelle sind adaptiv und können zur Laufzeit verändert werden. Des Weiteren kann OCL zum Process Mining eingesetzt werden, um Eigenschaften der abgearbeiteten Prozesse zu erhalten.

Im zweiten Teil wird eine hierarchische, dekompositionsorientierte Modellierung von Workflows betrachtet. Aufgabenbäume stellen eine etablierte Modellierung im Human Computer Interaction Bereich dar, die eine nutzerzentrierte, problemorientierte Aktivitätsmodellierung ermöglicht. Unterschiedliche Abstraktionsebenen werden in einem strukturierten Modell abgebildet, welches Verständlichkeitsvorteile im Unterschied zu anderen Sprachen bietet. Weitere Vor- und Nachteile dieser Modellierung werden im Fokus der Geschäftsprozessmodellierung und der dort etablierten Sprachen betrachtet. Transformationsregeln dienen dazu, Zusammenhänge von Aufgabenmodellen zu UML-Aktivitätsdiagrammen aufzuzeigen. Außerdem findet der Metamodellierungsansatz und das Tool vom ersten Teil auch Einsatz für Aufgabenmodelle. Hierfür ist ein präzises Metamodell zur Erstellung und Ausführung der Aufgabenmodelle im UML-Werkzeug entwickelt worden. Zudem wird ein Workflow Management System auf Basis von Aufgabenmodellen präsentiert, um eine verteilte Modellausführung bereitzustellen. Dieses System setzt vorhandene und neue Sprachkonzepte für diese um.

Abstract

This work presents new metamodel-based approaches for workflow or business process modeling. Two different ways for business process modeling are followed with a declarative and a hierarchical one that are not common nowadays. The main advantage for the declarative language lies in a better flexibility and the hierarchical approach promises structured models that are better to be understood. Therefore, the work is divided into two parts that comprises two chapters respectively to introduce the design and runtime for each language.

In the first part a declarative metamodel-based approach for business process modeling, execution and analysis is introduced. With the declarative background, we follow a more flexible way of modeling workflows compared to the traditional, imperative languages by giving more execution possibilities to the user by default. Different models for data, organization and workflows are used and mutually connected based on the presented metamodel. The developer creates them in the UML tool while static consistency properties are checked by OCL invariants. Afterwards, they can be executed based on the tool realizing parts of the UML action semantics. A plugin for the UML tool provides an appropriate interface to the user that presents the workflow instances at the runtime. She can validate the dynamic model properties with the plugin by executing the developed models. They are adaptive and can be changed or developed at runtime. Lastly, the developer can use OCL for process mining purposes to discover properties of the executed process scenarios.

The second part introduces a hierarchical, decomposition-oriented approach for modeling workflows. Task trees are an established modeling approach in the field of Human Computer Interaction that represent user-centered, goal-focused activity models. In contrast to established workflow languages, one structured model captures several abstraction levels of a workflow, that offers benefits for model understandability. Some advantages and shortcomings for this modeling approach are discussed in the context of business process modeling. Transformation rules show connections between UML activity diagrams and task models. In addition, a metamodel-based approach and the tool from the first part of this work are used to define task models. The precise metamodel enables the developer to find soundness problems and also execute these models employing the UML tool. Furthermore, a Workflow Management System is introduced on the basis of task models to provide a distributed workflow execution. This system implements existing and new language elements for task models.

Danksagung

Ich möchte mich als erstes bei meinen Gutachtern bedanken. Martin Gogolla gebührt ein besonderer Dank für die gute Zusammenarbeit und Unterstützung.

Weiterhin danke ich meinen langjährigen Kollegen Andreas Wolff, Gregor Buchholz und den Kollegen vom Lehrstuhl Wirtschaftsinformatik für die gute Arbeitsatmosphäre.

Bei folgenden meiner ehemaligen Diplomanden möchte ich mich für die rege Diskussion und die Umsetzung der Ideen bedanken: Christian Sauer, Karsten Bonhuis, Hannes Müller und Martin Kunert.

Als letztes danke ich meiner Familie und meinen Freunden für die stete Unterstützung. Diese Arbeit ist meinem Vater gewidmet.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Probleme etablierter Ansätze und Ziele der Arbeit	2
1.2	Eigenschaften und Vorteile der neuen Ansätze sowie Entwicklung der Arbeit	3
1.2.1	Metamodellbasierter Ansatz zur Workflowmodellierung	3
1.2.2	Hierarchieorientierter Ansatz zur Workflowmodellierung	5
1.2.3	Autoren-Literaturliste	6
1.3	Aufbau der Arbeit und Hinweise	7
2	Grundlagen: Modellierung in ausgewählten Gebieten	9
2.1	Modellierung in der Softwaretechnik	9
2.2	Daten- und Metamodellierung	11
2.3	Modellierung in der HCI mit nutzerzentrierten Entwurfstechniken	13
2.4	Modellierung von Unternehmen und betrieblicher Informationssysteme	14
2.5	Geschäftsprozessmodellierung	15
2.5.1	Kategorisierungen von Geschäftsprozessmodellen	15
2.5.2	Etablierte Modellierungssprachen	17
2.5.3	Metamodelle für Geschäftsprozessmodellierungssprachen	25
3	Deklarative UML metamodellbasierte Workflowmodellierung	29
3.1	Einführung	29
3.1.1	Ausgewählte UML-Diagramme zur Verwendung für DMWM	30
3.1.2	Spezifikationen von Zusicherungen mit OCL	32
3.2	Workflow-Metamodell	34
3.2.1	UML-Klassendiagramm	34
3.2.2	Zustandsdiagramme	37
3.2.3	OCL-Zusicherungen	38
3.2.4	Workflow Patterns	42
3.2.5	Ein Beispielprozess in DMWM	48

3.2.6	Sicherstellung statischer Eigenschaften der Modelle	49
3.3	Datenintegration	51
3.3.1	Konzepte und Probleme der Datenintegration in etablierten Modellierungssprachen	51
3.3.2	Metamodell für Datenintegration	53
3.3.3	Datenmodellierung und Spezifikation der Datenkonsistenz in Workflowmodellen .	55
3.4	Organisationsmodellierung bei DMWM	57
3.4.1	Metamodell für die Organisationsmodellierung	57
3.4.2	Organisationsmodellierung und Zusammenhang zu Workflowmodellen	59
3.5	Realisieren von DMWM mit USE	60
3.5.1	UML-Entwicklungsumgebung USE	60
3.5.2	Anwendungsfälle zur Modellierung von DMWM mit USE	62
3.5.3	Modellierung eines Workflows	63
3.5.4	Unterstützung des Nutzers bei Modellerstellung	64
4	Metamodellbasierte Workflow-Modellausführung	67
4.1	Einführung	67
4.2	Instanziierung und Grundlagen zur Ausführung der Modelle	69
4.2.1	ASSL Sprachvorstellung und Erweiterung	69
4.2.2	Imperative Teil des Metamodells	70
4.2.3	Verhältnis Metamodelle, Modelle und Instanzen	72
4.2.4	Workflow Designtime-Plugin und Toolchain	73
4.3	Workflow Runtime-Plugin	76
4.3.1	Sichten auf Prozessinstanz	76
4.3.2	Ausführung spezieller Aktivitäten	78
4.3.3	Anzeige der Datenabhängigkeiten	82
4.3.4	Adaptive Aspekte zur Workflow Modellierung	84
4.4	Analyse ausgeführter Prozesse und Process Mining	84
4.4.1	Ausführungsprotokoll und Analyse im Runtime-Plugin	84
4.4.2	UML-Sequenzdiagramme	85
4.4.3	Process Mining mit OCL	85
4.5	Verwandte Arbeiten	87
5	Hierarchieorientierte Workflowmodellierung	91
5.1	Einführung	91
5.1.1	Verbreitete hierarchische Modellierungstechniken	91
5.1.2	Aufgabenmodelle zur Spezifikation von Workflows	92
5.2	Transformation von CTT zu Aktivitätsdiagrammen	94

5.2.1	Transformation der Hierarchie	95
5.2.2	Einführung der temporalen CTT-Operatoren	96
5.2.3	Transformation der binären temporalen Operatoren	97
5.2.4	Transformation der unären temporalen Operatoren	99
5.3	Definition konsistenter CTT-Aufgabenmodelle	100
5.3.1	Soundness-Probleme bei CTT-Aufgabenmodellen	100
5.3.2	Metamodell für MCTT	102
5.3.3	OCL-Invarianten zum Prüfen von strukturellen Eigenschaften	104
5.3.4	Metamodellbasierte Spezifikation von MCTT-Modellen	106
5.4	Entscheidungsmodellierung und Datenaspekte in Aufgabenmodellen	108
5.4.1	Explizite Entscheidungsmodellierung mit Decisionnodes	108
5.4.2	Datenbasierte Entscheidung und Datenintegration bei MCTT	110
6	Workflow-Ausführung und -Management mit Aufgabenmodellen	115
6.1	Einführung	115
6.2	Modellausführung von MCTT-Aufgabenmodellen	117
6.2.1	Umsetzung der operationalen Semantik mit SOIL	117
6.2.2	Taskmodel-Plugin für USE	119
6.3	TTMS: A Task Tree Based Workflow Management System	121
6.3.1	TTMS-Workflowsprache und Editor	121
6.3.2	TTMS-WfMS	123
6.4	Weitere Ansätze	130
7	Zusammenfassung und Ausblick	133
7.1	Fazit	134
7.2	Weiterführende Diskussion	136
7.3	Ausblick	137
	Tabellenverzeichnis	139
	Abbildungsverzeichnis	142
	Listingsverzeichnis	143
	Literaturverzeichnis	145

Kapitel 1

Einleitung

Geschäftsprozessmodellierung wird für die Wirtschaft mit zunehmender Komplexität und Automatisierung der Abläufe immer wichtiger. Um Geschäftsprozesse zu dokumentieren, effizienter zu gestalten und Anforderungen für unterstützende Software zu spezifizieren, werden zunehmend Modelle eingesetzt.

Die vorliegende Arbeit beschäftigt sich mit Modellen für Geschäftsprozesse bzw. Workflows. Es werden bewährte Modellierungsmittel zum einen mit Metamodellen im Zusammenhang mit UML und deklarativen OCL-Constraints und zum anderen mit hierarchischen, dekompositionsorientierten Modellen aus dem Bereich der Human Computer Interaction genutzt, um sie zur Geschäftsprozessmodellierung einzusetzen. Bisher haben sich vor allem flache, imperative Sprachen, wie z.B. Ereignisgesteuerte Prozessketten (EPKs) [KNS92] oder die Business Process Model and Notation (BPMN) [BPM09] durchgesetzt, um Geschäftsprozesse abzubilden [PWZ⁺11]. Jede Sprache hat ihre spezifischen Eigenschaften und Modellierungsphilosophie, die Vor- und Nachteile für bestimmte Aspekte zum Abbilden von Geschäftsprozessen haben. Diese Arbeit verfolgt die zwei erwähnten von den etablierten unterschiedlichen Ansätze. Es werden Vor- und Nachteile dieser Herangehensweisen diskutiert und in manchen Bereichen die verwendete Sprache erweitert.

Der Aspekt der Darstellung, Verständlichkeit, Formalität und Ausführbarkeit spielt eine wichtige Rolle bei der Modellierung im Allgemeinen und insbesondere in dieser Arbeit. Visuelle Modelle sind in der Regel besser lesbar und intuitiver als textuelle Spezifikationen. Diese stellen häufig Spezifikationen dar, die dafür geeignet sind, Modelle ausführbar zu machen. Man kann diese Sprachen als eine Mischung aus Modell und Programmcode ansehen. Z.B. BPEL [BPE07], Algebraische Spezifikationen [BM04], Prozessalgebren [Bae04] oder die Object Constraint Language (OCL) [WK04] kann man zu dieser Art von Spezifikationssprachen zählen. Die Formalität bei den Modellierungssprachen spielt hier eine wesentliche Rolle. Sehr formale Modelle und Spezifikationen sind schlechter lesbar. Dagegen haben informelle Modelle einen hohen Grad an Mehrdeutigkeiten, die eine Ausführung der Modelle verhindert. Es muss permanent ein Kompromiss zwischen Formalität und Verständlichkeit bzw. Intuition gesucht werden. Eine solche Diskussion wird in Kapitel 2 für ausgewählte Bereiche der Informatik geführt.

Die Abstraktionsebene von Modellen spielt ebenfalls eine wichtige Rolle für diese Arbeit. Beispielsweise lassen sich in der Abstraktionsebene höher gelegene Softwarearchitekturmodelle schlecht in Hardwaremodellierungssprachen abbilden. Die Abstraktionsebene der hier verfolgten Ansätze ist relativ hoch angesiedelt und damit semantisch ausdrucksstärker als eher technische Modelle, wie z.B. Petri-Netze. Es ist die Ebene analog zu den populären Modellierungssprachen für Geschäftsprozesse wie EPKs oder BPMN angestrebt worden. Diese Sprachen lassen sich zwar in ausführbare Petri-Netze transformieren, haben dabei aber einen

Verlust an Semantik, weil sich nicht alle Modellierungselemente in Petri-Netze übersetzen lassen.

Der hierarchische Ansatz hat des Weiteren die Eigenschaft, mehrere Abstraktionsebenen in einem Modell auszudrücken. Hierbei spielen die Modellierung von Zielen einzelner Aktivitäten bzw. Aktionen eine Rolle. Beide Ansätze sind visuell und direkt ausführbar. Es müssen keine Transformationen von visuellen zu textuell ausführbaren Sprachen oder Petri-Netzen gemacht werden.

Abschnitt 1.1 führt Probleme auf, die sich bei den etablierten Modellierungssprachen gezeigt haben. Ziele und Lösungsperspektiven, die mit dieser Arbeit entstanden sind, werden aufgezeigt. Eine prägnante Aufzählung der Vorteile der hier verfolgten Ansätze findet in Abschnitt 1.2 statt. Außerdem werden Abgrenzungen vorgenommen, die angeben, wofür die neu eingeführten Techniken, Sprachen und Werkzeuge zur Geschäftsprozessmodellierung eingesetzt werden können bzw. (noch) nicht geeignet sind. Des Weiteren beschreibt dieser Abschnitt den wissenschaftliche Entstehungsprozess dieser Arbeit anhand der entstandenen Publikationen. Der Abschnitt 1.3 beschreibt schließlich den weiteren Aufbau und die Struktur.

1.1 Probleme etablierter Ansätze und Ziele der Arbeit

Aktuell populäre, verbreitete Sprachen zur Workflowmodellierung verfolgen einen imperativen Stil, der angibt, wie ein Prozess abzulaufen hat. Dabei wird beispielsweise bei EPKs und BPMN über Startereignisse ein definierter Anfang und über Endereignisse ein definiertes Ende spezifiziert. Dazwischen wird über Ketten von Aktivitäten angegeben, wie man vom Start zum Ende gelangt. Dieser Ansatz kann für bestimmte Einsatzzwecke zu restriktiv sein und den Nutzer in bestimmten Situationen in der Ausführung der Tätigkeit zu sehr einschränken.

Der hier vorgestellte Ansatz verfolgt dagegen einen deklarativen und damit flexibleren Stil, Workflows zu beschreiben (vgl. [APS09]). In den Modellen sind die modellierten Aktivitäten grundsätzlich unabhängig voneinander. Daraufhin können Ablaufreihenfolgen über die Constraintsprache OCL verboten werden. Alle anderen Ablaufpfade bleiben im Modell weiterhin erlaubt. Mit den verbietenden OCL-Constraints kommt der Modellierer nicht in Kontakt, da sie hinter visuellen Modellierungselementen verborgen sind. Somit wird eine intuitive visuelle Modellierung unterstützt. Mit dem deklarativen Ansatz werden dem Endnutzer während der Runtime mehr Möglichkeiten für die Prozessausführung gegeben. Es gibt zwar bereits Sprachen, die deklarativ Workflows beschreiben, aber keine nutzt einen metamodellbasierten Ansatz, der noch weitere im Folgenden angegebene Vorteile bietet.

In den bisherigen Workflow-Modellierungssprachen fehlen ausführbare Ansätze, die eine visuelle Datenmodellierung z.B. über UML-Klassendiagramme mit Workflowmodellen verbindet. Etablierte Modellierungssprachen wie z.B. eEPKs in ARIS (s. Abschnitt 2.5.2.1) sind nicht ausführbar. Dagegen werden in Workflowsystemen, die ausführbare Workflows unterstützen (z.B. YAWL [AH05]) nur nicht intuitive XML-basierte Datenspezifikationen genutzt. Eine visuelle Datenmodellierung mit unmittelbarer Nutzung der Modelle zur Ausführung ist hier nicht vorgesehen. Der Ansatz, der mit dieser Arbeit verfolgt wird, nutzt die grafische Workflow- und Datenmodelle unmittelbar ohne jegliche Transformation zur Ausführung. Damit wird eine Art *rapid prototyping* für Workflowmodelle unterstützt. Dynamische Modelleigenschaften im Zusammenhang mit Daten, die während des Prozesses abgefragt bzw. erhoben werden, werden damit validiert.

Etablierte Ansätze für die Modellierung von Workflows nutzen an Petri-Netzen angelehnte Modellierungssprachen, welche unstrukturierte Modelle [KHB00] erlauben, die das Verständnis beim Endanwender erschweren können [LG08]. Bei Programmiersprachen wurde unstrukturierter Programmcode mit *goto*-Anweisungen als schädlich für das Verständnis des Programms angesehen [Dij68]. Entsprechend haben sich

darauhin die strukturierten Programmiersprachen durchgesetzt. Es gibt im Bereich Geschäftsprozessmodellierung Arbeiten, die aussagen, dass auch dort vorwiegend strukturierte Modelle sinnvoll sind [LM10]. Einen zusätzlichen Verständnissvorteil kann eine praktikabel in die Workflowmodelle integrierte Zielmodellierung liefern. Diese Art der Modellierung wird bereits mit Aufgabenmodellen im Human Computer Interaction (HCI) Gebiet zur nutzerzentrierten Aktivitätsmodellierung genutzt.

Der Einsatz von Aufgabenmodellen birgt aber auch Probleme. Durch ihre semiformale Grundlage werden nicht alle Modellierungsfehler in den Modellen identifiziert. Somit müssen nicht alle Aufgabenmodelle fehlerfrei, d.h. *sound* sein. Dagegen sind im Workflow-Gebiet Sprachen wie Petri-Netze bzw. Workflow-Netze etabliert, die eine strenge formale Fundierung haben. Anhand dessen lassen sich Modelle, die *sound* von denen die nicht *sound* sind unterscheiden [AS11].

Weiterhin ist die operationale Semantik einzelner Modellierungselemente bei Aufgabenmodellen durch ihre semiformale Definition nicht klar und eindeutig definiert. Das bedeutet, dass es Modellierungselemente, obwohl sie gleich aussehen, in unterschiedlichen Modellierungssprachen für Aufgabenmodelle eine unterschiedliche Ausführungssemantik haben. Um dieses Problem zu lösen und eine eindeutige Semantik vorzuschlagen, werden in dieser Arbeit Metamodelle eingesetzt. Die Semantik der wohl populärsten Aufgabenmodellierungssprache CTT [Pat99] wird metamodelliert nachgebildet. Hier werden Soundness-Eigenschaften im Metamodell spezifiziert, um Dead- oder Lifelocks bereits während der Designzeit zu identifizieren. Somit wird über Metamodelle ein Beitrag für eine konsistente, eindeutige Definition für Aufgabenmodelle geleistet.

Es ist wahrscheinlich, dass Aufgabenmodelle Defizite in diversen Bereichen der Workflowmodellierung aufweisen, die hingegen von bestehenden, etablierten Ansätzen in diesem Gebiet unterstützt werden. Etablierte Sprachen wie EPKs oder BPMN haben u.a. eine visuelle Integration von Daten-, Organisationsmodellen und verschiedene Arten der Entscheidungsmodellierung. In diesem Zusammenhang wird eine Integration der expliziten Entscheidungsmodellierung und Datenintegration für Aufgabenmodelle vorgestellt.

1.2 Eigenschaften und Vorteile der neuen Ansätze sowie Entwicklung der Arbeit

In dieser Arbeit werden zwei verschiedene Modellierungstechniken bzw. Tools aus unterschiedlichen Bereichen der Informatik zur Modellierung von Workflows bzw. Geschäftsprozessen eingesetzt.

In den folgenden Unterabschnitten werden die Eigenschaften und Vorteile aufgeführt, die die Ansätze charakterisieren. Der jeweilige Entwicklungsprozess der Ansätze wird anhand der zugeordneten Publikationen dokumentiert.

1.2.1 Metamodellierter Ansatz zur Workflowmodellierung

Der erste Ansatz ist ein metamodellierter, der Geschäftsprozesse mit unterschiedlichen Aspekten der Unternehmensmodellierung wie z.B. das Daten- und Organisationsmodell anreichert. Im Gegensatz zu anderen Metamodellen, die vor allem die Syntax einer Modellierungssprache definieren (s. Abschnitt 2.5.3), wird in diesem Ansatz die operationale Ausführungssemantik ebenfalls spezifiziert. Bisher sehr verbreitete visuelle Modelle, wie z.B. EPKs, müssen nicht erst in ausführbare textuelle Modelle, wie z.B. BPEL, transformiert werden, damit sie ausführbar werden. Die visuellen Modelle lassen sich unmittelbar in der Modellierungsumgebung anhand einer imperativen Sprache, die als eine UML-Action Language genutzt wird, ausführen. Die Umsetzung in dieser Sprache wird zusammen mit dem Metamodell bereitgestellt, so

dass der Workflow-Modellierer mit dem textuellen, nicht intuitiven Code nicht in Kontakt kommt.

Mit diesem Ansatz lassen sich die Workflowmodelle unmittelbar testen. Hierfür wurden Mittel und ein Werkzeug zur semiformalen Validation von UML-Softwaremodellen aus dem Bereich der leichtgewichtigen formalen Softwaretechnik eingesetzt und im Bereich der Geschäftsprozessmodelle verwendet. Daten- und Organisationsaspekte können in diesem Ansatz ebenfalls modelliert werden. Das Vorgehen verspricht bessere und validierte Workflow-, Daten- und Organisationsmodelle. Diese lassen sich daraufhin zu unterschiedlichen Zwecken einsetzen. Zum einen können sie als Diskussionsgrundlage genutzt werden mit dem Ziel, Geschäftsabläufe zu optimieren ohne zusätzliche Software dafür zu entwickeln. Zum anderen können die validierten Modelle als Grundlage für eine weitere Systemimplementierung genutzt werden, die den Geschäftsprozess unterstützen soll. Dagegen dienen diese Modelle nicht für konkrete Workflows in einem Unternehmen, um Arbeitsabläufe zu steuern und Arbeit in Verbindung mit einem WfMS zu koordinieren. Folgende Vorteile machen den Ansatz sehr vielversprechend.

- **Flexibilität:** Der Ansatz erlaubt grundsätzlich alle Ausführungsreihenfolgen der modellierten Aktivitäten. Die Ausführungsreihenfolgen werden dann über Modellelemente und zugeordnete OCL-Invarianten eingeschränkt. Diese Herangehensweise ist zu konventionellen Sprachen wie z.B. BPMN [Whi04, BPM09] grundsätzlich verschieden. Deklarative Sprachen haben dabei Vorteile durch eine geringere Restriktivität bei der Modellausführung. Der Nutzer hat in der Regel mehr Auswahlmöglichkeiten, die Aktivitätsfolgen bei der Prozessausführung zu bestimmen [APS09].
- **Ausführbarkeit:** Allen Sprachelementen wird im Metamodell eine operationale Semantik gegeben, so dass die Modelle ausführbar sind.
- **Soundness:** Die Modelle werden auf Konsistenzeigenschaften bereits bei der Erstellung geprüft. Der Modellierer wird unmittelbar über Fehler in Kenntnis gesetzt. Ihm werden Werkzeuge zur Identifikation der Fehler bereitgestellt.
- **Mächtigkeit:** Es lassen sich alle Workflow Patterns [AHKB03] mit dem Ansatz ausdrücken.
- **Adaptivität:** Während der Modellausführung können weitere Elemente hinzugefügt bzw. temporale Beziehungen zwischen Aktivitäten verändert werden.
- **Datenintegration:** Die Datenmodellierung kann recht intuitiv mit UML-Klassendiagrammen geschehen. Datenmodelle lassen sich während der Modellausführung validieren. Gegenseitige Abhängigkeiten zwischen Daten und Workflowmodell können anhand von OCL spezifiziert werden.
- **Organisationsmodellintegration:** Eine umfangreiche Organisationsmodellierung ist auf Grundlage eines Organisationsmetamodells möglich. Eine Allokation von Ressourcen für die Workflow-Modellausführung kann aus den Modellen abgeleitet werden. Daraufhin können Workflowmodelle in unterschiedlichen Organisationskontexten getestet werden.

Anhand folgender Veröffentlichungen lassen sich die in dieser Arbeit präsentierten Ergebnisse, die die metamodellbasierte Workflowmodellierung betreffen, nachvollziehen. Der Ansatz hat sich im Laufe der Zeit und nach diversen Diskussionen (u.a. bei Workshops und Konferenzen) entwickelt und grundlegend verändert. Zunächst sind UML-Klassendiagramme und OCL-Invarianten eingesetzt worden, um eine deklarative und flexible Workflowmodellierung [Brü09] zu schaffen. Textuelle OCL-Modelle waren hier im Zentrum der Modellierung. Mit dem gleichen Ansatz wurde in [BW09] versucht, das Datenmodell und eine User Interface-Modellierung mit in die Modelle zu integrieren. Dabei hat sich herausgestellt, dass eine textuelle Modellierung anhand von OCL nicht praktikabel war. Die OCL-Invarianten sind daraufhin in

das Metamodell verlegt worden. Damit kam der Modellierer nur mit den visuellen Elementen und nicht mehr mit den textuellen Constraints in Berührung. Dieser grundlegend überarbeitete Ansatz ist in [BGF10] präsentiert worden. In dem Zusammenhang kam die Sprache ASSL zum Einsatz, um den imperativen Teil des Metamodells auszudrücken [BHW11]. In [BG11] wurde der Workflowmodellierungsansatz erweitert, um die Datenmodellierung und organisatorische Aspekte mit zu berücksichtigen. Außerdem ist in dem Artikel eine Workflow Pattern Analyse zu finden, die gezeigt hat, dass sich alle Patterns mit dem Ansatz ausdrücken lassen. Bei der Datenintegration hat sich gezeigt, dass Probleme, die in anderen Sprachen wie z.B. Aktivitätsdiagrammen oder BPMN auftreten und in [BF08a] analysiert wurden, nicht auftreten. Zusätzlich ist das entwickelte Plugin für das UML-Tool USE in [BG11] und [BHW11] präsentiert worden, das eine angemessene Interaktionsschnittstelle dem Nutzer bereitstellt, der die Workflowmodelle damit testen kann. Möglichkeiten einer komplexen Fehlersuche auf Basis von OCL-Termauswertungen und eines OCL-Debuggers wurden in [BGHK12] vorgestellt, die auch für die Workflowansätze dieser Arbeit geeignet sind.

1.2.2 Hierarchieorientierter Ansatz zur Workflowmodellierung

Der zweite in dieser Arbeit behandelte Ansatz unterscheidet sich von dem vorher diskutierten insofern, dass hier eine andere Sprache und damit ein anderes Metamodell genutzt wird. Es wird ein streng hierarchieorientierter Ansatz zur Modellierung von Geschäftsprozessen verfolgt. Diese Art der Aktivitätsmodellierung wird im Bereich der Human Computer Interaction genutzt und hat sich dort bewährt. Die Aufgabenmodelle beinhalten die Kontrollflussspezifikation innerhalb der Baumstruktur. Diese Modelle berücksichtigen die sehr begrenzten kognitiven Fähigkeiten des Menschen. Es sind dafür ausschließlich strukturierte Modelle erlaubt.

Folgende Aspekte machen den hierarchieorientierten Ansatz sehr vielversprechend für die Workflowmodellierung.

- **Mehrschichtigkeit:** Mehrere Abstraktionsebenen werden in einem Modell ausgedrückt. Dieses verspricht Vorteile für die visuelle Erfassbarkeit der Modelle.
- **Zielmodellierung:** Kontextziele werden dem Modellierer während der Modellierung und Endanwender während der Workflow-Modellausführung präsentiert.
- **Strukturiertheit:** Modelle sind strukturiert (vgl. [KHB00]), das ebenfalls Verständnissvorteile hat. Dagegen können Petri-Netz ähnliche Geschäftsprozessmodellierungssprachen schnell zu unstrukturierten, spaghettiartigen Modellen führen, die sehr schwer zu verstehen sind [AG07].
- **Visualisierung und Ausführbarkeit:** Kontrollflussspezifikation ist mächtig und ist visuell im Baum integriert. Sie basiert bei den populären ConcurTaskTrees (CTT) [Pat99] auf der etablierten Modellierung von Prozessalgebren.
- **Toolunterstützung:** Aufgabenmodelle stellen eine etablierte Modellierungsart dar, die bereits von mehreren Tools unterstützt wird.
- **Verteilte Ausführbarkeit:** Modelle können verteilt in einem WfMS ausgeführt werden. Die entsprechenden Tools werden in Kapitel 6 vorgestellt.

Anhand folgender Veröffentlichungen lassen sich die hier präsentierten Ergebnisse, die die hierarchieorientierte Workflowmodellierung betreffen, nachvollziehen. Zunächst wurde in [BDFR07] das Verhältnis zwischen Aufgabenmodellen und UML-Aktivitätsdiagrammen präsentiert, indem Transformationen von

Aufgabenmodellen zu Aktivitätsdiagrammen vorgeschlagen wurden. Die Strukturiertheit der Workflowmodelle lassen sich anhand der Transformationsregeln und der erzeugten Aktivitätsdiagramme erkennen. Die Entscheidungsmodellierung spielt in Geschäftsprozessmodellen eine wichtige Rolle und wird in EPKs grundsätzlich unterschiedlich zu Aufgabenmodellen behandelt. Als Erweiterung der Entscheidungsmodellierung wurden die Instanziierungsoperatoren für EPKs in [BF08b] eingeführt, die später für Aufgabenbäume und das entwickelte WfMS adaptiert wurden. EPKs haben mit konfigurierbaren Entscheidungsoperatoren die Grundlage für die Instanziierungsoperatoren gelegt. Die unterschiedliche Modellierung und Präsentation von Entscheidungsoperatoren in einem WfMS wurde in [BF09] untersucht. Die hierarchieorientierte Modellierung von Workflows anhand von Aufgabenbäumen und das zugehörige WfMS wurde in [BF11] vorgestellt. Ein metamodellbasierter Ansatz zur Modellierung von ConcurTaskTrees ist in [BKL12] präsentiert worden, bei der Soundness-Eigenschaften und die operationale Semantik im Metamodell integriert wurde. Weitergehende Vorschläge, wie man hierarchische baumartige Modelle auf Basis von ConcurTaskTrees auch für komplexe Eventmodellierung für Workflows einsetzen könnte, wurden in [BFSZ12] präsentiert.

1.2.3 Autoren-Literaturliste

Zur Übersicht wird in diesem Abschnitt die Literatur dokumentiert, die im Laufe meiner Forschungsaktivitäten entstanden ist und auf der diese Arbeit basiert.

- [BDFR07] BRÜNING, Jens ; DITTMAR, Anke ; FORBRIG, Peter ; REICHARD, Daniel: Getting SW Engineers on Board: Task Modelling with Activity Diagrams. In: GULLIKSEN, Jan (Hrsg.) ; HARNING, Morten B. (Hrsg.) ; PALANQUE, Philippe A. (Hrsg.) ; VEER, Gerrit C. d. (Hrsg.) ; WESSON, Janet (Hrsg.): *Engineering Interactive Systems (EIS2007)* Bd. 4940, Springer, 2007 (LNCS)
- [BF08a] BRÜNING, Jens ; FORBRIG, Peter: Behaviour of flow operators conected with object flows in workflow specifications. In: WRYCZA, Stanislaw (Hrsg.): *Perspectives of Business Informatics Research (BIR2008)*, University of Gdansk, 2008
- [BF08b] BRÜNING, Jens ; FORBRIG, Peter: Methoden zur Adaptiven Anpassung von EPKs an Individuelle Anforderungen vor der Abarbeitung. In: LOOS, Peter (Hrsg.) ; NÜTTGENS, Markus (Hrsg.) ; TUROWSKI, Klaus (Hrsg.) ; WERTH, Dirk (Hrsg.): *Proceedings of the Workshops colocated with the MobIS2008 Conference: Including EPK2008, KobAS2008 and ModKollGP2008, Saarbrücken, Germany, November 27-28, 2008* Bd. 420, CEUR-WS.org, 2008 (CEUR Workshop Proceedings), S. 31–43
- [BF09] BRÜNING, Jens ; FORBIG, Peter: Modellierung von Entscheidungen und Interpretation von Entscheidungsoperatoren in einem WfMS. In: NÜTTGENS, Markus (Hrsg.) ; RUMP, Frank J. (Hrsg.) ; MENDLING, Jan (Hrsg.) ; GEHRKE, Nick (Hrsg.): *Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (EPK2009)* Bd. 554, 2009 (CEUR-WS)
- [BF11] BRÜNING, Jens ; FORBRIG, Peter: TTMS: A Task Tree Based Workflow Management System. In: HALPIN, Terry A. (Hrsg.) ; NURCAN, Selmin (Hrsg.) ; KROGSTIE, John (Hrsg.) ; SOFFER, Pnina (Hrsg.) ; PROPER, Erik (Hrsg.) ; SCHMIDT, Rainer (Hrsg.) ; BIDER, Ilia (Hrsg.): *Enterprise, Business-Process and Information Systems Modeling - 12th International Conference, BPMDS 2011, and 16th International Conference, EMMSAD 2011, held at CAiSE 2011, London, UK, June 20-21, 2011. Proceedings* Bd. 81, Springer, 2011 (Lecture Notes in Business Information Processing), S. 186–200

- [BFSZ12] BRÜNING, Jens ; FORBRIG, Peter ; SEIB, Enrico ; ZAKI, Michael: On the Suitability of Activity Diagrams and ConcurTaskTrees for Complex Event Modeling. In: ASEEVA, Natalia (Hrsg.) ; BABKIN, Eduard (Hrsg.) ; KOZYREV, Oleg (Hrsg.): *Perspectives in Business Informatics Research (BIR2012)* Bd. 128, Springer, 2012 (LNBIP), S. 54–69. – Best Paper Award
- [BG11] BRÜNING, Jens ; GOGOLLA, Martin: UML Metamodel-based Workflow Modeling and Execution. In: KUTVONEN, Lea (Hrsg.): *Enterprise Distributed Object Computing Conference (EDOC2011)*, IEEE Computer Society, 2011, S. 97–106
- [BGF10] BRÜNING, Jens ; GOGOLLA, Martin ; FORBRIG, Peter: Modeling and Formally Checking Workflow Properties Using UML and OCL. In: FORBRIG, Peter (Hrsg.) ; GÜNTHER, Horst (Hrsg.): *Perspectives in Business Informatics Research (BIR2010)* Bd. 64, Springer, 2010 (LNBIP), S. 130–145
- [BGHK12] BRÜNING, Jens ; GOGOLLA, Martin ; HAMANN, Lars ; KUHLMANN, Mirco: Evaluating and Debugging OCL Expressions in UML Models. In: BRUCKER, Achim D. (Hrsg.) ; JULLIAND, Jacques (Hrsg.): *Tests and Proofs - 6th International Conference, TAP 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings* Bd. 7305, Springer, 2012 (Lecture Notes in Computer Science). – ISBN 978–3–642–30472–9, S. 156–162
- [BHW11] BRÜNING, Jens ; HAMANN, Lars ; WOLFF, Andreas: Extending ASSL: Making UML Metamodel-based Workflows executable. In: CABOT, Jordi (Hrsg.) ; CLARISÓ, Robert (Hrsg.) ; GOGOLLA, Martin (Hrsg.) ; WOLFF, Burkhart (Hrsg.): *OCL and Textual Modelling (OCL2011)* Bd. 44, ECEASST, 2011
- [BKL12] BRÜNING, Jens ; KUNERT, Martin ; LANTOW, Birger: Modeling and Executing Concur-TaskTrees using a UML and SOIL-based Metamodel. In: BALABAN, Mira (Hrsg.) ; CABOT, Jordi (Hrsg.) ; GOGOLLA, Martin (Hrsg.) ; WILKE, Claas (Hrsg.): *OCL and Textual Modelling (OCL2012)*, ACM Digital Library (DL), 2012. – to appear
- [Brü09] BRÜNING, Jens: Declarative Workflow Modeling with UML Class Diagrams and OCL. In: ABRAMOWICZ, Witold (Hrsg.) ; MACIASZEK, Leszek A. (Hrsg.) ; KOWALCZYK, Ryszard (Hrsg.) ; SPECK, Andreas (Hrsg.): *Business Process, Services Computing and Intelligent Service Management, Leipzig, Germany, March 23-25, 2009* Bd. 147, GI, 2009 (LNI), S. 227–228
- [BW09] BRÜNING, Jens ; WOLFF, Andreas: Declarative Models for Business Processes and UI Generation using OCL. In: CABOT, Jordi (Hrsg.) ; CHIMIACK-OPOKA, Joanna (Hrsg.) ; GOGOLLA, Martin (Hrsg.) ; JOUAULT, Frédéric (Hrsg.) ; KNAPP, Alexander (Hrsg.): *The Pragmatics of OCL and Other Textual Specification Languages (OCL2009)* Bd. 24, ECEASST, 2009

1.3 Aufbau der Arbeit und Hinweise

Kapitel 2 erläutert die Grundlagen für diese Arbeit und geht auf die Modellierung in ausgewählten Gebieten der Informatik ein. Im speziellen wird die Geschäftsprozess- bzw. Workflowmodellierung und die etablierten Ansätze vorgestellt.

Kapitel 3 und 4 führen in das Konzept der deklarativen und metamodelbasierten Workflowmodellierung ein. Zunächst legt Kapitel 3 die Grundlagen dafür und stellt die für die Metamodellierung verwendete Sprachen und Diagramme vor. Anschließend folgt die Beschreibung des Metamodells zur Workflowmodellierung und spezifische Eigenschaften davon. Außerdem wird eine Workflow Pattern Analyse für die

neue Workflowmodellierungssprache gemacht. OCL-Invarianten zur Sicherstellung statischer Eigenschaften der Workflowmodelle ist ebenfalls Bestandteil des Kapitels. Daraufhin wird eine Daten- und Organisationsmodellierung in den Ansatz integriert. Schließlich zeigt das Kapitel, wie anhand des Metamodells und des UML-Werkzeugs Workflowmodelle erstellt werden und wie das Tool den Modellierer bei der Fehlersuche unterstützt.

Kapitel 4 behandelt die Modellausführung der neuen Sprache. Zunächst wird dafür die Relationen zwischen Metamodell, Workflowmodell und Workflowinstanz geklärt. Ein Plugin zur Erstellung von initialen Workflowinstanzen aus Workflowmodellen wird vorgestellt. Außerdem beschreibt eine *Toolchain* die verschiedenen Werkzeuge bzw. Plugins, die beginnend bei der Workflow-Modellerstellung bishin zur Analyse von abgearbeiteten Workflows genutzt werden können. Die UML-Action Language *ASSL* und die dafür implementierten Erweiterungen, die für die Workflowsausführungen notwendig sind, werden eingeführt. Die Workflowmodelle während der Ausführung und die Repräsentation im dafür implementierten Workflow Runtime-Plugin sind ebenfalls Bestandteil dieses Kapitels. Letztendlich lassen sich abgearbeitete Prozesse in diesem Ansatz auch analysieren. Verschiedene Mittel und Ansichten werden hier beschrieben.

Die Kapitel 5 und 6 beschäftigen sich mit dem hierarchieorientierten Ansatz zur Workflowmodellierung, die Aufgabenmodelle als Grundlage nutzen.

Kapitel 5 beschreibt zunächst bestehende, bereits etablierte hierarchische Ansätze zur Geschäftsprozessmodellierung. Daraufhin wird die Aufgabenmodellierung eingeführt und Vor- und Nachteile dieser Modelle im Kontext der Workflowmodellierung betrachtet. Transformationsregeln von Aufgabenmodellen zu Aktivitätsdiagramme zeigen die Verbindungen dieser Modellierungssprachen. Diese veranschaulichen außerdem die Strukturiertheit der Aufgabenmodelle. Ein Metamodellansatz zur Aufgabenmodellierung wird daraufhin eingeführt. Hierbei lassen sich die aus den Kapiteln davor diskutierten Vorteile und Tools des metamodellbasierten Ansatzes auch für Aufgabenmodelle nutzen. Die Entscheidungsmodellierung ist für Workflowmodelle sehr wichtig. Eine Diskussion, wie unterschiedliche Entscheidungsmodellierungen in Aufgabenmodellen aussehen können, wird als letztes in diesem Kapitel geführt.

In Kapitel 6 liegt der Schwerpunkt in der Ausführung der Aufgabenmodelle. Zunächst werden die Grundlagen für die Ausführung von Workflowmodellen insbesondere in Workflow Management Systemen in der Einleitung des Kapitels gegeben. Eine metamodellbasierte Ausführung mit dem UML-Tool *USE* und einem dafür entwickelten Runtime-Plugin ist daraufhin Thema der Betrachtung. Dann wird die Workflowsprache für das aufgabenbaumbasierte Workflow Management System *TTMS* vorgestellt und eine *Toolchain* vorgeschlagen, die bereits vorhandene Software in den Entwicklungsprozess für Workflows berücksichtigt. Der Workflow-Editor zur Erstellung der *TTMS*-Workflowmodelle und das *WfMS* zum Ausführen der Aufgabenmodelle wird daraufhin genauer betrachtet.

Schließlich fasst Kapitel 7 die Ergebnisse der Arbeit zusammen. Außerdem werden weitere zukünftige Arbeiten, die sich für die vorgestellten Ansätze anbieten, vorgestellt.

Ein Hinweis ist zu der verwendeten Umgangssprache zur Bezeichnung der Elemente bei den Metamodellen, Modellen und Constraints zu machen. Entsprechend der üblichen Konvention, dass Programmcode in Englisch zu schreiben und zu dokumentieren ist, wurde in dieser Arbeit auch Englisch als Umgangssprache für die Modelle verwendet. Die Publikationen, auf denen diese Dissertation basiert, sind überwiegend in Englisch und damit wurden diese Modelle übernommen. Ausnahmen stellen einige Modelle dar, die von anderen Quellen genommen wurden, wie z.B. in Abbildung 2.6.

Kapitel 2

Grundlagen: Modellierung in ausgewählten Gebieten

Modelle stellen Ausschnitte der Wirklichkeit abstrakt dar. Sie sind häufig domänenspezifisch und können zu unterschiedlichen Zwecken eingesetzt werden. Im Bereich der Informatik werden Modelle sehr vielseitig eingesetzt. In diesem Kapitel werden verschiedene Bereiche in der Informatik vorgestellt, in denen Modellierung eine wichtige Rolle spielt und die das Forschungsumfeld beschreiben, in der diese Arbeit geschrieben wurde. Dabei werden die Techniken und Ansätze, die in dieser Arbeit genutzt werden kurz vorgestellt. Es wird vor allem der Formalitätsaspekt von Modellen im Einsatz der Informatik und Wirtschaftsinformatik diskutiert. Daraufhin wird in Abschnitt 2.5 konkret auf den Bereich der Geschäftsprozessmodellierung eingegangen und bewährte Techniken beschrieben.

2.1 Modellierung in der Softwaretechnik

Modellierung ist bei der Erstellung von Software ein nützliches Hilfsmittel, um Software zu entwerfen und zu dokumentieren. Mit objektorientierte Techniken und Sprachen hat sich hier vor allem die Unified Modeling Language (UML) [UML10] durchgesetzt. Vor der Entwicklung objektorientierter Sprachen wurde bereits in den 70er Jahren zur Modellierung z.B. die Strukturierten Analyse (SA) [DeM79] entwickelt. Dort wurde mit Funktionen, Schnittstellen und Datenflüssen das zu entwickelnde Softwaresystem entworfen und dokumentiert. Die SA umfasst des Weiteren ein Hierarchiekonzept, das es ermöglicht, Funktionen nach bestimmten Regeln zu verfeinern.

Objektorientierte Sprachen und Modelle kamen später zum Einsatz. Anfang und Mitte der 90er Jahre haben sich viele objektorientierte Modellierungssprachen entwickelt. Von dieser Modellierungsweise wurde erwartet, dass angefangen von der Analyse über das Design bis zur Implementierung der Software durchgehend objektorientierte Sprachkonzepte genutzt werden können. Damit sollte eine modellgetriebenen Softwareentwicklung ermöglicht werden, in der Modelle eine zentrale Rolle spielen. Ende der 90er wurden die unterschiedlichen objektorientierten Modellierungssprachen zur Unified Modeling Language (UML) [UML10] zusammengefasst, die sich als allgemein akzeptierter Standard etabliert hat. Die UML ist ausschließlich eine Modellierungssprache und gibt keine Vorgehensweise vor, wie sie in der Softwareentwicklung einzusetzen ist. Damit ist auch nicht vorgegeben, wie intensiv und formal die UML vor der Systemimplementierung zu nutzen ist. Sie kann sehr intensiv zur Validation von Software eingesetzt

werden oder lediglich als grobe Skizze dienen. Jedoch ist anzumerken, dass die UML zu den semiformalen Spezifikationssprachen zählt, die eine Verifikation der zu erstellenden Software nicht ermöglicht.

Sehr formale Ansätze nutzen textuelle, an mathematischen Modellen angelehnte Spezifikationen, die durch die strengen Vorschriften und mathematischen Symbole für Laien schwer zu verstehen sind. Hier sind u.a. die algebraischen Spezifikation abstrakter Datentypen [BM04] oder das Hoare-Kalkül [Hoa69] zu nennen. Diese Ansätze haben den Anspruch bewiesene, korrekte Software zu erstellen, die jedoch äußerst schwierig zu bekommen ist. Die Gefahr ist groß, dass Fehler in den komplizierten Beweisen vorkommen. Anforderungen an die Software lassen sich außerdem anhand der komplizierten Spezifikationen nicht mit Stakeholdern diskutieren. Somit kann es passieren, dass von falschen Voraussetzungen ausgegangen wird und obwohl der Beweis korrekt geführt wurde kann eine Software entstehen, die nicht dem gewünschten Ergebnis entspricht.

Diese Ansätze sind des Weiteren äußerst teuer, da sehr lange Analysephasen vor der Implementierung durchgeführt werden müssen. Außerdem kann schwer während des Entwicklungsprozesses auf wechselnde Anforderungen der Endanwender an die Software reagiert werden, da sehr viele Analysedokumente und Beweise bei veränderten Anforderungen vorgenommen werden müssen. Zudem sind die Spezifikationen nicht als Dokumentation der Software geeignet, da sie i.d.R. schwerer zu lesen sind als der Programmcode selber, auf den sie sich beziehen. Eine ausführliche Diskussion der Vor- und Nachteile von formalen Methoden in der Softwareentwicklung, die u.a. einige der oben aufgeführten Probleme beinhaltet, ist in [Gog04] zu finden.

Die vielen Nachteile machen die streng formalen Methoden für weite Teile der Softwareentwicklung nicht effizient nutzbar. Für einzelne wenige Bereichen, wie z.B. Luft- und Raumfahrt, bei denen die korrekte Funktionsweise äußerst wichtig ist und wenige Anforderungen sich während des Entwicklungsprozesses ändern, finden die Methoden Anwendung. Dagegen ist gerade der Bereich der Wirtschaftsinformatik, bei denen Computer Nutzer bei der Arbeit unterstützen bzw. Arbeitsabläufe automatisieren sollen, für solche streng formalen Methoden nicht geeignet. In diesem Bereich hat man es mit permanenten Änderungen bei Geschäftsabläufen und Umstrukturierungen zu tun, so dass Software bzw. deren Entwicklungsprozess sehr flexibel sein muss.

In nicht formalen, agilen Entwicklungsmethoden für Software wird die Analysephase sehr kurz gehalten. Es werden höchstens sehr informelle, visuelle Modelle (z.B. UML) als Kommunikationsmittel in der Entwicklergruppe genutzt und es wird frühzeitig zur Implementierung übergegangen. Somit sind diese Methoden eher als codezentriert zu bezeichnen. Generell ist die Kommunikation in der Entwicklergruppe bei diesen Methoden sehr wichtig. Bei den Entwicklungsansätzen sind z.B. Extreme Programming [Bec00] und SCRUM [Pic07] zu nennen. Der Vorteil hier liegt darin, dass schnell Software produziert wird und der Endnutzer in die Entwicklung der Software intensiver mit eingebunden werden kann. Es kann rascher auf sich wechselnde Anforderungen reagiert werden. Diese Entwicklungsmethoden bergen aber auch große Gefahren. Da unmittelbar in die Implementierungsphase übergegangen wird, fehlt i.d.R. ein verlässliches Design. Ein sorgfältiger Einsatz von Design Patterns [GHJV02], der einen guten Softwareentwurf und damit wartbare und wiederverwendbare Software verspricht, ist damit nicht möglich.

Ein Kompromiss zwischen formalen und nicht formalen Ansätzen liegen in der semiformalen bzw. leichtgewichtigen formalen Softwarespezifikation. Hier werden die erstellten Modelle auf Eigenschaften geprüft, die sie erfüllen müssen. Damit wird sichergestellt, dass ein guter (objektorientierter) Entwurf vorliegt bevor in die Implementierungsphase weitergegangen wird. Diesen Ansätzen werden u.a. ermöglicht durch die UML-Werkzeuge USE [GBR07] und Alloy [Jac03]. Das Modell wird als UML-Klassendiagramm angegeben und Objektdiagramme repräsentieren Systemzustände zur Runtime. Diese können semiautomatisch von den Tools generiert werden und sind Testfälle für das Designmodell. Hierbei können zwei Arten von

Fehlern auftreten. Entweder enthält das Modell Fehler oder der Testfall. Diese Fehler kann der Entwickler mit Werkzeugen analysieren und daraufhin beseitigen.

Bei den modellzentrierten bzw. -getriebene Softwareentwicklung ist der bisher übliche Ansatz Modelle in Code von objektorientierte Programmiersprachen zu transformieren, um daraus lauffähige Software weiterzuentwickeln. Es haben sich hier jedoch weitere Ansätze entwickelt, die visuellen Modelle angereichert mit textuellen Spezifikationen direkt ausführen sollen. Damit kann eine visuelle Programmierung, die eine bessere Dokumentation und Erfassbarkeit der Software für den Systementwickler verspricht, realisiert werden. Z.B. werden mit der UML-Action Language Alf [OMG10] oder SOIL [Büt11] die Modelle ausführbar.

Bei der Softwareentwicklung ist bei großen Softwaresystemen darauf zu achten, dass eine verständliche Architektur entsteht. Hierfür haben sich Patterns stark verbreitet, welche bewährte Lösungen für konkrete Probleme unter Verwendung von Modellen bereitstellen. Design Patterns [GHJV02] sollen z.B. bewährte Lösungen für konkrete Probleme im objektorientierten Softwaredesign wiederverwendbar machen. Architekturmodelle und -patterns werden des Weiteren eine Abstraktionsebene höher zur Erstellung verständlicher und bewährter Softwarearchitekturen eingesetzt. Hier sind u.a. das Model-View-Controller Pattern [Ree79] bzw. das Seeheim-Modell [Tou07] zur Gestaltung von Softwaresystemen zu nennen.

2.2 Daten- und Metamodellierung

Datenmodellierung ist im Bereich der Datenbanken notwendig, um Datenbankentwürfe zu analysieren und zu diskutieren bevor sie implementiert werden. Die Analyse kann sich sowohl auf technische als auch fachliche Aspekte beziehen. Datenredundanzen sollen vermieden und adäquate Datenbezeichnungen gefunden werden. Hierfür werden typischerweise auch Fachexperten, die in der Regel keine IT-Experten sind, in den Datenentwurf mit einbezogen. Daher ist es wichtig, dass die Modellierungssprachen intuitiv und leicht verständlich sind, damit Systementwickler und Fachexperten eine Grundlage zur Diskussion haben. Hierfür wurde das Entity-Relationship-Modell [Che76] (ERM) entwickelt. Sind validierte Datenbankentwürfe erstellt, lassen sich daraufhin Transformationstechniken anwenden, um diese Modelle in Sprachen zu überführen, die Datenbank-Management-Systeme (DBMS) interpretieren. Analog zu ER-Modellen können auch UML-Klassendiagramme eingesetzt werden, um Daten und ihre Beziehungen visuell in einem Datenmodell zu beschreiben.

Es gibt auch textuelle nicht grafische Beschreibungsarten von Daten. Vor allem hat sich hier XML durchgesetzt, wofür Dokument Type Definitions (DTDs) gültige XML-Datensätze textuell definiert können. XML selbst kann mit XML-Schemas (XSD) ebenfalls zur Beschreibung von gültigen XML-Datensätzen eingesetzt werden. Bei der sehr verbreiteten Technologie der Webservices wird die XML-basierte Datenbeschreibung WSDL genutzt, um die Schnittstelle des Webservice zu spezifizieren. Damit werden die Daten und deren Datentypen, die zum Aufruf des Webservice erwartet werden und die, die zurückgeliefert werden, beschrieben. Webservices werden von Service Orientierten Architekturen (SOA) und der Business Process Execution Language (BPEL) dazu genutzt, um die gegenseitige verteilte Kommunikation aufzubauen, die auf XML-Nachrichten besteht. Zusätzlich werden noch Informationen zur Kommunikation wie z.B. Netzadressen mit WSDL bereitgestellt. Eine weitere textuelle Beschreibung von Daten gibt es mit dem Data Dictionary in der Strukturierten Analyse (SA). Diese Datenbeschreibung wird bei der SA genutzt, um Daten über Gleichungen ähnlich zur grammatikalischen Beschreibung der Backus-Naur-Form zu spezifizieren [DeM79].

In der Regel haben grafische Notationen bei der Modellierung gerade für ungeübte Nutzer Vorteile gegenüber rein textuelle Datenbeschreibungen. Hierfür ist die weite Verbreitung der ER-Diagramme oder UML-

Klassendiagramme zur Datenmodellierung ein Beleg. Geht es dann um die Umsetzung der grafischen Datenmodelle können z.B. die textuellen sehr populären XML-Techniken eingesetzt werden. Hier gibt es z.B. Transformationstechniken von UML-Klassendiagrammen zu XSD-Schemata [Mal03]. Diese Technik wird ebenfalls eingesetzt um mit XML in Verbindung mit Metamodellen Austauschformate für UML-Modelle festzulegen. Hat man ein werkzeugunabhängiges XML-basiertes Austauschformat, so lassen sich Modelle zwischen unterschiedlichen Tools austauschen.

Das UML-Metamodell beschreibt mit Sprachmitteln des UML-Klassendiagramms die eigene UML-Modellierungssprache [UML10]. Wird das UML-Metamodell in ein XSD-Schema transformiert, erhält man ein Austauschformat für UML-Modelle. Die dann erhaltenen XML-Dateien stellen UML-Modelle dar. Mit XMI wurde diese Vorgehensweise verfolgt, mit dem Ziel, ein verlässliches Austauschformat für UML-Modelle zu erhalten [XMI05].

Metamodelle werden aber nicht nur zur Definition von Austauschformaten im Zusammenhang mit XML genutzt. Sie können ebenfalls (domänenspezifische) Sprachen definieren [Kle08]. Analog dazu wie Backus-Naur-Form Grammatiken eingesetzt werden, um Programmiersprachen zu definieren, können Metamodelle Modellierungssprachen definieren. Aus einer Grammatik werden Programme und aus Metamodellen Modelle abgeleitet. In Metamodellen wird üblicherweise erst die strukturellen Eigenschaften der Modelle festgelegt. Die operationale Semantik der einzelnen Modellelemente bleibt zumindest beim Metamodell für die UML [UML10] unterspezifiziert. Diese Eigenschaft der Metamodelle ist analog zu Grammatiken für Programmiersprachen zu sehen, bei denen in der Regel die Semantik der einzelnen Kommandos auch noch nicht definiert ist.

Ist der von der Grammatik abgeleitete Compiler vorhanden, in dem zusätzlich auch die Semantiken der Kommandos umgesetzt sind, können Programme übersetzt und gestartet werden, die daraufhin im Computer ablaufen. Analog gibt es auf Modellseite auch ausführbare Modelle. Die dafür notwendigen Ausführungssprachen sind mit Alf [OMG10] oder SOIL [Büt11] für die UML vorhanden, die die operationale Semantik für Modelle definieren. Mit einem entsprechenden UML-Tool lassen sich diese Modelle dann ausführen und testen.

Mit Query View Transformation (QVT) [QVT08] und diverse Tools können in Verbindung mit Metamodellen Transformationen definiert werden, die Veränderungen des Systemzustands bewirken. Bei solchen Transformationen wird das gleiche Metamodell für Quelle und Ziel genutzt. Ursprünglich ist QVT jedoch dafür vorgesehen, verschiedene Metamodelle als Quelle und Ziel zu nutzen. Damit soll auf Basis von Metamodellen ermöglicht werden, Modelle einer Sprache in eine andere zu transformieren (z.B. von UML-Klassendiagramme zu Entity-Relationship-Diagramme). Seiteneffektfreie Sprachen wie OCL [OCL10] reichen für die Umsetzung von Modelltransformationen nicht aus, so dass für QVT eine imperative Version von OCL entwickelt wurde um die Modelltransformationen umzusetzen [QVT08].

Metamodelle kann man des Weiteren auch zur Beschreibung von textuell dargestellten Sprachen nutzen. So ist für die OCL selber ein UML-basiertes Metamodell entwickelt worden, um die Syntax der textuellen Sprache zu beschreiben [Ric01]. Es gibt noch viele weitere Bereiche, in denen Metamodelle Anwendung finden, die hier jedoch nicht weiter aufgeführt werden. Die Einsatzmöglichkeiten von Metamodellen reichen von sehr informellen Ansätzen, die rein konzeptionell Sachverhalte ausdrücken zu eher formalen Ansätzen, die eine modellzentrierte Entwicklung mit Modelltransformationen beinhaltet.

2.3 Modellierung in der HCI mit nutzerzentrierten Entwurfstechniken

Diverse nichtfunktionale Anforderungen sind hier zu erfassen, mit dem Ziel, die zu entwickelnde Software für die Endanwender nützlicher und benutzbarer zu machen. Diese Betrachtung kann modellbasiert anhand von mehreren Modellarten geschehen. Beispielsweise werden nach [Sta94] die vier Modelle: Aufgabe-, Benutzer-, Interaktions- und Geschäftsobjektmodell zur Modellierung von interaktiven Systemen benötigt, die in bestimmten Relationen zueinander stehen.

Das Aufgabenmodell ist dabei das zentrale Modell. Es beschreibt die statischen und organisatorischen Aspekte von Arbeit. Die statischen Eigenschaften werden durch die hierarchische Darstellung widerspiegelt. Jede weitere Ebene im Baum stellt eine Dekomposition und damit ein Zerlegen von Aufgaben bzw. Problemen dar. Dieser Vorgang kann als eine natürliche Herangehensweise zur Lösung von Problemen angesehen werden [MPS02]. Die dynamischen Eigenschaften werden über temporale Beziehungen innerhalb des hierarchischen Modells dargestellt [For07]. Viele Sprachen zur Aufgabenmodellierung haben eine grafische Syntax und sind damit intuitiver als textuelle Spezifikationen. So können sie als mentale Modelle zum Ausführen von Aufgaben angesehen werden.

Das Geschäftsobjektmodell umfasst Objekte und Daten aus dem Anwendungsgebiet, die durch die im Aufgabenmodell spezifizierte Arbeit genutzt bzw. verändert werden. Vorbedingungen und Effekte im Aufgabenmodell stellen die Verbindung zum Geschäftsobjektmodell her, welche üblicherweise textuell modelliert sind. Eine grafische Repräsentation ist durch explizite Modellelemente in den meisten Sprachen nicht vorgesehen.

Interaktionsmodelle spezifizieren die Mensch-Maschine-Schnittstelle. Damit wird die Struktur und das Verhalten interaktiver Elemente, ihre Fähigkeiten und Modalitäten spezifiziert [For07]. Hierfür gibt es z.B. Dialogmodelle, um die Folge von GUI-Fenstern und Elementen zu spezifizieren. Aufgabenmodelle lassen sich mit Dialogmodellen koppeln [FR07]. Mit Aufgabenmodellen wird eine zielorientierte Aktivitätsmodellierung angegeben [Ann03]. Durch die Kopplung kann man somit versuchen, Dialogspezifikationen zu erreichen, die den Nutzer zielorientiert zur Vollendung seiner Aufgabe führt.

Im Benutzermodell spielt der Nutzer und seine Fähigkeiten und Kompetenzen eine zentrale Rolle. Außerdem ist die Einordnung des Nutzers in die Arbeitsorganisation zu beachten.

Mit dem Verbinden der vier Modelle wird eine ganzheitliche nutzerzentrierte Modellierung von Arbeit verfolgt. Aufgaben, Arbeitsgegenstände, Arbeitsmittel und nicht zuletzt der Nutzer und seine Fähigkeiten stehen im Fokus. Es wird daraufhin auf Grundlage der Modelle untersucht, inwieweit und wie die zu entwickelnde Software den Nutzer bei der Erledigung seiner Aufgaben unterstützen kann.

Es wird versucht, die Aspekte über Modelle vor der Systementwicklung zu erfassen. Im Gegensatz zu modellzentrierten Ansätzen steht jedoch der Endanwender im Fokus der Entwicklung. Modelle werden häufig als Diskussionsgrundlage genutzt, um die nichtfunktionalen und funktionalen Anforderungen für die Software zu diskutieren. Dafür müssen die Modelle verständlich und intuitiv sein. Die diskutierten weichen Aspekte sind des Weiteren schwer zu formalisieren, um sie sinnvoll für (automatische) Transformationen zur modellgetriebenen Softwareentwicklung weiterzuverwenden. Somit wird im HCI-Bereich häufig informelle bis semiformale Spezifikationssprachen genutzt. Formelle, textuelle Modelle sind dagegen für diese Einsatzzweck eher nicht geeignet, wobei es auch Arbeiten in diese Richtung gibt. Bspw. wurde ein modellbasierter Ansatz zur (semi-)automatischen Generierung von Benutzeroberflächen unter Verwendung der oben beschriebenen Modellarten in [Wol11] verfolgt.

Bei der Verwendung von Modellen ist des Weiteren zu beachten, dass sich gerade bei den hier betrach-

teten Aspekten nicht alle vor der Systemimplementierung einplanen lassen. Häufig ist die Software in Benutzung und eine iterative Herangehensweise in der Softwareentwicklung erforderlich, um eine dem Endnutzer nützliche und gut nutzbare Software bereitzustellen [May99]. Das Vorgehen für einen solchen Softwareentwicklungsprozess ist hier jedoch nicht weiter Gegenstand der Betrachtung.

2.4 Modellierung von Unternehmen und betrieblicher Informationssysteme

Modelle spielen eine entscheidende Rolle in der Unternehmensmodellierung. Im Gegensatz zur Modellierung in der Softwaretechnik geht es in diesem Bereich darum, nicht nur technische sondern vor allem auch soziale und organisatorische oder wissensorientierte Aspekte des Unternehmens auszudrücken. Somit ist der Einsatz der Modelle hier analog zu den Modellen aus Abschnitt 2.3 zu sehen, in denen auch weitere Aspekte als die rein technischen, die die Umsetzung betreffen, ausgedrückt werden.

Modelle können in diesem Bereich z.B. rein zur Diskussion genutzt werden, um die Organisation bzw. Arbeitsprozesse im Unternehmen umzustellen und zu optimieren. Dieser Art des Einsatzes von Modellen unterscheidet sich grundlegend von dem zur Softwareentwicklung aus Abschnitt 2.1. Jedoch können die Modelle auch als Grundlage zur Entwicklung von geeigneten Informationssystemen genutzt werden, um die Arbeit im Unternehmen optimal zu unterstützen. Hier können die Modelle als Anforderungsdokumente für die Softwareentwicklung genutzt werden.

Formal sollten die Modelle hier nicht sein, da diese die Diskussion mit den betroffenen Personen behindert. Die ist jedoch notwendig, um zu einem guten Lösungsvorschlag für das Unternehmen zu kommen. Außerdem ist das Umfeld ein hochdynamisches, welches sich sehr schnell ändern kann. Damit müssen die Modelle ebenfalls schnell änderbar sein, wenn sie im Unternehmen längerfristig eingesetzt und gelten sollen. In diesem Umfeld können sich die Anforderungen an die Software ebenfalls schnell ändern. Somit gelten die in Unterabschnitt 2.1 diskutierten Nachteile der streng formalen Methoden gerade hier.

Um bewährte Lösungen und Wissen in der Analysephase wiederzuverwenden gibt es Analyse-Patterns. So hat Fowler in [FCJ96] einige Patterns identifiziert, die häufig verwendete Abbildungen der Wirklichkeit widerspiegeln und damit Modellierungsvorschläge für künftige Modelle in anderen Unternehmen bzw. Kontexten darstellen. Der Fokus der Modelle liegt nicht auf der technischen Seite zur Implementierung der Systeme sondern behandelt den Aspekt, wie man Ausschnitte der Wirklichkeit in objektorientierten Modellen ausdrückt. Patterns gibt es des Weiteren bei der Workflowmodellierung [AHKB03] und in dem Zusammenhang auch mit Ressourcen- [RHEA04b] und Datenmodellierung [RHEA04a]. Die Anwendung und Verbindung der verschiedenen Modelle im Workflowkontext ist ursprünglich im ARIS-Ansatz (Architektur integrierter Informationssysteme) zu finden [Sch02, Sch01]. Dort wird das Unternehmen in die folgenden unterschiedliche Modelle dargestellt: Daten-, Organisations-, Funktions-, Leistungs- und Steuerungs- bzw. Geschäftsprozessmodell. Das zuletzt genannte Modell wird mit *ereignisgesteuerten Prozessketten* dargestellt und verwendet bzw. verbindet alle davor genannten Modelle. Dieser Ansatz findet sowohl im akademischen Bereich als auch in der Praxis breite Anwendung.

Im Bereich der Wissensmodellierung für Unternehmen wird versucht, Wissen modellbasiert abzubilden, um z.B. Nutzer beim Erfüllen ihrer Aufgaben mit einem betrieblichen Informationssystem zu unterstützen. Objektorientierte Modellierungsweisen sind zur Wissensrepräsentation weniger gut geeignet und finden somit hier wenig Anwendung. Es werden hier eher Techniken eingesetzt, die im *semantik Web* anzufinden sind, wie z.B. Ontologien [MMS⁺03]. Hier gibt es Ansätze, Ontologien mit UML-Klassendiagrammen zu visualisieren [BBC⁺10]. Ontologien können des Weiteren auch eingesetzt werden, um Geschäftsprozess-

modelle auszudrücken [TF07]. Auch beim ARIS-Ansatz ist Wissensrepräsentation und -management ein Thema [Sch02]. Diese Arbeit geht hier nicht weiter auf dieses Gebiet ein und beschränkt sich im Folgenden auf die Geschäftsprozessmodellierung.

2.5 Geschäftsprozessmodellierung

Um den Bereich näher zu betrachten, wird in diesem Abschnitt die Modellierung und Sprachen näher vorgestellt. Modelle werden in diesem Bereich zu unterschiedlichen Zwecken eingesetzt. Der Abschnitt 2.5.1 nimmt eine Kategorisierung vor. In diesem Zusammenhang wird auf die Begrifflichkeiten *Geschäftsprozess-* und *Workflowmodellierung* eingegangen. Daraufhin werden in Abschnitt 2.5.2 etablierte Sprachen und deren spezielle Eigenschaften und Unterschiede betrachtet. Schließlich beschreibt Abschnitt 2.5.3 bestehende Metamodelle in diesem Bereich und wofür diese eingesetzt werden.

2.5.1 Kategorisierungen von Geschäftsprozessmodellen

Der Einsatz von Geschäftsprozessmodellen lässt sich in beliebige Kategorien einteilen. In verschiedenen Büchern zur Geschäftsprozessmodellierung und -management findet man unterschiedliche Klassifizierungen. So hat Gadatsch Geschäftsprozessmodelle in *datenflussorientierte*, *kontrollflussorientierte* und *objektorientierte* Herangehensweisen eingeteilt [Gad07].

Weske hat in [Wes07] dagegen eine Klassifizierung von Geschäftsprozessmodellen vorgenommen, die konzeptionellen („organizational“) von operationalen („operational“) Modellen unterscheidet. Diese Einteilung ist ähnlich zu der in diesem Abschnitt vorgestellten Kategorien *nicht ausführbare* und *formale, ausführbare* Modelle. Außerdem gibt es Modellierungssprachen, die entweder eine intraorganisationale („intraorganizational“) oder organisationseinheitsübergreifende („Process Choreographies“) Modellierung verfolgen. Davon unabhängig können Prozessmodelle am Grad der Automatisierung, Wiederholungsrate (Anzahl der Prozessinstanzen) und Strukturierung klassifiziert werden.

Es gibt offensichtlich keine allgemeingültigen Klassifizierungen für Geschäftsprozessmodelle. Im Folgenden wird daher eine eigene an diese Arbeit ausgerichtete Kategorisierung für Geschäftsprozessmodelle vorgenommen.

- **Nicht ausführbare Modelle zur Geschäftsprozessoptimierung und Anforderungsanalyse:** Nicht ausführbare Modelle werden häufig zur Diskussion mit Anwendern genutzt. Bei diesen Modellen gibt es Mehrdeutigkeiten, die eine Ausführung der Modelle verhindern. Als prominente Beispiele sind hier vor allem EPKs, UML-Aktivitätsdiagramme und BPMN zu nennen. Der Vorteil dieser Sprachen liegt darin, dass sie visuell, verständlich und intuitiv sind und damit zur Diskussion zwischen Fachexperten untereinander, Beratern oder Systementwickler geeignet sind.

In den Modellen wird vor allem verdeutlicht, welcher Weg der Prozess durch das Unternehmen nimmt und welche Organisationseinheiten und Rollen mit dem Prozess in Berührung kommen. Prozesse können anhand von Modellen umorganisiert werden. Mit Ist- und Soll-Modellen lassen sich Vor- und Nachteile diskutieren. Hierbei müssen die Modelle nicht zwangsläufig computergestützte Geschäftsprozesse beschreiben. Es können auch Geschäftsprozesse ohne jeglicher Beteiligung von Computern umorganisiert werden. Jedoch können sie auch zur Anforderungsanalyse für künftige Software verwendet werden. Es kann damit u.a. analysiert werden, welche Aktivitäten vom Computer übernommen werden können und welche Daten dieser zur Erledigung der Aufgabe braucht.

Die Modelle können dann als Vorlage zur Systemimplementierung genutzt werden, um damit den Geschäftsprozess zu teilautomatisieren und zu optimieren.

- **Formale und ausführbare Modelle zur Analyse dynamischer Eigenschaften:** Hier sind vor allem Petri-Netze und Prozessalgebren zu nennen. Diese Formalismen sind sehr formale Modellierungssprachen. Mehrdeutigkeiten kommen hier nicht vor und damit sind diese Modelle ausführbar. Ein Kritikpunkt ist, wie bei allen sehr formalen Spezifikationsformalissen, dass sie für Fachexperten nicht gut lesbar sind. Zur Diskussion für Umorganisationen von Geschäftsprozessen sind diese Formalismen somit nicht gut geeignet.

Petri-Netze wurden ursprünglich dafür entwickelt, technische Prozesse zu modellieren [Pet62]. Jedoch werden sie inzwischen in vielen weiteren Bereichen eingesetzt, wie z.B. zur Modellierung von Workflows. Sie wurden u.a. verwendet, um die Workflow Patterns formal auszudrücken [RHAM06]. Anhand des Werkzeugs CPN Tools sind diese Modelle dann auch ausführbar. Durch die Formalität lassen sich strukturelle Eigenschaften, wie Deadlockfreiheit und Soundness [AS11] überprüfen. Auch dynamische Eigenschaften lassen sich anhand von Testläufen der Modelle testen [AS11]. Hierdurch versucht man z.B. eine ideale Ressourcenauslastung und damit eine Kostenoptimierungen zu erlangen.

Prozessalgebren sind weitere formale Methoden, Prozesse zu spezifizieren. Zu diesen Formalismen gehören CCS, CSP, das π -Kalkül und LOTOS. So wurde in [PW05] gezeigt, dass die Workflow Patterns sich auch mit dem π -Kalkül ausdrücken lassen. Die Prozessalgebra LOTOS wird auch in Verbindung mit Aufgabenmodellierungssprache CTT verwendet [MPS02]. Die Prozessalgebraerme sind dort in den Aufgabenbaum integriert. Diese Modellierung wird Teil der hierarchieorientierten Workflowmodellierung in Kapitel 5 und 6 sein.

- **Modelle für Workflow Management Systeme und Ausführung für konkrete Geschäftsprozesse:** Workflow Management Systeme stellen generische Softwaresysteme dar, die eine web-basiert verteilte Ausführung der Geschäftsprozesse ermöglicht. Die Systeme stellen Schnittstellen für die Nutzer bereit, die benötigte Daten in den entsprechenden Aktivitäten eingeben müssen. Aktivitäten können automatisch den Angestellten zugewiesen werden, die dann die Ausführung dieser Aufgabe übernehmen.

Die Grundlage zur Ausführung der Geschäftsprozesse bilden Workflowmodelle. Hier wird zwischen *Designtime* und *Runtime* unterschieden. Während der *Designtime* werden die Prozesse anhand eines Modellierungswerkzeugs modelliert. Das WfMS instanziiert diese Modelle und daraufhin werden die Prozesse in der *Runtime* für ein konkreten Fall ausgeführt.

Der große Vorteil von Workflow Management Systemen liegt darin, dass sie sehr flexibel sind. Wenn ein Geschäftsprozess verändert werden soll, muss lediglich das Prozessmodell verändert werden und dem WfMS übergeben werden. Wenn man dagegen fest codierte Software zur Umsetzung eines Geschäftsprozesses verwendet, lassen sich Veränderungen nur deutlich schwerer umsetzen.

Einen ähnlichen Ansatz verglichen zu WfMS hat die Sprache BPEL [BPE07]. Durch BPEL-Modelle werden Geschäftsabläufe basierend auf Webservices automatisch von einer BPEL-Engine ausgeführt. Diese nimmt die XML-basierten BPEL-Modelle entgegen und kann diese ausführen. BPEL fokussiert sich aber auf automatisch ablaufende Geschäftsprozesse, in denen die Interaktion zwischen BPEL-Engine und dem Angestellten vernachlässigt und in Web Services verlagert wird.

Basierend auf diesen Kategorien lassen sich die Begrifflichkeiten *Geschäftsprozess-* und *Workflowmodellierung* diskutieren. Eine Abgrenzung dieser Begriffe ist schwierig, wie Gadatsch in [Gad07] bereits analysiert hat.

In [Hol98] wird der Begriff *Workflow* wie folgt definiert: „*The computerised facilitation or automation of a business process, in whole or part.*“ Nach dieser Interpretation sind also Workflows eine Teilmenge von Geschäftsprozessen. Der Fokus der Workflowsprachen liegt auf der computergestützten Ausführung von Geschäftsprozessen. Nach dieser Interpretation gehören Sprachen, die nicht ausführbar sind (die erste oben aufgeführte Kategorie), wie z.B. EPKs oder UML-Aktivitätsdiagramme eher nicht zu der Klasse der Workflowsprachen. Diese Sprachen sind reine Geschäftsprozessmodellierungssprachen.

Bei der BPMN, die ursprünglich zur ersten obigen Kategorie gehörte, gibt es Arbeiten, die versuchen, ihr eine operationale Semantik zu geben und damit die Modelle ausführbar zu machen [DG11]. In der BPMN-Spezifikation ist dafür ein Kapitel vorhanden, in dem die operationale Semantik der Sprache umgangssprachlich beschrieben wird [BPM11]. Ähnliche Ansätze sind bereits bei der UML-Spezifikation für Aktivitätsdiagramme zu finden [UML10, Kapitel 12]. Auch bei EPKs gibt es Ansätze, ihr eine präzise operationale Semantik zu geben [Men08]. Mit diesen Arbeiten nähern sich die ursprünglich rein konzeptionellen Sprachen den ausführbaren Workflowsprachen an. Dieses sind Beispiele dafür, dass die beiden Begriffe *Geschäftsprozess-* und *Workflowmodelle* zunehmend synonym angewendet werden.

Der Fokus der in dieser Arbeit verwendeten Sprachen liegt vor allem auf der Ausführung der Modelle. Eine präzise, operationale Semantik ist bei den eingeführten Ansätzen hinterlegt. Daher wurde für den Titel diese Arbeit der Begriff *Workflowmodellierung* verwendet. Der andere Begriff *Geschäftsprozessmodellierung* hätte aus den beschriebenen Gründen aber durchaus auch genutzt werden können.

2.5.2 Etablierte Modellierungssprachen

Populär wurde die Geschäftsprozessmodellierung mit Ereignisgesteuerten Prozessketten (EPKs) [KNS92] im Jahre 1992 in Verbindung mit der Modellierung zur Architektur integrierter Informationssysteme (ARIS) [Sch01]. Daraufhin wurde Ende der 90er die objektorientierte Modellierung mit UML sehr populär und damit die Verwendung der Aktivitätsdiagramme. Weitere Anforderungen an Prozessmodellierungssprachen sind im Laufe der Zeit aufgetreten. Beispielsweise sollte die Modellierung von Unternehmensgrenzen und Kooperation zwischen verschiedenen Unternehmen in der Modellierungssprache ausdrückbar sein. In 2004 wurde die BPMN [Whi04] erfunden, die entsprechende Erweiterungen aufgegriffen hat (s. Abschnitt 2.5.2.3). Die BPMN hat sich inzwischen als de facto Standard zur Geschäftsprozessmodellierung durchgesetzt.

Im Allgemeinen wurde den Modellierungssprachen in der Reihenfolge ihrer Entstehung Ausdrucksmächtigkeit hinzugefügt. Betrachtet man die Modellierungssprachen jedoch im Detail, weisen sie semantisch deutliche Unterschiede auf. Spracheigenschaften und einige Unterschiede werden im Folgenden an Beispielen aufgeführt.

Nach den konzeptionellen Modellierungssprachen EPKs, Aktivitätsdiagramme und BPMN werden im Folgenden mit BPEL, YAWL und ADEPT noch Sprachen betrachtet, die zur Ausführung von konkreten Geschäftsprozessen geeignet sind. Diese sind sowohl in der Forschung als auch in der Praxis renommierte Sprachen, die mit entsprechenden Softwaresystemen angewendet werden. Schließlich wird noch ein deklarativer Ansatz zur Beschreibung von Geschäftsprozessen auf Grundlage von logischen Formeln mit DECLARE vorgestellt.

2.5.2.1 Ereignisgesteuerte Prozessketten

Die EPK wurde an der Universität des Saarlandes am Institut für Wirtschaftsinformatik entwickelt [KNS92]. Die Modellierungssprache wurden u.a. für den Entwurf, Dokumentation und Konfiguration der Geschäftsabläufe für die SAP Software genutzt und wurde damit sehr populär. Die sehr umfangreichen SAP-Referenz-

prozesse wurden mit EPKs dokumentiert [Men08]. In Abbildung 2.1 ist eine Ereignisgesteuerte Prozesskette als Beispiel angegeben, die einen Prüfungsprozess für Kreditanträge modelliert.

Der Prozess beginnt mit zwei Startereignissen *Credit application arrived* und *Customer advised*, die zusammen auftreten müssen, damit der Prozess beginnen kann. Daraufhin wird der Antrag in einer Aktivität geprüft. Dort wird des Weiteren entschieden, welcher alternative Pfad im Prozessmodell zu nehmen ist. Wenn der Kunde kreditwürdig ist, ist der Antrag anzunehmen, ansonsten abzulehnen. Letztendlich ist der Kunde noch über die Entscheidung zu informieren.

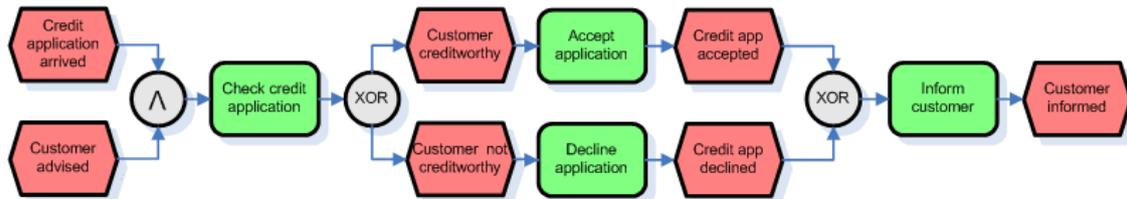


Abbildung 2.1: Eine Kreditantragsprozesse modelliert mit einer einfachen Ereignisgesteuerten Prozesskette

Die wichtigsten Bestandteile der Modellierung sind Ereignisse und Funktionen. Ereignisse und Funktionen werden alternierend in einer Kette mit Pfeilen verknüpft. Zudem können mit Konnektoren Prozesspfade verzweigt und wieder zusammengeführt werden. Prozesse beginnen immer mit Startereignissen und enden mit Endereignissen. Aktivitäten sind dazwischen zu spezifizieren. Startereignisse spezifizieren die Situation, in der der Prozess zu instanzieren ist. Damit wird modellinhärent angegeben, wann ein neuer Prozess startet [DM09]. Die EPK wurde des Weiteren in das ARIS-Modellierungskonzept zur Unternehmensmodellierung eingebettet, was in Abbildung 2.2 zu sehen ist.

Die sehr einfache Modellierung und wenigen Modellierungsmittel machen die Sprache auch für Fachexperten sehr gut verständlich. Außerdem kann man durch die Bezeichnung des Ereignisses hinter einer Funktion spezifizieren, wann eine Aufgabe erledigt ist. Ereignisse können zudem visuell die Vor- und Nachbedingungen einer Funktion ausgedrückt. Eine weitere Eigenschaft der EPKs ist die Entscheidungsmodellierung. Semantisch wird bei EPKs vorgegeben, dass Entscheidungen aktiv in Funktionen zu treffen sind [KNS92]. Somit muss unmittelbar vor einem Entscheidungskonnektor eine Funktion stehen. Ein Ereignis ist nicht erlaubt. Diese Modellierungsvorschrift hat Konsequenzen auf die Bezeichnung der Entscheidungsfunktion. Typischer Weise werden die Aktivitäten *prüfe*, *untersuche* oder *entscheide* bezeichnet.

Bei den erweiterten Ereignisgesteuerten Prozessketten (eEPKs) werden weitere Modellelemente aus weiteren Modellen des ARIS-Ansatzes integriert. Dieser umfasst weitere Modelle wie das Daten-, Funktions-, Organisations- und Leistungsmodell. In Abbildung 2.2(a) ist das ARIS-Haus als konzeptionelles Bild visualisiert. Das Geschäftsprozessmodell als eEPK verbindet die Modelle miteinander. In Abbildung 2.2(b) ist ein Organisationsmodell-Beispiel angegeben. Das Organisationsmodell wird hierarchisch dargestellt. Das Unternehmen als ganzes wird in der Wurzel repräsentiert. Daraufhin werden die Abteilungen darunter modelliert. In Abbildung 2.2(b) gliedert sich die Bank in drei Abteilungen. In der Abteilung *Business* sind des Weiteren noch zwei Rollen assoziiert.

Ein Beispiel für das Datenmodell ist in Abbildung 2.2(c) angegeben. Ursprünglich sind in der ARIS-Methode Entity-Relationship-Diagramme zur Datenmodellierung zu verwenden. Im aktuellen ARIS-Toolset werden aber auch UML-Klassendiagramme zur Datenmodellierung unterstützt, welche Diagrammart auch im Beispiel von Abbildung 2.2(c) angewendet wurde. Die benötigten Daten für den Prüfungsantrag für Kreditanträge sind hier mit den Klassen *Customer*, *CustomerInformation*, *CreditApplication*, *Notification* und den gegenseitigen Beziehungen modelliert.

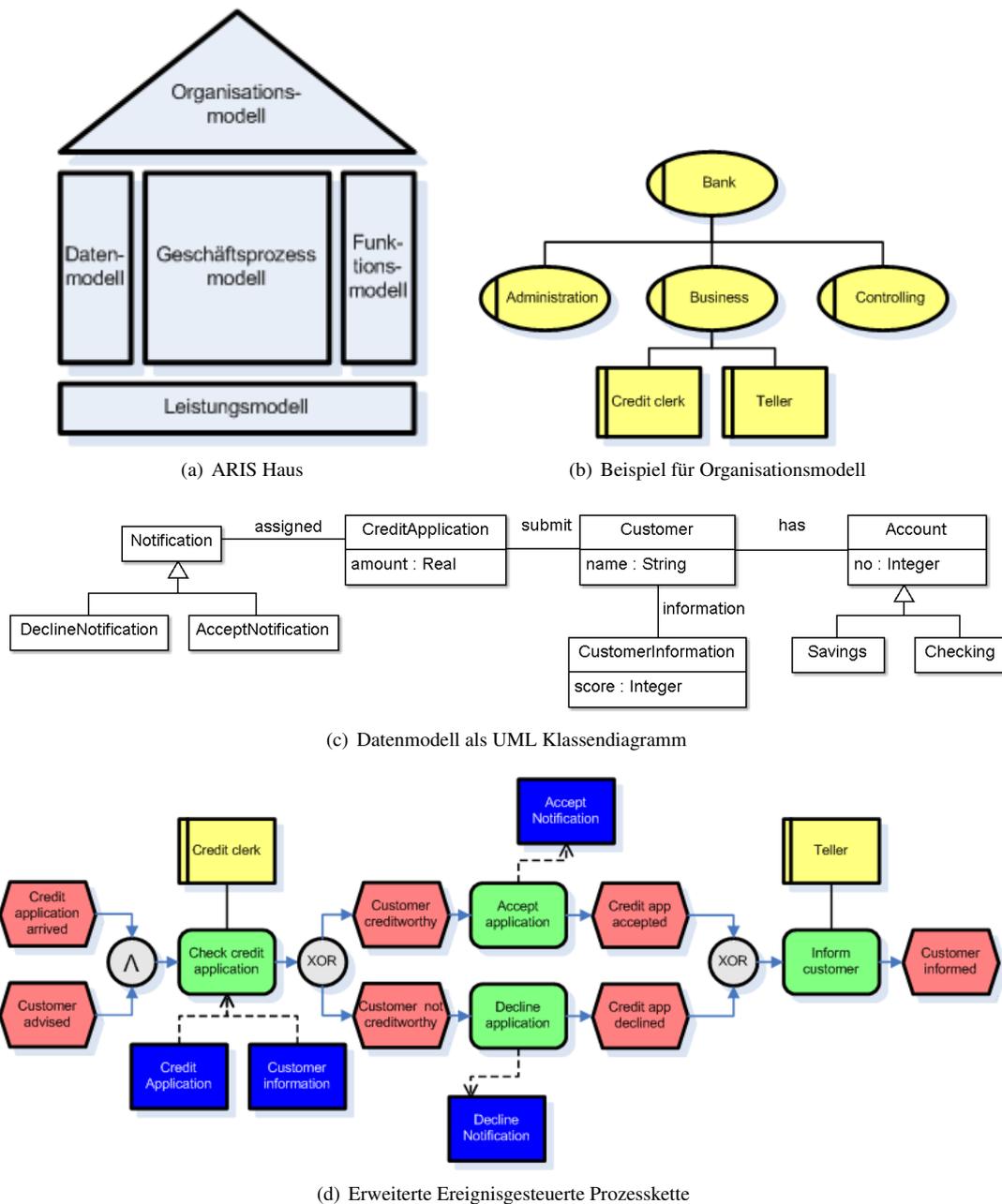


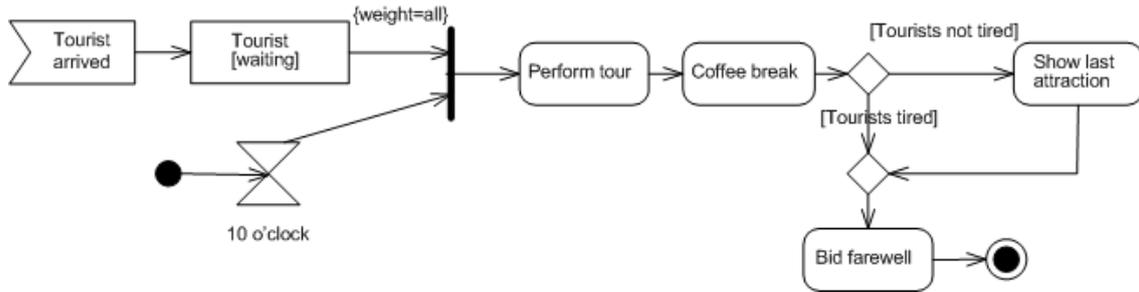
Abbildung 2.2: Das ARIS Konzept und Beispielmodelle dafür

In der eEPK von Abbildung 2.2(d) ist zu erkennen, dass Elemente sowohl aus dem Organisationsmodell als auch aus dem Datenmodell genutzt werden. Die eEPK dient damit als Geschäftsprozessmodell, das zudem visuelles die unterschiedlichen ARIS-Modelle zusammenführt [Sch02].

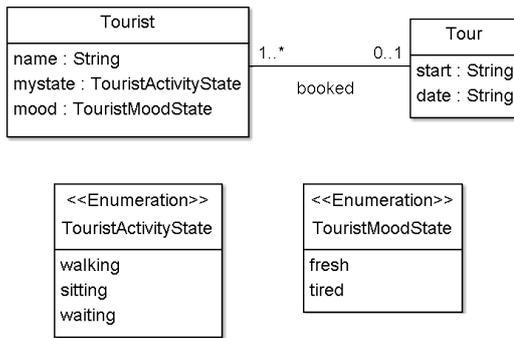
2.5.2.2 UML-Aktivitätsdiagramme

Aktivitätsdiagramme sind nicht ausschließlich mit dem Fokus auf die Geschäftsprozessmodellierung entwickelt worden. Sowohl technische als auch Geschäftsprozesse lassen sich damit modellieren. Die strikte alternierende Modellierung von Aktivitäten und Ereignissen, die von den EPKs bekannt ist, wird hier nicht

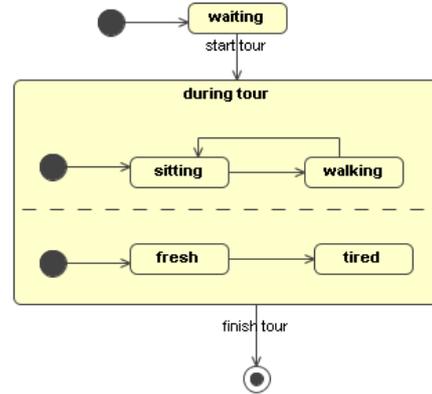
verfolgt. Bei Aktivitätsdiagrammen können Aktivitäten direkt aneinandergereiht werden. Die Aktivitätsdiagramme können in der UML mit diversen anderen Diagrammen ähnlich zu den eEPKs im ARIS-Ansatz verknüpft werden. Abbildung 2.3 liefert ein Beispiel als Aktivitätsdiagramm mit zugehörigen weiteren Diagrammen der UML. Es wurde hier ein anderes Beispiel als bei der Einführung der EPKs in Abschnitt 2.5.2.1 gewählt, um spezifische Eigenschaften der Aktivitätsdiagramme zu zeigen.



(a) Eine Städtetour als Aktivitätsdiagramm



(b) Ein zugehöriges Klassendiagramm



(c) Ausschnitt eines Objektlebenszyklus eines *Tourist* Objekts

Abbildung 2.3: Aktivitätsdiagramm und zugehörige UML Klassen- und Zustandsdiagramme

In Abbildung 2.3(a) ist ein Touristentour als Geschäftsprozess modelliert. Das Modell ist in dieser Detaillierung und den verwendeten Modellierungselementen so nicht mit EPKs oder BPMN modellierbar. Es werden Events in Verbindung mit Objektflüssen verwendet, die spezifisch für Aktivitätsdiagramme sind [BFSZ12]. Nachdem der Prozess instanziiert ist und das Ereignis *Tourist arrived* eintritt, wird der Tourist im Objektknoten *Tourist* gespeichert. Nachdem das Event *10 o'clock* eingetreten ist, werden die bis dahin eingetroffenen Touristen mit der gewichteten Kante eingesammelt und die Aktivität *Perform tour* gestartet.

Bei Aktivitätsdiagrammen sind Objektflüsse als Modellierungsmittel eingeführt und damit Objektspezifikationen direkt in Geschäftsprozessmodellen integriert worden. In den Objektknoten kann mit eckigen Klammern zusätzlich der Zustand des modellierten Objektes angegeben werden (z.B. *Tourist [waiting]*). Damit wird unmittelbar auf Objektzustände im Objektlebenszyklus, die in einem UML-Zustandsautomaten modelliert werden (s. Abbildung 2.3(c)), verwiesen. Objektflüsse lassen sich sehr gut zur Geschäftsprozessmodellierung verwenden. In [For07] wurde z.B. aufgezeigt, wie mit Aktivitätsdiagrammen und Objektflüssen eine EPK ähnliche Modellierung erfolgen kann.

Obwohl Objektflüsse eine gute Integration von Objektinformationen in den Geschäftsprozessmodellen

darstellen, bergen sie jedoch semantische Probleme. Die an die Petri-Netz angelehnte Semantik der Aktivitätsdiagramme besagt, dass erst nach Beenden der Tätigkeit das Objekt die Aktivität verlässt. Mit dieser Eigenschaft lassen sich nicht parallel laufende Aktivitäten modellieren, die Daten austauschen müssen. Das *stream* Konzept ist in Verbindung mit Objektflüssen für Geschäftsprozessmodellierung eher schlecht geeignet. Es besagt, dass ein Strom von Objekten die Aktivität verlässt, solange diese aktiv ist. Dieses Konzept ist eher für technische Prozesse zur Spezifikation von Datenübertragungen geeignet.

Mit Swimlanes wurden Modellierungselemente aufgenommen, die sich wiederum sehr gut zur Geschäftsprozessmodellierung verwenden lassen, indem Zuständigkeitszuordnungen von Aktivitäten zu Rollen spezifiziert werden können [UML10]. Ein Nachteil ist hier jedoch, dass Aktivitäten nur exklusiv einer Rolle zugeordnet werden können. Bei eEPKs war es noch möglich, mehrere Rollen einer Aktivität zuzuordnen.

Es wird jedoch nicht nur die Modellierungselemente im Vergleich zu EPKs erweitert. Bei einigen Modellierungselementen wurde eine unterschiedliche Semantik hinterlegt. Der Entscheidungsknoten, genannt *DecisionNode*, wird nun automatisch auf Basis von Objektdaten bzw. -zuständen ausgewertet. Die Kriterien zur Auswahl des korrekten Pfades werden in *Guards* an den Kanten direkt hinter dem *DecisionNode* annotiert. Eine Aktivität muss damit nicht mehr davor modelliert werden, die aktiv die Entscheidung trifft. Dieser Fakt ist am Beispiel von Abbildung 2.3(a) ersichtlich.

Zudem ist modellinhärent nicht mehr erkenntlich, wann das Prozessmodell zu instanziierten ist, da die Startereignisse nicht mehr vorhanden sind. Um in Aktivitätsdiagrammen Ereignisse zu empfangen, muss der Prozess bereits instanziiert sein.

UML spezifisch ist des Weiteren, dass i.d.R. mehrere der insgesamt 13 Diagramme verwendet werden, um verschiedene Aspekte eines Systems zu verdeutlichen. Die Modelle hängen dabei zusammen. Am Beispiel von Abbildung 2.3 hängen Aktivitäts-, Klassen- und Zustandsdiagramm unmittelbar zusammen. Im Aktivitätsdiagramm wird dabei auf Objekte vom Klassendiagramm (s. Abbildung 2.3(b)) und zusätzlich auf deren Zustände, die im Zustandsdiagramm modelliert sind (s. Abbildung 2.3(c)), verwiesen.

2.5.2.3 Business Process Model and Notation

Bei der BPMN wurden im Vergleich zu EPKs und Aktivitätsdiagrammen weitere Modellierungselemente hinzugefügt. Die Ausdrucksmächtigkeit ist zwar gegenüber EPKs deutlich gestiegen, aber die Sprache ist damit auch deutlich komplizierter geworden. Ein BPMN-Beispielmodell ist in Abbildung 2.4 angegeben.

Das Swimlane-Konzept, das bei den Aktivitätsdiagrammen eingeführt wurde, wurde bei BPMN mit einer Poolmodellierung erweitert. In Verbindung mit *Message Flows* ist nun unternehmensübergreifende Kommunikation modellierbar. Die *Message Flows* müssen nicht zwangsläufig mit Aktivitäten bzw. Events innerhalb der Pools verbunden werden. Sie können auch exklusiv an Pools geschickt werden. Es bleibt dann unterspezifiziert, wie der externe Prozess innerhalb der fremden Organisationseinheit aussieht. Beispielsweise bleibt in Abbildung 2.4 der Prozess in der *Rating agency* unterspezifiziert. Dort wird hingegen der Kreditantragsprozess in einer Bank näher betrachtet, der ähnlich zu dem EPK-Prozess von Abbildung 2.1 modelliert wurde.

Vergleicht man weiterhin die Kreditantragsmodellierung von BPMN (Abbildung 2.4) mit der EPK (Abbildung 2.1) sieht man, dass die Entscheidungsmodellierung in BPMN unterschiedlich ist. Bei BPMN wird nicht mehr vorgeschrieben, dass eine Funktion vor einem *Exclusive Gateway* stehen muss. Die Entscheidung ist datenbasiert und analog zu den *DecisionNodes* der Aktivitätsdiagramme zu sehen. Diese Vorschrift hat zur Konsequenz, dass die Funktionen in BPMN anders bezeichnet werden können. In der Literatur ist aber weiterhin sowohl bei Aktivitätsdiagrammen als auch bei BPMN die von EPKs bekannte Entscheidungsmodellierung zu finden, in der die Bezeichnung der Aktivität vor einem *Choice* so gewählt ist, dass dort

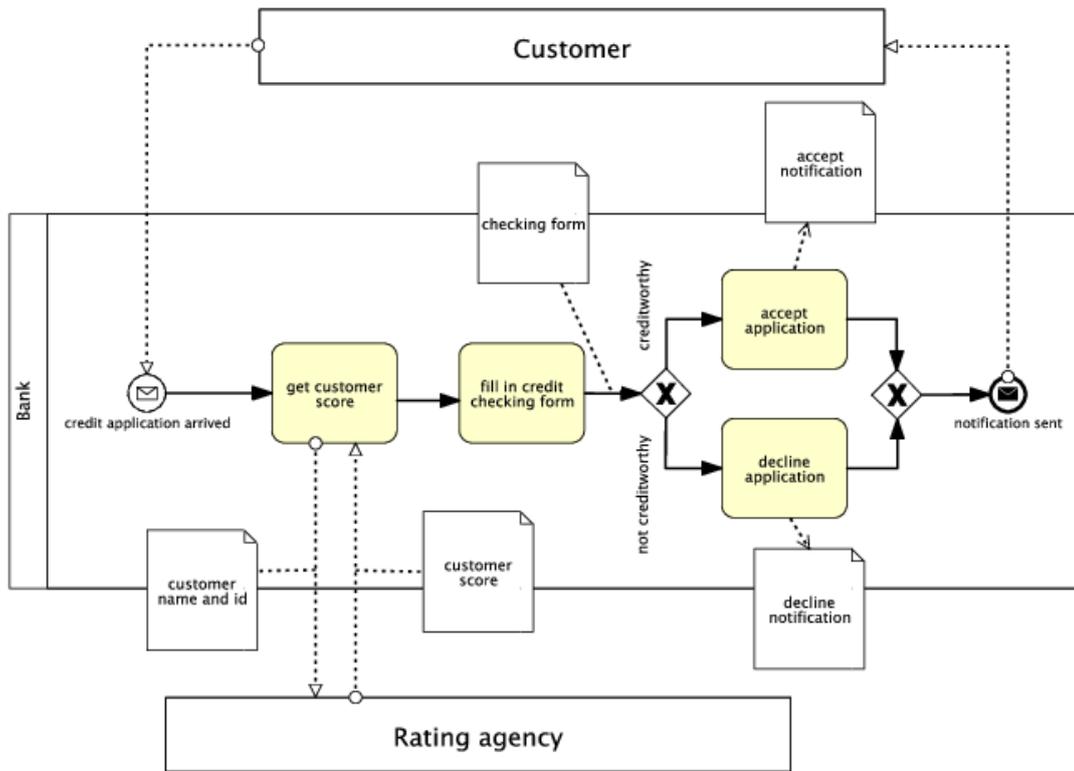


Abbildung 2.4: Ein Kreditantragsprüfungsprozess als BPMN Geschäftsprozessmodell

die Entscheidung getroffen wird. Beim Modell von Abbildung 2.4 ist dagegen eine andere Modellierung verfolgt. Dort wird ausgesagt, dass in der Aktivität ein Formular vom Angestellten auszufüllen ist. Daraufhin entscheidet nicht mehr der Angestellte, sondern das System anhand des ausgefüllten Formulars, ob der Kunde kreditwürdig ist oder nicht.

Die Objekt- bzw. Datenintegration ist bei der BPMN nicht so restriktiv wie bei Aktivitätsdiagrammen in Verbindung mit Objektflüssen. Bei Aktivitätsdiagrammen sind die Flüsse getypt und beim Zusammenführen bzw. Verzweigen von Kanten können Inkonsistenzen auftreten. Problematische Modelle, in denen z.B. unterschiedlich getypte Objektflüsse über Flussoperatoren (z.B. einem *Join*) zusammengeführt werden, werden in [BF08a] diskutiert. Bei BPMN bilden Assoziationen die Verbindung zu Datenobjekten und sind lediglich als Kommentare zu verstehen. Konsistenzbedingungen werden von der BPMN-Spezifikation nicht vorgegeben. Beispielsweise könnten zwei Flüsse mit unterschiedlichen Datenobjekten zusammengeführt werden. Obwohl Modellierungstools auf Grundlage der Spezifikation keine Unterstützung geben können, sollten die Modellierer selbständig auf Konsistenz der Datenflüsse achten.

Bei BPMN wurde die Instanziierungssemantik, die über die Startereignisse ausgedrückt wird, von EPKs übernommen. Endereignisse werden analog dafür verwendet, um auszudrücken, wenn der Prozess beendet ist. Zudem können dazwischen *catching-* und *throwing intermediate events* innerhalb der Prozessmodells genutzt werden.

An den vielen bei BPMN eingeführten neuen Modellierungselementen und an der datenbasierten Entscheidungsmodellierung sieht man, dass die Modelle deutlich technischer werden als es noch bei EPKs der Fall

war. Jedoch bleiben BPMN-Modelle semiformal und sind weiterhin wie EPKs und Aktivitätsdiagramme nicht ausführbar. Vielen Sprachelementen fehlt eine operationale Semantik. Insbesondere die Integration des Datenmodells und der Or-Join wirft weiterhin Probleme auf. Auch bleibt unklar, wie der datenbasierte *Choice* zu spezifizieren und auszuführen ist. Wie mit Message Flows, die z.B. an zugeklappte Pools gesendet werden (s. Abbildung 2.4), zu verfahren ist bleibt ebenfalls unklar. In der Spezifikation wird jedoch ein Abbildungsmechanismus zur Sprache BPEL eingeführt. Insbesondere die Verknüpfung der Message Flows mit Webservice-Aufrufen macht durchaus Sinn, um die unternehmensübergreifende verteilte Kommunikation zu realisieren.

2.5.2.4 Business Process Execution Language

BPEL ist eine Sprache, die im Zusammenhang mit Webservices eingesetzt wird. Sie steuert die Aufrufe der Webservices und stellt selber eine Webservice-Schnittstelle bereit. Diese Technik ist Grundlage für die z.Zt. modernen Service Orientierten Architekturen (SOA). Mit BPEL lassen sich Geschäftsprozesse implementieren bzw orchestrieren und ausführen. Die Schnittstellen bzw. Formulare zum Nutzer werden dabei über Webservices implementiert. Die Ablauflogik der einzelnen Aktivitäten (Webservice-Aufrufe) wird von der BPEL-Engine anhand der entsprechenden BPEL-Modelle festgelegt.

BPEL-Modelle sind XML-basiert und haben erstmal keine grafische Repräsentation im Standard [BPE07]. Es gibt aber durchaus Modellierungswerkzeuge, die einen grafischen Editor dafür bereitstellen. So z.B. kann die Netbeans-Umgebung BPEL-Prozesse grafisch modellieren. Mit dem Glassfish-Server können diese Modelle auf dem Server bereitgestellt werden. Zudem wird sogar eine Möglichkeit gegeben, diese Prozesse in Netbeans zu debuggen [Net06].

Ähnlich eines modellgetriebenen Ansatzes in der Softwareentwicklung gibt es auch Ansätze grafbasierte BPMN-Modelle in blockbasierte BPEL-Modelle zu transformieren [ODHA06]. Hierbei ist jedoch zu beachten, dass BPMN-Modelle unstrukturiert sein können, wohingegen BPEL durch die baumartige Hierarchisierung der XML-Dokumente strukturiert sind. Bei der Transformation ergeben sich also Probleme und nicht jeder BPMN-Prozess lässt sich mit BPEL ausdrücken.

Ein Organisationsmodell ist bei BPEL nicht vorhanden. Eine entsprechende Allokation und Zuweisung von Personen zu Aufgaben ist nicht möglich. Ein Datenmodell ist insoweit vorhanden, dass die Webservice-Schnittstellen definiert sein müssen. Diese Schnittstellen werden XML-basiert mit WSDL beschrieben.

2.5.2.5 Yet Another Workflow Language

Mit YAWL wurde sowohl eine Sprache als auch ein Workflow Management System entwickelt [HAAR09]. Das System wurde an der Universität Eindhoven und Queensland entwickelt. Im Gegensatz zu BPEL hat das WfMS direkt eine Umgebung, die dem Endnutzer erlaubt, die Aktivitäten auszuführen. Formulare zur Dateneingabe werden dem Nutzer während der Workflowausführung über das web-basierte WfMS präsentiert. Entsprechend der eingegebenen Daten wird der Prozess anhand des Workflowmodells dann weiter ausgeführt.

Das Datenmodell liegt nur implizit vor. Während der Modellierung zur *Designtime* sind über XML Dateneingaben zu spezifiziert. Eingabedaten werden mit Variablen verknüpft, die dann wiederum für Datenabfragen zu nutzen sind. Die im Vergleich zu grafischen Datenmodellen unintuitive XML-Spezifikationen ermöglichen eine Ausführung der Modelle. Wie bei BPMN wird bei YAWL ein datenbasierter *Choice* bereitgestellt, der durch die YAWL-Engine automatisch ausgewertet wird. Über XQuery werden die Datenabfragen zur Pfadauswahl gestellt und einer boolesche Formel zugeordnet. Diese werden nacheinander ausgewertet und

der entsprechende Pfad wird automatisch genommen. Daten, die für den Nutzer zur Aufgabenausführung benötigt werden, werden ebenfalls über XQuery abgefragt. Das Datenmodell liegt somit über textuelle XML-Daten vor. Eine intuitive Datenmodellierung mit UML ist hier nicht vorgesehen.

Ressourcenallokationen kann das WfMS automatisch vornehmen und Aktivitäten-ausführenden Personen zuweisen. Das Organisationsmodell liegt bei YAWL ebenfalls nicht grafisch vor. Rollen und Personenzuordnungen lassen sich über das web-basierte Administrationsinterface festlegen. Anhand dieser Zuordnungen und der vorher festgelegten Zuordnungsstrategie kann die YAWL-Engine dann die Arbeit automatisch an die Nutzer verteilen.

Alle Workflow-Patterns lassen sich in YAWL ausdrücken und letztendlich mit dem WfMS ausführen. Die Ausführungssemantik ist stark angelehnt an Petri-Netze. Aktivitäten lassen sich vom Nutzer starten und wieder beenden. Für jede Aktivität wird ein Petri-Netz als Subnetz angelegt und dieser Sachverhalt durch *start-* und *finish* Transitionen ausgedrückt [AH05, Abb.7]. Von dieser Modellierung und Verhalten zur Runtime wird in den YAWL-Workflowmodellen zur Designzeit jedoch abstrahiert.

Wie schon in Abschnitt 2.5.1 erwähnt, bieten WfMS große Flexibilitätsvorteile gegenüber konventioneller Software, die Geschäftsprozesse hart codieren. Soll ein Geschäftsprozess verändert werden, muss hier nicht die Software angefasst werden, sondern es reicht, das Geschäftsprozessmodell anzupassen. Dieses kann visuell mit dem YAWL-Editor geschehen. Daraufhin wird das Modell mit einem internen XML-Format auf die YAWL-Engine übertragen und der Prozess kann für weitere Geschäftsvorfälle instanziiert werden.

Ein Nachteil gegenüber konventioneller Software ist die Usability der generischen Nutzerschnittstelle im WfMS. Die Schnittstelle passt sich nicht den aufgabenspezifischen Anforderungen an, sondern ist generisch über die Web-Schnittstelle festgelegt.

2.5.2.6 ADEPT

Zusammen mit dieser Sprache ist ebenso wie bei YAWL ein Workflow Management System implementiert worden [DRRM11]. Es wurde an der Universität Ulm entwickelt und steht für „Application DEvelopment Based on Pre-Modeled Process Templates“. Es bietet zusätzliche Flexibilität für Geschäftsprozesse im Vergleich zu anderen WfMSen. Modellinstanzen lassen sich hier nach bestimmten Regeln zur Laufzeit verändern. Wenn sich z.B. während eines Geschäftsvorfalles herausstellt, dass Aktivitäten im Prozessmodell fehlen oder für diesen speziellen Fall anders angeordnet werden sollen, so bietet das System eine Möglichkeit, das Prozessmodell zur Runtime anzupassen. Autorisierungsmechanismen sind hier außerdem zu beachten. ADEPT ermöglicht Spezifikationen inkl. *Exception handlings* und *Adhoc-Änderungen* bei Workflows.

Außerdem kann das Prozessschema verändert werden während Instanzen von ihm laufen und die laufenden Instanzen werden darauf umgestellt. Dabei können Migrationsprobleme auftreten, die vom System beachtet werden. Beispielsweise könnte keine Aktivität im Modellschema vor eine schon beendeten Aktivität in einer laufenden Prozessinstanz eingefügt werden. ADEPT behandelt mit dieser Funktionalität die *Workflow Schema Evolution* und *Änderungspropagierung* [DR09].

Die hier betrachtete Flexibilitätseigenschaft wird als „flexible by change“ bezeichnet [SMR⁺08]. Der Nutzer des WfMS wird bei diesem System durch den Geschäftsvorfall durch das System geführt und bei Ausnahmefällen kann das Prozessmodell geändert werden. Zudem gibt es noch andere Flexibilitätskategorien wie „flexible by design“. Dieses Prinzip wird von deklarativen Prozessspezifikationen verfolgt, die Aktivitätsfolgen lediglich verbieten, die nicht erwünscht sind. Alle anderen Aktivitätsfolgen sind erlaubt. Dem Nutzer werden daher im voraus mehrerer Ausführungsmöglichkeiten gegeben als bei imperativen Prozessspezifikationen, die nur erlaubte Aktivitätsreihenfolgen spezifizieren.

Vor- und Nachteile der einzelnen Flexibilitätskategorien lassen sich diskutieren. „Flexible by change“ hat z.B. den Vorteil gegenüber „flexible by design“, dass die Geschäftsprozesse vorhersagbarer ablaufen und dadurch z.B. eine bessere Ressourcenplanung möglich ist. Es werden nur Änderungen im Prozessmodell vorgenommen, wenn entsprechende Ausnahmen auftreten. Außerdem braucht der Nutzer nicht ständig zu entscheiden, welche Aktivität er als nächstes zu tun hat. Dieses hat in der Regel eine Zeitersparnis zur Folge. Die ADEPT Prozessmodelle sind blockorientiert und daher strukturiert. Die Prozessmodelle sind damit weniger fehleranfällig und gut lesbar [LM10]. Die Modellierung der Workflows mit dem Editor wird auch als „correct by construction“ [DRRM11] bezeichnet. Das Modellierungswerkzeug erlaubt nur gültige Prozessmodelle und weist ungültige, die evtl. Deadlocks beinhalten zurück.

2.5.2.7 DECLARE

Mit DECLARE wird eine „flexible by design“ Strategie angewendet [APS09]. Der Fokus der Modelle liegt nun nicht mehr darin, Nutzer durch den Prozess zu führen und Ausführungsreihenfolgen von Aktivitäten festzulegen. Die Strategie liegt hier darin, dem Nutzer alle Freiheiten der Workflowausführung zu überlassen und nur explizit nicht erlaubte Aktivitätssequenzen zu verbieten. Der Ansatz basiert auf LTL-Formeln zur Spezifikation, welche Aktivitätssequenzen nicht erlaubt sind [Peš08].

Da eine textuelle Constraint basierte Workflowmodellierung sehr unintuitiv und nicht praktikabel ist, wurde für diesen Ansatz eine grafische Repräsentation entwickelt. Temporale Relationen lassen sich im DECLARE-Designer – dem Workflow Editor – definieren. Damit lassen sich Prozessmodelle grafisch modellieren, die dann letztendlich jedoch deklarative Prozessmodelle darstellen. Daraufhin lassen sich diese Modelle in eine Ausführungsumgebung laden und ausführen [Peš08].

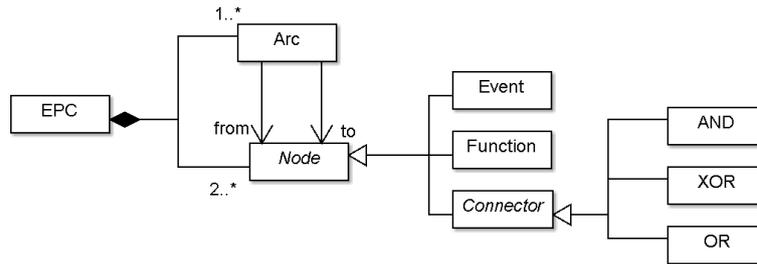
In der DECLARE-Ausführungs-Engine werden Rollen und ausführende Personen verwaltet und Zuordnungen während der Ausführung vorgenommen. Der Administrator hat die Aufgabe, die Rollen und Personen anzulegen und die Zuordnungen vorzunehmen. Hierarchische, grafische Organisationsmodelle, die z.B. von ARIS aus Abbildung 2.2(b) bekannt sind, können hier nicht verwendet werden. Ebenso wie das Organisationsmodell wird auch das Datenmodell nur sehr minimalistisch behandelt. Es können In- und Outputdaten für einzelne Aktivitäten definiert werden. Hier sind jedoch nur primitive Datentypen möglich. Zusammengesetzte Datentypen, Vererbung, Assoziationen, Aggregationen und Kompositionen sind nicht vorgesehen.

Jedoch ist eine Schnittstelle zum WfMS YAWL implementiert, so dass Kontrollflussspezifikationen für lose strukturierte Prozesse von DECLARE und stärker strukturierte Prozesse wieder von YAWL übernommen werden können. Zusätzlich ist auch eine Schnittstelle zum ProM-Tool (s. [ADG⁺07]) vorgesehen, die es ermöglicht, unter DECLARE abgearbeitete Prozesse zu analysieren.

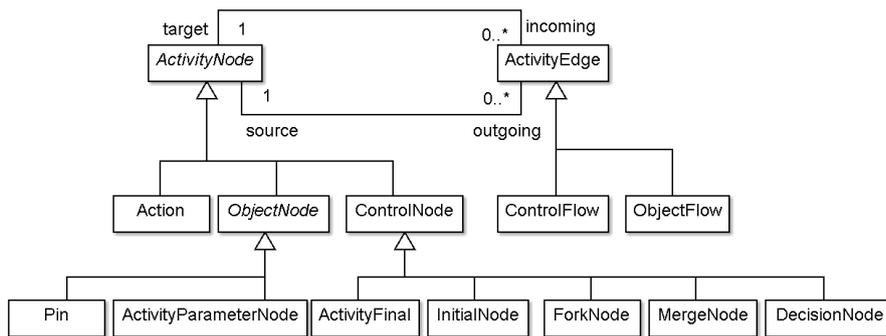
2.5.3 Metamodelle für Geschäftsprozessmodellierungssprachen

In diesem Abschnitt werden einige Metamodelle für verschiedene Modellierungssprachen in Abbildung 2.5 vorgestellt. Auffällig ist, dass es viele Ähnlichkeiten, aber auch sprachspezifische Unterschiede bei den Metamodellen existieren.

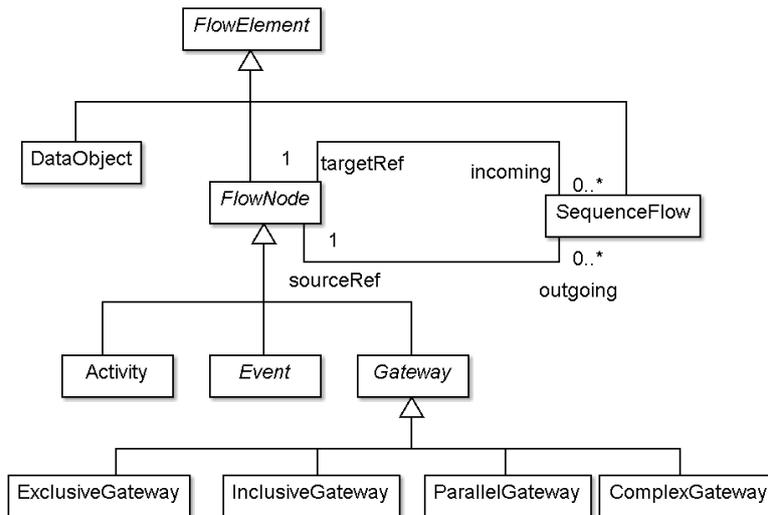
Mit UML hat sich die Technik des Metamodellierens etabliert. Die Metamodelle werden häufig durch umgangssprachliche Zusicherungen oder formale OCL-Invarianten, die an Klassen des Metamodells assoziiert werden, deutlich verfeinert. Abbildung 2.5(b) zeigt einen Ausschnitt des UML-Metamodells für Aktivitätsdiagramme [UML10, S.306ff]. Im Abschnitt der UML-Spezifikation zu den Aktivitätsdiagrammen [UML10, Kapitel 12] sind für die Zuweisungen der Constraints zu den Klassen zumeist jedoch um-



(a) Metamodel für EPKs nach [GLKK08]



(b) Ausschnitt des Metamodells für Aktivitätsdiagramme basierend auf [UML10]



(c) Ausschnitt des Metamodells für BPMN basierend auf [BPM11]

Abbildung 2.5: Ausschnitte der Metamodelle für EPKs, Aktivitätsdiagramme und BPMN

gangssprachliche Beschreibungen verwendet worden. Beispielsweise wird gesagt, dass wenn ein Objektfluss in ein *DecisionNode* fließt, ebenfalls Objektflüsse diesen verlassen müssen. Das gleiche gilt für Kontrollflüsse [UML10, S.370]. Bei *JoinNodes* können dagegen Objekt- und Kontrollflüsse gemixt werden. Dann muss jedoch ein Objektfluss den *JoinNode* auch wieder verlassen [UML10, S.394]. Diese Modellierungsvariante wurde auch in Abbildung 2.3 verwendet. Diese umgangssprachliche Zusicherung wird jedoch nicht von allen Aktivitätsdiagramm Modellierungstools berücksichtigt [BF08a]. Das mag auch ein Grund dafür sein, dass in der Literatur viele Modelle existieren, die diese Zusicherungen aus dem Metamodel verletzten und damit

keine korrekten Aktivitätsdiagramme widerspiegeln. Hier könnte die Formulierung der Bedingung mit OCL helfen. Werden diese Bedingungen bei der Entwicklung des UML-Modellierungstools, das die Implementierung des Metamodells bedeutet, berücksichtigt, würden diese Modellierungsfehler auffallen. Ein solcher formalerer Weg, OCL zu verwenden, wird bei dem metamodellbasierten Workflowmodellierungsansatzes vorgeschlagen, der in dieser Arbeit vorgestellt wird.

Für EPKs wurde zur Entwicklung des *bflow*-Modellierungswerkzeugs ein Metamodell entwickelt [GLKK08]. In Abbildung 2.5(a) ist zu sehen, dass ausschließlich die EPK-Modellierungselemente *Ereignis*, *Funktion*, *Konntektor* und *Kanten* definiert sind. Kanten verbinden die restlichen Modellierungselemente mit den beiden Assoziationen, die zur Klasse *Node* führen. Kanten sind im Metamodell wiederum durch die Klasse *Edge* ausgedrückt. Wichtige Regeln für EPKs werden über die OCL ähnliche Constraintsprache *Check* spezifiziert. Diese geben EPK-Regeln an, die in Abschnitt 2.5.2.1 schon eingeführt wurden. Z.B. müssen EPKs mit Ereignissen beginnen und enden. Außerdem ist zu prüfen, dass vor einem Entscheidungskonnektor eine Funktion und kein Ereignis steht.

Für eine weitergehende Integration der anderen ARIS-Sichten in der EPK kann man die eEPK benutzen, so wie in Abbildung 2.2(d) verdeutlicht. Eine abstrakte eEPK wird daraufhin in [Sch02, S.46] angegeben, die die Zusammenhänge der einzelnen Modelle visualisiert und eine Instanz des Metamodells darstellt. Die Steuerungssicht im ARIS-Ansatz wird anhand eines Metamodells in [Sch02, S.45] vorgestellt, das als eine Art Metamodell für die eEPK angesehen werden kann. Für eine Auslagerung der eEPK Modellierungselemente können auch Funktionszuordnungsdiagramme verwendet werden, in denen Funktionen z.B. mit Ein- und Ausgabedaten verknüpft werden. In [Sch01, S.105,118] werden dafür die Verbindungen der Funktionssicht zu anderen Sichten auf Metamodellebene angegeben.

Ein Metamodell ist auch in der BPMN-Spezifikation hinterlegt. In Abbildung 2.5(c) ist ein Ausschnitt aus diesem zu sehen, in der die Klasse *FlowElement* genauer betrachtet wird [BPM11, S.87]. In der BPMN Spezifikation wurde auf sowohl umgangssprachliche als auch OCL-basierte Constraintzuweisungen verzichtet. Die Metamodelle für BPMN bieten daher mehr Interpretationsspielraum als die Metamodelle der UML.

Vergleicht man die Metamodelle von Aktivitätsdiagrammen (s. Abbildung 2.5(b) und BPMN (s. Abbildung 2.5(c)) ist zunächst auffällig, dass die Diagramme sich sehr ähneln. Die Integration der Kanten im Metamodell und die Verknüpfung mit den restlichen Modellelementen durch die zwei Assoziationen ist gleich gelöst worden. BPMN hat hier offenbar den Teil für sein Metamodell übernommen. Jedoch ist die Datenspezifikation mit der Metaklasse *DataObject* bei BPMN ausgelagert worden.

Für ARIS wurde ein Metamodell in [Sch01] angegeben. Dort werden die Modellierungselemente der einzelnen Diagramme definiert. Bei der Funktionssicht (s. Abbildung 2.2(a)) wird eine hierarchische Zielmodellierung und hierarchische Funktionsmodellierung verfolgt. Es werden dafür zu den jeweiligen Klassen im Metamodell reflexive Assoziationen modelliert (s. Abbildung 2.6). Zudem wird die Verbindung zwischen Funktion und Ziel über eine Assoziation zwischen beiden Klassen angegeben [Sch01, S.38]. Außerdem hat Scheer die Anordnung und Ablaufreihenfolge bereits in der Funktionssicht über eine reflexive Assoziation *Anordnung* im Metamodell ausgedrückt.

Zu den in Abbildung 2.6 genutzten Assoziationsklassen ist zu sagen, dass sie zur Bezeichnung der Assoziationen genutzt wurden. Eine direkte Benennung der Assoziation wäre in UML-Klassendiagrammen stattdessen ebenfalls möglich gewesen.

Ähnlichkeiten weist das Metamodell zur ARIS Funktionssicht von Abbildung 2.6 mit dem des Metamodells für Aufgabenmodelle, das später in dieser Arbeit behandelt wird, auf. Beide Metamodelle beinhalten eine Hierarchisierung, Zielmodellierung und Angabe der Ablaufreihenfolgen von Funktionen bzw. Aufgaben.

Kapitel 3

Deklarative UML metamodellbasierte Workflowmodellierung

In diesem Kapitel wird der eigene, deklarative UML metamodellbasierte Ansatz zur Workflowmodellierung (*DMWM*) vorgestellt. Zunächst gibt Abschnitt 3.1 eine Einführung, in der die Grundlagen und verwendeten Sprachen zur Metamodellierung vorgestellt werden.

In Abschnitt 3.2 wird das Metamodell beschrieben und die damit definierte Workflowsprache erklärt. Zum Metamodell gehören zusätzlich zum UML-Klassendiagramm auch UML-Zustandsautomaten und OCL-Zusicherungen zur Definition der operationalen Semantik der Sprachelemente. Daraufhin wird die Mächtigkeit der neu definierten Workflowmodellierungssprache durch eine Workflow Pattern-Analyse [AHKB03] nachgewiesen. Es wird ein *DMWM*-Beispielprozess eingeführt. Schließlich werden noch Soundness-Eigenschaften anhand von OCL-Invarianten im Metamodell definiert.

Abschnitt 3.3 zeigt die Datenmodellierung und deren Integration in die Workflowmodelle. Hier werden die Möglichkeiten zur Spezifikation von Datenintegritätsbedingungen im Zusammenhang mit der Workflowmodellausführung aufgezeigt.

Die Modellierung der Organisation anhand eines Organisationsmetamodells ist Thema in Abschnitt 3.4. Der Zusammenhang zum Workflowmodell wird hergestellt und eine Möglichkeit zur Ressourcenallokation für die Runtime angegeben.

In Abschnitt 3.5 wird gezeigt, wie das Metamodell im Zusammenhang mit dem UML-Werkzeug USE eingesetzt wird, um Workflowmodelle zu erstellen. Es wird angegeben, wie das Tool den Nutzer auf Inkonsistenzen der Modelle aufmerksam macht und ihn bei der Fehlersuche unterstützt.

3.1 Einführung

In diesem Abschnitt werden die Sprachmittel zur Definition des *DMWM*-Ansatzes näher vorgestellt. In Abschnitt 3.1.1 wird die UML mit Fokus auf die metamodellbasierte Verwendung im *DMWM*-Ansatz eingeführt. Es werden die dort verwendeten UML-Diagrammarten näher beschrieben. Daraufhin wird in Abschnitt 3.1.2 die Spezifikation von Zusicherungen mit der Constraint-Sprache OCL eingeführt.

3.1.1 Ausgewählte UML-Diagramme zur Verwendung für DMWM

Die Unified Modeling Language (UML) stellt eine Sammlung von verschiedenen Diagrammart bereit, um vorwiegend Software zu entwerfen und zu dokumentieren. Sie stellt eine Sammlung von vorher entwickelten objektorientierten Softwarespezifikationssprachen dar, die von Grady Booch, Ivar Jacobson und James Rumbaugh entwickelt wurden. Die Modelle sind visuell und sollen intuitiv erfassbar sein, um Diskussionen zwischen Entwicklern untereinander oder mit dem Endanwender anzuregen.

Die unterschiedlichen UML-Diagramme werden grob in die folgenden zwei Gruppen unterteilt: Struktur- und Verhaltensmodelle. Zu den Strukturmodellen zählen die folgenden sechs. Die Modelle, die für den *DMWM*-Ansatz von Bedeutung sind, werden näher erläutert und angegeben, wofür sie in *DMWM* genutzt werden. Die restlichen UML-Diagramme werden daraufhin lediglich aufgelistet.

- **Klassendiagramm:** Dieses ist das zentrale Modell in der UML. Es wird zur visuellen Softwaremodellierung für die Beschreibung der Struktur bzw. Architektur genutzt. Die Sprachelemente sind Klassen und Assoziationen. Klassen beschreiben eine Menge von gleichartigen Objekten. Diese besitzen Attribute gleichen Typs und gleiche Operationen. Assoziationen beschreiben Beziehungen zwischen Objekten.

Es können damit objektorientierte Analysemodelle erstellt und diskutiert werden. Diese modellieren Ausschnitte der Realität, die sich daraufhin in der zu erstellenden Software widerspiegeln sollen. Die Analysemodelle lassen sich dann für (plattformspezifische) Softwaredesign- und Architekturmodelle transformieren [Kle08], die einen Hierfür werden Transformationstechniken angewendet, die eine modellgetriebene Softwareentwicklung realisieren [Kle08].

Klassendiagramme können des Weiteren zur Datenmodellierung und Metamodellierung genutzt werden. Im *DMWM*-Ansatz wird das Klassendiagramm für diese beiden Zwecke genutzt. Als Beispiel zur Datenmodellierung sind in Abbildung 2.2(c) und 2.3(b) bereits zwei Datenmodelle angegeben. Des Weiteren wurden auch schon Klassendiagramme zur Metamodellierung in Abbildung 2.5 und 2.6 präsentiert.

- **Objektdiagramm:** Das Modell beinhaltet eine Menge von Objekten, die Instanzen von Klassen darstellen. Zudem gibt es Links zwischen Objekten, die Instanzen von Assoziationen aus dem Klassendiagramm sind. Es lassen sich also Objektdiagramme aus Klassendiagrammen erstellen. Somit haben diese beiden Diagrammart eine unmittelbare Beziehung zueinander. Ein Objektdiagramm stellt einen Schnappschuss dar, der ein Systemzustand zu einem bestimmten Zeitpunkt im Programmablauf repräsentiert.

Im *DMWM*-Ansatz wird diese Diagrammart zur Modellierung der Workflowmodelle genutzt. Objektdiagramme stellen dabei eine abstrakte Syntax der Workflowsprache dar. Im Eclipse Modeling Framework (EMF) wird mit dem Graphical Modeling Framework (GMF) sogar eine Möglichkeit gegeben, den Instanzen der Metamodelle eine konkrete Syntax zu geben, indem bestimmten Typen aus dem Metamodel grafische Symbole zugeordnet werden [Gro09].

Bei *DMWM* werden zudem Objektdiagramme für Workflowinstanzen genutzt, die im Ablauf befindliche Workflowmodelle repräsentieren. Diese Modelle werden dem Nutzer über ein Workflow-Plugin für ein UML-Werkzeug, zur Interaktion mit dem Modellierer aufbereitet. Die Modelle lassen sich über die Sprache ASSL [GBR05], die eine OCL-basierte UML-Action Language repräsentiert, ausführen. Schnappschüsse des Datenmodells und das Organisationsmodell werden ebenfalls im Objektdiagramm repräsentiert, so dass sich die Abhängigkeiten der einzelnen Modelle mit dem hier vorgestellten Ansatz sehr gut testen lassen.

- **Paketdiagramm**
- **Verteilungsdiagramm**
- **Komponentendiagramm**
- **Kompositionsstrukturdiagramm**

Verhaltensmodelle gibt es sieben in der UML, die im Folgenden aufgelistet sind. Wiederum werden die für den *DMWM*-Ansatz verwendeten Diagramme näher erklärt. Zusätzlich werden Aktivitätsdiagramme und Anwendungsfalldiagramme erläutert. Diese werden zwar in *DMWM* nicht angewendet, sie sind aber die UML-Diagramme zur Modellierung von Aktivitäten bzw. Geschäftsprozessen. Daher haben sie Relevanz für diese Dissertation und werden ebenfalls kurz erklärt.

- **Zustandsdiagramm:** Die UML-Zustandsdiagramme stammen von Harel Statecharts [Har84] ab. Sie beinhalten zusätzlich Sprachmittel zu den Grundelementen *Zustände* und *Transitionen*. Es können dort *hierarchische Zustände*, *Gedächtniszustände* und *parallele Zustände* verwendet werden. Außerdem können noch *Guards* zur Spezifikation von Bedingungen an Transitionen annotiert werden.

Es gibt zwei Arten, UML-Zustandsdiagramme einzusetzen [UML10, S.541]. Zum einen lassen sich damit *Behavioral State Machines* repräsentieren. Diese spezifizieren das (selbständige) Verhalten von Objekten. Zum anderen kann mit *Protocol State Machines* die (externe) Nutzung von Objekten spezifiziert werden. Hierbei werden die erlaubten Aufrufreihenfolgen der Operationen von Objekten einer bestimmten Klasse angeben. Genau eine solche Spezifikation wird im *DMWM*-Ansatz verfolgt. Es werden die erlaubten Reihenfolgen der Operationsaufrufe von Aktivitäten angegeben, die der Nutzer verwenden darf. In der Workflow-Software werden Aktivitätsinstanzen (sog. *Workitems* [Hol98, HAAR09]) in einer Liste (sog. *Worklist* [Hol98, HAAR09]) dargestellt. Der Nutzer kann mit diesen interagieren, indem er entsprechende Operationen auf den *Workitems* aufruft. Ähnlich zur Nutzung eines Workflow Management Systems ist so auch die Schnittstelle für das *DMWM*-Plugin gestaltet.

- **Sequenzdiagramm:** Sequenzdiagramme werden genutzt, um Szenarien von Operationsaufrufen zu beschreiben. In der Regel wird genau eine Ablaufreihenfolge abgebildet. UML2 ermöglicht es zusätzlich, u.a. alternative, parallele, optionale und explizit nicht erlaubte Aufrufreihenfolgen anzugeben. Für alternative bzw. parallele Sequenzflüsse bieten sich aber eher Aktivitätsdiagramme als Modellierungsmittel an.

Bei *DMWM* können Operationen vom Nutzer auf Aktivitätsobjekten aufgerufen werden. Je nach temporaler Beziehung im Workflowmodell können Seiteneffekte auf andere Aktivitätsobjekte auftreten, indem entsprechende Operationen von den Aktivitätsobjekten aufgerufen werden. Diese Seiteneffekte werden sehr gut mit Sequenzdiagrammen protokolliert. Damit lassen sich die Workflowabläufe von abgearbeiteten Prozessinstanzen visualisieren und Analysen können darauf aufgebaut werden.

- **Aktivitätsdiagramm:** Diese Diagramme werden genutzt, um Aktivitätsabfolgen zu spezifizieren. Somit sind sie ebenso wie EPKs und BPMN zur Modellierung von Workflows geeignet und wurden in diesem Zusammenhang schon in Unterabschnitt 2.5.2.2 vorgestellt. Im Unterschied zu *DMWM* wird mit entsprechenden Symbolen ein expliziter Anfang und Ende bei imperativen Modellierungssprachen (wie Aktivitätsdiagrammen) spezifiziert.

Die Aktivitätsdiagramme werden im weiteren Verlauf der Dissertation in Abschnitt 5.2 noch zur Repräsentation bzw. Transformation von Aufgabenmodellen genutzt. Die Zusammenhänge der Sprachen und die Strukturiertheit der Aufgabenmodelle werden damit gezeigt.

- **Anwendungsfalldiagramm:** Diese Modelle werden häufig als erste Modellierungsart in Softwareprojekten zum Requirements Engineering genutzt. Dort wird dargestellt, welche Akteure wie mit dem zu entwickelnden System über Anwendungsfälle (Use Cases) interagieren. Im Diagramm können Anwendungsfälle mit *include* und *extend* in Beziehung gesetzt werden. *Include* besagt, dass der inkludierte Use Case für eine erfolgreiche Ausführung des übergeordneten Anwendungsfalles zwingend erforderlich ist. Dagegen wird *extend* verwendet, um Use Cases für Ausnahmefälle zu spezifizieren. Zur Behandlung des Ausnahmefalles wird dann der über die *extend* Beziehung definierte Use Case ausgeführt, um das Ziel des zu erweiternden Use Cases wieder zu erreichen. Über *extension points* können die Bedingungen angegeben werden, die zutreffen müssen, damit der Ausnahmefall eintritt.

Es gibt des Weiteren auch textuelle Repräsentationen, die eine strukturierte, formularbasierte Beschreibung von Anwendungsfällen darstellt [For07]. Dort sind zunächst der Hauptakteur und die Nebenakteure zu benennen. Dann ist das Ziel anzugeben, das der Hauptakteur mit dem von ihm ausgelösten Anwendungsfall verbindet. Außerdem ist ein Szenario als Interaktionsfolge zwischen Akteur und System zu beschreiben, das das vorher angegebene Ziel erfüllt. Dieses angegebene Szenario wird als Hauptszenario bezeichnet. Typischer Weise können mehrere Szenarien zum Erfolg eines Anwendungsfalles führen. Zudem sind die Startereignisse zu benennen, die den beschriebenen Use Case auslösen. Hier gibt es eine starke Ähnlichkeit zu Geschäftsprozessmodellen. Wie in den Unterabschnitten 2.5.2.1 und 2.5.2.3 erwähnt, ist die Spezifikation der Startereignisse ebenfalls Bestandteil von EPK- und BPMN-Geschäftsprozessmodellen.

Use Cases können des Weiteren synonym zu Prozessen bzw. Aktivitäten gesehen werden, so dass damit eine grobe Geschäftsprozessmodellierung, die noch keine Angaben über Reihenfolgen der Aktivitäten macht, ermöglicht wird. Die *include* und *extend*-Beziehungen können in Use Case-Diagrammen eine Hierarchie bilden, ähnlich zu Funktionsbäumen (s. [OWS⁺03, 81]), die z.B. Bestandteil der Geschäftsprozessmodellierung in ARIS sind [Sch01]. Die Hierarchisierung ist dagegen nicht Gegenstand der Betrachtung bei DMWM und kann in Modellen dieser Sprache nicht ausgedrückt werden.

- **Kommunikationsdiagramm**
- **Interaktionsübersichtsdiagramm**
- **Zeitverlaufdiagramm**

3.1.2 Spezifikation von Zusicherungen mit OCL

Da visuelle Modelle häufig nicht ausreichen, um eine Softwaresysteme genau genug zu beschreiben, wurde der UML eine textuelle Spezifikationssprache hinzugefügt. Diese Sprache ist seiteneffektfrei und wird als Object Constraint Language (OCL) bezeichnet [OCL10]. Die Spezifikationsmittel sind Invarianten und Vor- und Nachbedingungen. Diese sind integraler Bestandteil des *DMWM*-Ansatzes zur Spezifikation des Metamodells.

Invarianten werden Klassen aus dem UML-Klassendiagramm zugeordnet und stellen Zusicherungen dar, die die Objekte dieser Klassen nicht verletzen dürfen. Zudem gibt es OCL Vor- und Nachbedingungen, die Zustandsänderungen bei Operationsaufrufen spezifizieren können. Dabei wird durch die Vorbedingung angegeben, wie der Zustand vor der Operationsausführung sein muss. Durch die Nachbedingung wird der Zustand nach Ausführung der Operation spezifiziert. Eine imperative Angabe, wie die Zustandsänderung erreicht wird, ist mit OCL nicht möglich. Dafür können UML-Action Languages wie z.B. ALF [OMG10] oder SOIL [Büt11] genutzt werden.

Im folgenden werden Beispiele für OCL-Invarianten und Vor- und Nachbedingungen gegeben. Zunächst ist hierfür ein Klassendiagramm notwendig, welches in Abbildung 3.1 zu sehen ist.

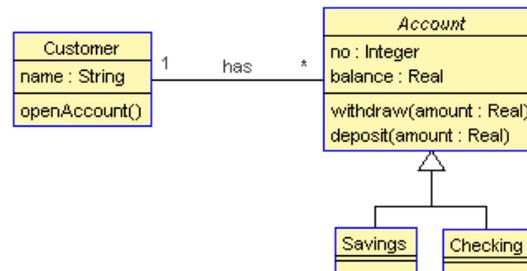


Abbildung 3.1: Bankenbeispiel als UML-Klassendiagramm zur OCL-Einführung

Es sind die Klassen *Customer* und *Account* spezifiziert. Die Klasse *Account* ist abstrakt. Kontenobjekte werden durch die spezielleren Klassen *Savings* und *Checking* in Spar- und Girokonten eingeteilt. Mit der Assoziation *has* und ihrer Multiplizitäten wird angegeben, dass ein Kunde mehrere Konten haben kann. Die zugeordnete Menge an Konten kann dabei sowohl Giro- als auch Sparkonten umfassen. Mit der Operation *openAccount()* wird ein neues Konto erstellt und dem entsprechenden Kunden zugeordnet. Es bleibt hier unterspezifiziert, welche Kontoart ausgewählt wird. Dieses kann z.B. vorkonfiguriert sein oder über eine UI-Abfrage ermittelt werden. Außerdem kann über die Operationen *withdraw* und *deposit* Geld von Konten abgebucht und wieder raufgebucht werden.

Ein Beispiel für eine Invariante wird in Listing 3.1 gegeben. Die Invariante *PositiveBalance* ist der Klasse *Savings* zugeordnet und sagt aus, dass ein Sparkonto keinen negativen Saldo haben darf. Zudem sind in Listing 3.1 die Operationen *withdraw()* und *openAccount()* mit Vor- und Nachbedingungen spezifiziert. Die Vorbedingung *withdrawAmountPositive* sagt aus, dass nur positive Beträge vom Konto abgebucht werden dürfen. Die Nachbedingung *withdrawDone* sichert zu, dass nur der ausbezahlte Betrag vom Kontosaldo abgezogen wurde. Eine letzte Bedingung ist mit *openAccount()* angegeben. Diese sagt aus, dass nach Ausführung der Operation *openAccount()* ein zusätzliches Konto dem Kunden zugeordnet wurde.

```

1 context Savings inv PositiveBalance:
2   balance >= 0
3
4 context Account::withdraw(amount:Real)
5   pre withdrawAmountPositive: amount > 0
6   post withdrawDone: self.balance = self.balance@pre - amount
7
8 context Customer::openAccount()
9   post accountAdded: account->size() = account@pre->size() + 1
  
```

Listing 3.1: OCL-Invariante und Vor- und Nachbedingungen für das Bankbeispiel

OCL kann auch für weitere Zwecke eingesetzt werden. Es sind auch OCL-Terme denkbar, die losgelöst von UML-Klassendiagrammen existieren können. Beispielsweise berechnet der OCL-Ausdruck in Listing 3.2 alle Quadratzahlen zwischen 1 und 100.

```

1 Set{ 1..100 } ->select(m | Set{ 1..100 } ->exists(n | m = n*n ))
  
```

Listing 3.2: OCL Term zur Berechnung aller Quadratzahlen zwischen 1 und 100

Außerdem lassen sich OCL-Terme auch für präzise Bedingungsspezifikationen an Transitionen in UML-Zustandsdiagrammen oder Guards an Sequenzflüssen in Aktivitätsdiagrammen verwenden. Es gibt noch diverse weitere Anwendungsmöglichkeiten für OCL. In diesem und folgendem Kapitel werden zudem noch weitere im Zusammenhang mit der Workflowmodellierung aufgeführt.

3.2 Workflow-Metamodell

In diesem Abschnitt wird das Workflow-Metamodell als UML-Klassendiagramm mit UML-Zustandsautomaten vorgestellt. OCL spielt eine wichtige Rolle, um die operationale Semantik deklarativ, plattformunabhängig auszudrücken. Diese Teile bilden den Kern des DMWM-Ansatzes.

3.2.1 UML-Klassendiagramm

Die Diagramme für das Workflow-Metamodell umfassen ein UML-Klassendiagramm und zwei UML-Zustandsdiagramme. Das Klassendiagramm ist in Abbildung 3.2 zu sehen. Zur Definition der operationalen Semantik werden UML-Zustandsautomaten benötigt. Sie werden in Abbildung 3.3 angegeben und definieren die Lebenszyklen der Aktivitätsobjekte.

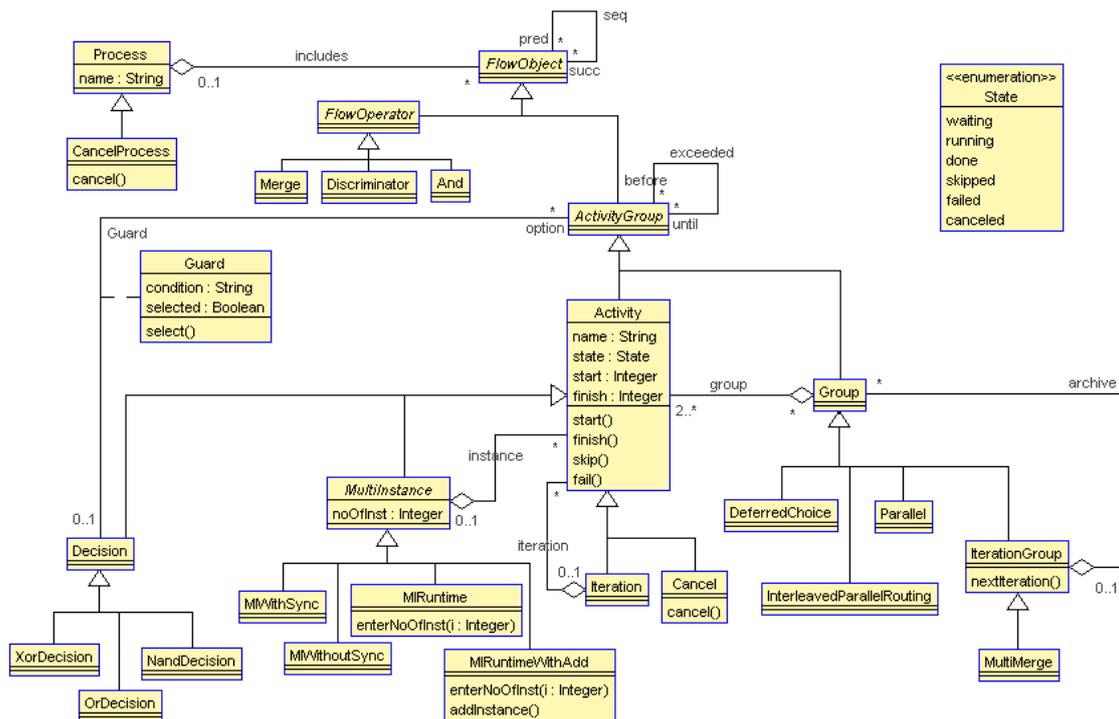


Abbildung 3.2: Das Metamodell für den DMWM-Ansatz

Die allgemeinste Klasse im Metamodell von Abbildung 3.2 ist *FlowObject*, an der die reflexive Assoziation *seq* modelliert ist. Anhand dieser Assoziation werden die Aktivitäten (*Activity*) bzw. Gruppen von Aktivitäten (*Group*) und Flussoperatoren (*FlowOperator*) in *succ* und *pred* Objekte eingeteilt. Die Abarbeitungsreihenfolge der Aktivitäten wird anhand dessen angegeben. Die Bezeichnung *FlowObject* wurde vom Metamodell

der BPMN 1.2 Spezifikation übernommen [BPM09]. Damit wurden alle Elemente im BPMN-Prozessmodell bezeichnet, die mit Kanten verbunden werden. In der neueren BPMN Spezifikation [BPM11] wurde ein Metamodell entwickelt, in dem *FlowObjects* nun als *FlowElements* bezeichnet werden (s. Abbildung 2.5(c)).

Die Klasse *Process* beschreibt Prozessobjekte. Diese werden über das Attribut *name* bezeichnet. Zudem können spezielle Prozesse anhand der Klasse *CancelProcess* im Workflowmodell verwendet werden, die während der Ausführung vom Nutzer abgebrochen werden können. Näheres dazu wird in Unterabschnitt 3.2.4.5 erläutert. Über die Assoziation *includes* werden Prozessobjekte mit *FlowObjects* verbunden und damit als Elemente vom Prozess aufgenommen. Aus praktischen Gründen brauchen nicht alle zugehörigen Modellierungselemente mit dem Prozessobjekt verlinkt werden. Es gilt hier der Grundsatz: Wenn ein Modellierungselement zu einem Prozess gehört, dann auch die, die indirekt über beliebige Assoziation aus dem Metamodell erreichbar sind. Die Berechnung der transitiven Hülle, um alle enthaltenen Modellelemente einzusammeln ist Teil des Metamodells und Thema in Unterabschnitt 3.2.3.2. Somit spart man viele Verbindungen der Assoziation *includes* in den Prozessmodellen. Diese werden dadurch deutlich übersichtlicher und lesbarer.

Die abstrakte Klasse *FlowObject* hat wiederum die abstrakte Klasse *FlowOperator* als Unterklasse. Deren Unterklassen stellen Flussoperatoren dar. Diese werden bei EPKs als Konnektoren, bei Aktivitätsdiagrammen als *ControlNodes* und BPMN als *Gateways* bezeichnet. Die Klasse *AndOperator* repräsentiert einen And-Konnektor in EPKs bzw. einen *split-* bzw. *join-*Knoten bei UML-Aktivitätsdiagrammen. Führen mehrere *seq-*Links in den Knoten, ist dieser Knoten als *join* anzusehen. Die Kontrollflüsse werden an diesem Knoten synchronisiert. Es wird also auf die Abarbeitung aller verbundener Aktivitäten gewartet, bis die nachfolgenden Aktivitäten ausgeführt werden können. Hat der Knoten dagegen mehrere *seq-*Links, die herausführen ist ein *split* vorhanden. Die damit verbundenen Aktivitäten können dann also unabhängig voneinander ausgeführt werden. Analog zu UML-Aktivitätsdiagrammen können auch beim DMWM-Ansatz hybride *AndOperator*-Knoten existieren, in die mehrere *seq-*Links hineingehen und auch herauskommen. Diese Knoten sind dann sowohl als *join* als auch als *split*-Knoten anzusehen.

Des Weiteren gibt es bei den Flussoperatoren die Klasse *MergeOperator*, die dazu dient, Flüsse zusammenzuführen, ohne sie zu synchronisieren. Als letztes gibt es noch die Klasse *Discriminator*. Diese Klasse wird für ein spezielles Workflow Pattern benötigt. Aktivitätsabfolgen, die mit *seq-*Links verknüpft werden, werden nicht synchronisiert. Bereits nach Erledigung der ersten Vorgängeraktivität, können die Nachfolgeaktivitäten abgearbeitet werden. Alle weiteren Vorgänger-Aktivitäten können dann ohne weitere Konsequenzen beendet werden. In Unterabschnitt 3.2.4.2 erfolgt eine näher Erklärung zum Workflow Pattern und zu deren Modellierung in DMWM.

Die abstrakte Klasse *ActivityGroup* umfasst die reflexive Assoziation *exceeded*. Hiermit können ebenso wie bei der *seq-*Assoziation Aktivitäten und Gruppen von Aktivitäten verbunden werden. Es können damit eine Art Timeout-Punkte im Prozessmodell definiert werden, die bei Überschreitung andere Aktivitäten überspringen und nicht weiter ausführen lassen. Die Semantik dieser Beziehung wird für die *cancellation and force completion* Workflow Patterns benötigt und in Unterabschnitt 3.2.4.5 näher beschrieben.

Die zentrale Klasse im Metamodell von Abbildung 3.2 ist *Activity*. Die Klasse hat ein Attribut *state*, das mit Werten der Enumeration *State* belegt wird. Zudem umfasst die Klasse die Operationen *start()*, *finish()*, *skip()* und *fail()*. Die spezielle Klasse *Cancel* umfasst des Weiteren die Operation *cancel()*. Die Operationen stellen die Schnittstelle für den Nutzer bereit, der anhand dieser mit den Aktivitäten zur Runtime interagiert und damit den Workflow steuern kann. Wie bereits in Unterabschnitt 1.2.1 beschrieben, liegt die Bestimmung der Ausführungsreihenfolge des Workflows beim Nutzer. Dieses ist analog zu [Peš08] ein eher deklarativer Ansatz, der weniger die Automatisierung der Workflows zum Ziel hat. Die Zustände und Methoden bei den Aktivitätsobjekten bilden die Grundlage zur Definition der Zustandsdiagramme von Abbildung 3.3.

Neben *Cancel* gibt es noch diverse weitere spezielle Aktivitäten, die mit Vererbungsbeziehungen zur Klasse *Activity* im Metamodell vorhanden sind. Zu ihnen gehört die Klasse *Iteration*, welche die Eigenschaft hat, dass sie mehrfach ausgeführt werden kann. Damit hat sie ein anderes Verhalten verglichen zu dem, das im Objektlebenszyklus in Abbildung 3.3(a) dargestellt ist. Ruft der Nutzer die Operation *start()* auf, leitet er einen neuen Iterationszyklus ein. Damit wird ein neues Aktivitätsobjekt erstellt und mit Hilfe der Assoziation *iteration* an das Iterationsobjekt gehängt. Der *start()*-Aufruf wird an das neu erstellte Aktivitätsobjekt weiterdelegiert. Das Verhalten ist in einem Sequenzdiagramm in Abbildung 4.6(b) veranschaulicht. Anhand dessen können zur Runtime die Anzahl und die Zeitpunkte der einzelnen Operationsaufrufe festgehalten werden. Die abstrakte Klasse *MultiInstance* enthält Aktivitäten, von denen mehrere Instanzen zur gleichen Zeit unabhängig abgearbeitet werden können. Näheres dazu und den zugehörigen Unterklassen wird in Unterabschnitt 3.2.4.3 erklärt.

Mit *Decision* ist eine weitere spezielle Aktivität im Metamodell vorhanden. Diese Klasse stellt Entscheidungsaktivitäten dar, in denen der Nutzer zur Runtime entscheidet, welcher Pfad zu nehmen ist. Während der Design-time werden die Kriterien zur Entscheidungsauswahl in den zugeordneten *Guards* festgelegt. Die mit diesen verbundenen Aktivitäten werden dann aktiviert, wenn der Nutzer den entsprechenden Pfad ausgewählt hat. Diese Modellierungsphilosophie wird auch in EPKs verfolgt, in denen vor Entscheidungskonnektoren eine Funktion zu modellieren ist, in der die Entscheidung aktiv zu treffen ist (s. Abschnitt 2.5.2.1). In BPMN wird dafür ein datenbasierter Choice verwendet (s. Abschnitt 2.5.2.3). Mit der Klasse *XorDecision* wird festgelegt, dass genau ein Pfad zur Runtime ausgewählt werden muss. Bei *OrDecision* muss mindestens ein Pfad, es können aber auch mehrere ausgewählt werden. Diese Auswahlmöglichkeiten haben sich bei den Sprachen EPK, BPMN und YAWL bewährt. Zusätzlich gibt es bei DMWM die Möglichkeit mit der Klasse *NandDecision*, keinen Prozesspfad auszuwählen. In diesem Falle würden alle über *Guards* mit der Entscheidungsaktivität verbundenen Prozesspfade geskippt werden. Das Verhalten der Entscheidungsaktivitäten und die Benutzung zur Runtime wird Thema in Abschnitt 4.2.4 sein.

Ein weiterer wichtiger Aspekt im Metamodell ist die Gruppierung von Aktivitäten. Dieser wird ausgedrückt durch die Klasse *Group* und durch die Aggregationsbeziehung *group*. Gruppen können spezielle temporale Beziehungen ausdrücken. Hierüber können diverse Workflow Patterns sehr einfach modelliert werden. Dazu gehört beispielsweise die *Deferred Choice* und *Interleaved Parallel Routing*-Beziehung, die in Abschnitt 3.2.4.4 näher erläutert werden.

Eine temporale Beziehung, die nicht bei den Workflow Patterns wiederzufinden ist, ist die *Parallel*-Beziehung. Bei Aktivitäten, die zu einer solchen Gruppe gehören, wird zugesichert, dass diese parallel auszuführen sind. Hier sind Assistenz- oder Observierungstätigkeiten denkbar, die zugesichert parallel geschehen müssen. Beispiele für die *Deferred Choice* und *Parallel*-Gruppen werden im Modellierungsbeispiel von Abschnitt 3.5.3 verwendet. Abschnitt 3.2.4 gibt vorher eine formale strukturierte Betrachtung der temporalen Beziehungen mit einer Workflow Pattern-Analyse.

Eine letzte wichtige Gruppe im Metamodell stellt die Klasse *IterationGroup* dar. Alle Aktivitäten, die dieser Gruppe zugeordnet sind, können zurückgesetzt werden, indem die Operation *nextIteration()* aufgerufen wird. Hier ist zu beachten, dass nur bei in der Gruppe komplett abgearbeiteten Iterationszyklen anhand dieser Operation ein neuer Iterationszyklus eingeleitet werden kann. Die durch OCL ausgedrückte formale Semantik dafür wird in Abschnitt 3.2.3.1 näher vorgestellt. Eine spezielle Klasse, die von *IterationGroup* abgeleitet wurde, ist *MultiMerge*. Diese realisiert das gleichnamige Workflow Pattern. Eine genauere Betrachtung dazu wird in Abschnitt 3.2.4.2 gegeben.

Das in Abbildung 3.2 vorgestellte Klassendiagramm repräsentiert ein einziges (integriertes) Metamodell sowohl für die Modellierung von Workflows (zur Design-time) als auch für die Beschreibung der Workflowinstanzen (zur Runtime). Im Metamodell ist die operationale Semantik der Modellelemente mit OCL

hinterlegt. Diese wird erst zur Ausführung der Modelle relevant, sie ist jedoch bei der Workflowmodellierung zur Designtime bereits vorhanden. Mit diesem Ansatz eröffnen wir uns die Möglichkeit, während der Runtime die Modelle auf Basis des integrierten Metamodells zu verändern. Dieser Adaptivitätsaspekt zur Workflowmodellierung wird in Abschnitt 4.3.4 beschrieben.

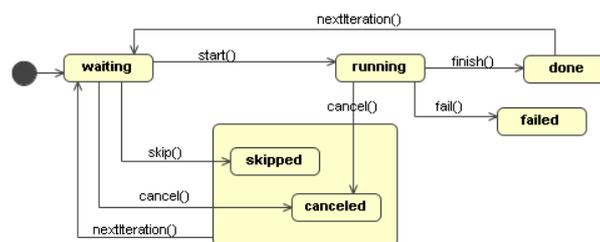
Die einzigen Daten, die für Workflowmodelle im Gegensatz zu Workflowinstanzen nicht benötigt werden sind die Attribute *start* und *finish* in der Klasse *Activity*. In diesen Attributen werden die Zeitstempel festgehalten, in denen die Zustandsänderungen der Aktivitätsobjekte stattgefunden haben. Aus Gründen der Einfachheit wurden diese Attribute aber im Metamodell (auch für die Designtime) belassen, so dass im DMWM-Metamodell diese Daten sowohl für Workflowmodelle als auch für Workflowinstanzen vorhanden sind. Die Beziehungen zwischen Workflowmodell und Workflowinstanz und die dafür verwendeten Diagrammartentypen werden in Abschnitt 4.2.3 näher erläutert.

3.2.2 Zustandsdiagramme

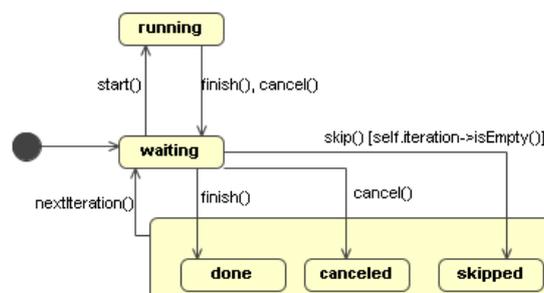
Die Aktivitätsobjekte und deren in Abbildung 3.3 dargestellten Objektlebenszyklen haben zunächst für die Designtime keine Konsequenzen. Sie sind jedoch für die Runtime existentiell notwendig, indem sie die Grundlage zur Definition der operativen Semantik der Workflowsprache DMWM legen.

In Abbildung 3.3 sind nun die Zustandsdiagramme der verschiedenen Aktivitätsklassen aus dem Metamodell von Abbildung 3.2 zu sehen. Es gibt zwei verschiedene Arten von Aktivitätsklassen im Workflowmetamodell, die für den Nutzer ein grundlegend unterschiedliches Verhalten haben. Das Verhalten der Aktivitäten vom Typ *Activity*, *Decision* (inklusive Unterklassen), *MultiInstance* (inklusive Unterklassen) und *Cancel* wird mit dem Zustandsautomaten von Abbildung 3.3(a) ausgedrückt. Ein anderes Verhalten haben Objekte der Klasse *Iteration*, das durch das Zustandsdiagramm von Abbildung 3.3(b) spezifiziert wird.

In den Zustandsdiagrammen ist ersichtlich, dass die Übergänge auf Operationsaufrufen basieren, die in



(a) Lebenszyklus eines Aktivitätsobjekts



(b) Lebenszyklus eines Iterationsaktivitätsobjekts

Abbildung 3.3: Lebenszyklen von Aktivitätsobjekten modelliert in UML-Zustandsautomaten

den Klassen vom Metamodell in Abbildung 3.2 bereitgestellt werden. Die Aktivitäten werden mit dem Zustand *waiting* initialisiert. Daraufhin hat der Nutzer die Möglichkeit, die Aktivität mit dem Aufruf *start()* zu starten oder mit *skip()* zu überspringen. Befindet sich das Aktivitätsobjekt im Zustand *running* kann es mit *finish()* regulär beendet werden. Mit dem Aufruf *fail()* gibt der Nutzer an, dass während der Ausführung ein Fehler passiert ist. Nur speziellen *Cancel* Aktivitäten ist es möglich, sie während der Ausführung abzubrechen. Ganze Prozesse können mit *cancel()* abgebrochen werden, wenn es sich um Prozesse vom Typ *CancelProcess* handelt.

Grundsätzlich sind Aktivitätsobjekte der Klassen *Activity*, *Cancel* und der Unterklassen von *Decision* nur einmal ausführbar. Die Operationen, die von der Klasse *Activity* bereitgestellt werden, erlauben kein Zurücksetzen des Zustandes, was ein wiederholtes Ausführen ermöglichen würde. Die dafür vorgesehene Operation *nextIteration()* ist der Klasse *IterationGroup* zugeordnet. Wenn sich die Aktivität im Zustand *done*, *skipped* oder *canceled* befindet und einer Iterationsgruppe zugeordnet ist, ist darüber eine wiederholte Ausführung möglich. Diese Operation *nextIteration()* ist jedoch weiteren Ausführungsbedingungen unterworfen, die in Abschnitt 3.2.3.1 näher erläutert werden. Es sind hier nur geordnete Iterationsdurchläufe möglich. D.h. es ist erst dann ein neuer Iterationsdurchlauf erlaubt, wenn bei allen in der Gruppe enthaltenen Aktivitätsobjekten die Transition in den Zustand *waiting* nach den State Charts von Abbildung 3.3 möglich ist.

Aktivitäten, die inhärent ohne explizites Zurücksetzen der Aktivität mehrfach ausgeführt werden können, sind *Iteration*-Aktivitäten. Deren Verhalten ist im Zustandsdiagramm von Abbildung 3.3(b) verdeutlicht. Mit den Operationsaufrufen *start()* und *finish()* können beliebig viele Iterationen durchgeführt werden. Ein Iterationszyklus ist dafür im Sequenzdiagramm von Abbildung 4.6(b) angegeben. Bei einem Iterationszyklus kann statt *finish()* auch *fail()* aufgerufen werden. Die Ausführungssemantik ist damit folgende: Mit dem *fail()*-Aufruf schlägt nicht die Iterationsaktivität an sich fehl, sondern der aktuell ausgeführte Zyklus. Somit kann z.B. wenn ein Zyklus fehlerhaft beendet wurde, ein weiterer ausgeführt werden, der dann die komplette Aktivität zum Erfolg führt.

Iterationen werden schließlich explizit über die Methode *finish()* beendet, wenn sich das Objekt im Zustand *waiting* befindet. Der Aufruf *cancel()* ist nur indirekt über die Klasse *CancelProcess* möglich. Mit *skip()* können Iterationen nur übersprungen werden, wenn noch kein Iterationszyklus durchlaufen wurde, was durch den zugeordneten OCL-Guard [*self.iteration->isEmpty()*] ausgedrückt wird. Falls schon ein Iterationszyklus initiiert wurde, ist ein Aktivitätsobjekt, das diesen repräsentiert, über die Assoziation *iteration* verbunden. Bereits beendete, übersprungene oder abgebrochene Iterationen können nach dem Zustandsdiagramm von Abbildung 3.3(b) genauso wie normale Aktivitäten über die Methode *nextIteration()* wieder zurückgesetzt werden, wenn sie sich in einer Iterationsgruppe befinden.

3.2.3 OCL-Zusicherungen

Im Gegensatz zu allen anderen bekannten Metamodellansätzen zur Workflowmodellierung (s. Abschnitt 2.5.3) wurde in DMWM umfangreich und zu verschiedenen Zwecken von OCL-Constraints Gebrauch gemacht. Insbesondere wird OCL zur Definition der operationalen Semantik der neu entwickelten Workflowsprache genutzt, um eine weitgehend plattformunabhängige, deklarative Spezifikation zu erreichen.

Zunächst wird zur Definition der operationalen Semantik in Abschnitt 3.2.3.1 beschrieben, wie die in Abbildung 3.3 gezeigten Zustandsautomaten anhand von OCL-Vor- und Nachbedingungen ausgedrückt werden. Daraufhin werden in Abschnitt 3.2.3.2 die Grundlagen und OCL-Hilfsfunktionen aus dem Metamodell erklärt, die benötigt werden, um die temporalen Beziehungen mit OCL zu spezifizieren. Hierzu gehört u.a. die Berechnung von transitiven Hüllen im Zusammenhang mit reflexiven Assoziationen aus dem UML-Klassendiagramm. Schließlich werden in Abschnitt 3.2.3.3 noch exemplarisch temporale Bezie-

hungen anhand von OCL definiert, die von Workflow Patterns nicht abgedeckt sind, aber Bestandteile des Metamodells und damit der neuen Sprache sind. Eine detaillierte Analyse der Elemente, die die Workflow Patterns abdecken wird in Abschnitt 3.2.4 folgen.

3.2.3.1 Angabe der Objektlebenszyklen mit OCL

Die Lebenszyklen von Aktivitätsobjekten sind mit Hilfe von UML-Zustandsautomaten in Abbildung 3.3 spezifiziert. Die Transitionen werden durch Operationsaufrufe repräsentiert. Diese Art der Spezifikation von Objektlebenszyklen ist in der UML Spezifikation [UML10, Kap. 15.1] mit *Protocol State Machines* bezeichnet.

Die State Charts können über OCL Vor- und Nachbedingungen ausgedrückt werden. Der Zustand des Objekts ist durch das Attribut *state* im Zusammenhang mit der Enumeration *State* im Klassendiagramm angegeben. In Listing 3.3 ist anhand der angegebenen OCL Vor- und Nachbedingung die Transition mit dem Operationsaufruf *start()* aus Abbildung 3.3(a) angegeben. Die Vorbedingung *isActivityWaiting* gibt an, dass der Zustand des Aktivitätsobjekts *waiting* sein muss. Die Nachbedingung *isActivityRunning* spezifiziert die Zustandsänderung auf Zustand *running*. Zusätzlich ist in Listing 3.3 noch die OCL-Spezifikation für die Operation *finish()* angegeben. Analog werden die weiteren Transitionen für die Operationsaufrufe *skip()*, *fail()* und *cancel()* von Abbildung 3.3(a) mit Hilfe von OCL gebildet. Diese werden hier jedoch nicht weiter angegebenen.

```

1 context Activity::start()
2   pre isActivityWaiting: state=#waiting
3   post isActivityRunning: state=#running
4
5 context Activity::finish()
6   pre isActivityRunning: state=#running
7   post isActivityDone: state=#done
8
9 context IterationGroup::nextIteration()
10  pre isDoneSkippedOrCanceled: self.activity->forAll(state=#done or state=#skipped or state=#canceled)
11  post isIGWaiting: self.activity->forAll(state=#waiting)

```

Listing 3.3: OCL-Vor- und Nachbedingungen für Transitionen im Zustandsdiagramm der Klasse *Activity*

In Listing 3.3 ist noch die Spezifikation für die Operation *nextIteration()* spezifiziert. Diese Operation ist wie im Abschnitt 3.2.1 erwähnt der Klasse *IterationGroup* zugeordnet. Entsprechend der Zustandsdiagramme von Abbildung 3.3 besagt die Vorbedingung, dass alle zur Gruppe gehörende Aktivitäten im Zustand *done*, *skipped* oder *canceled* sein müssen. Die Nachbedingung besagt, dass die Aktivitäten auf *waiting* zurückzusetzen sind.

Im Gegensatz zum Automaten von normalen Aktivitäten in Abbildung 3.3(a) haben Iterationsaktivitäten den Objektlebenszyklus von Abbildung 3.3(b). Vergleicht man die Automaten, können die Vor- und Nachbedingungen für die *start()*-Operation für die *Iteration* Klasse gleich bleiben. Die Vor- und Nachbedingungen werden vererbt, so dass für diese Operation keine neuen Bedingungen angegeben werden müssen. Für die Operation *finish()* sieht es da anders aus. Setzt man den Automaten in OCL-Vor- und Nachbedingungen um, ergibt sich eine Spezifikation von Listing 3.4.

Schließlich sei hier noch das *Design by contract* Prinzip [Mey92] im Zusammenhang mit dem Substitutionsprinzip [LW94] erwähnt. Dieses besagt, dass Objekte der Unterklassen Objekte der Oberklassen ersetzen können und dieses auch für Verhaltensspezifikation gelten soll. Objekte der Unterklassen sollen

```

1 context Iteration::finish()
2 pre isIterRunning: state=#waiting or state=#running
3 post isIterDone: state@pre=#waiting implies state=#done
4 post isIterWaiting: state@pre=#running implies state=#waiting

```

Listing 3.4: OCL-Vor- und Nachbedingungen für die Operation *finish()* der Klasse *Iteration*

also das Verhalten der Oberklasse haben und evtl. zusätzliche Funktionalität hinzufügen. Das hat zur Folge, dass Vorbedingungen von gleichen Operationen nur gleich bleiben oder abgeschwächt werden können. Nachbedingungen können nur gleich bleiben oder verschärft werden.

Bei der Programmiersprache Eiffel werden zur Umsetzung für Operationen, die in Unterklassen überschrieben werden folgende Klauseln verwendet. Vorbedingungen können mit *require else <precondition>* nur abgeschwächt werden. D.h. die Vorbedingungen der Oberklasse und die Vorbedingungen der Unterklasse werden mit einem *or* logisch verknüpft. Nachbedingungen können mit *ensure then <postcondition>* nur verschärft werden. Dort werden die Nachbedingungen der Oberklasse mit der Unterklasse mit dem logischen *and* verknüpft.

In der OCL-Spezifikation werden zwar *contracts* in Verbindung mit Vor- und Nachbedingungen erwähnt [OCL10], aber das Substitutionsprinzip, das eine Abschwächung der Vorbedingung und Verschärfung der Nachbedingung erfordert wird nicht vorgegeben. Das Verhalten ohne Substitutionsprinzip ist für das DMWM-Metamodell das gewünschte.

Sollte bei der Implementierung des DMWM-Ansatzes das *Design by contract* inkl. Substitutionsprinzip gelten, müssten die bisher vorgestellten Vor- und Nachbedingungen angepasst werden. Ein Problem würde ansonsten bei der Nachbedingung von *finish()* auftreten. Ein Objekt, das sich im Zustand *running* befindet, wird durch die Verknüpfung der Zeile 7 in Listing 3.3 und Zeile 4 in Listing 3.4 mit dem logischen *and* Operator die Nachbedingung *state=#done and state=#waiting* erhalten. Diese würde offensichtlich ein Widerspruch enthalten.

3.2.3.2 OCL-Hilfsfunktionen im Metamodell

In UML lassen sich Operationen im Klassendiagramm mit OCL-Termen verknüpfen, wovon u.a. in der UML-Spezifikation gebrauch gemacht wird [UML10]. Die Operationen haben nach dem Prinzip von OCL keine Seiteneffekte auf das Objektmodell.

Mit dieser Eigenschaft ist die Möglichkeit zur Berechnung der transitiven Hülle bei reflexiven Assoziationen gegeben. Ein Beispiel ist dafür in [KHGB] spezifiziert. Auch für DMWM werden verschiedene transitive Hüllen unter Verwendung der OCL berechnet.

Die Operation *getSuccObjects()* in Listing 3.5 ist u.a. für die Soundness-Prüfung der Prozessmodelle notwendig, um potenzielle Deadlocks schon während der Designtime zu finden. Dieses Thema wird in Abschnitt 3.2.6 näher behandelt. Die dort eingeführten OCL-Invarianten benötigen diese Operationen.

Die Operation *getFlowObjects()* in Listing 3.5 wird u.a. für das Einsammeln der Modellierungselemente im Prozessmodell benötigt. Wie in Abschnitt 3.2.1 beschrieben, müssen somit nicht alle Elemente im Prozessmodell mit dem entsprechenden Prozessobjekt über die Assoziation *includes* verbunden werden. Sie können indirekt über die Berechnung der transitiven Hülle dem Prozessobjekt zugeordnet werden.

Im DMWM-Metamodell, dem Klassendiagramm von Abbildung 3.2, sind die hier vorgestellten Funktionen aus Übersichtlichkeitsgründen weggelassen worden. Dort sind nur die Operationen aufgeführt, die eine Schnittstelle zum Nutzer darstellen.

```

1 FlowObject::getSuccObjects(act:Set(FlowObject)):Set(FlowObject)=
2   let newElems:Set(FlowObject)=self.succ - act in
3   if newElems->isEmpty() then act
4   else newElems->collect(getSuccObjects(act->union(newElems)))->asSet()->union(act)
5   endif
6
7 Activity::getFlowObjects(act:Set(FlowObject)):Set(FlowObject)=
8   let dec:Set(FlowObject)=if decision.isUndefined then
9     oclEmpty(Set(FlowObject)) else oclEmpty(Set(FlowObject))->union(Set{decision}) endif in
10  let newElems:Set(FlowObject) = (self.pred->union(self.succ)->union(self.before)->
11    union(self.until)->
12    union(group)->union(dec)) - act in
13  if newElems->isEmpty() then act else
14    newElems->collect(getFlowObjects(act->union(newElems)))->asSet()->union(act)
15  endif

```

Listing 3.5: OCL-Operationen *getSuccObjects()* und *getFlowObjects()* zur Berechnung der transitiven Hülle

Die Operationen bekommen eine Menge von *FlowObjects* als Argument übergeben und liefern ebenfalls eine Menge von *FlowObjects* zurück. Das Prinzip zum rekursiven Einsammeln der Elemente ist bei beiden Operationen dasselbe.

Die Operation *getSuccObjects()* sammelt alle *FlowObject*-Elemente die durch die Navigation über das Assoziationsende *succ* der Assoziation *seq* erreichbar sind ein. Neue Elemente werden in Zeile 2 von Listing 3.5 mit *self.succ* erreicht. Der rekursive Aufruf der Operation *getSuccObjects()* ist dann in Zeile 4 zu sehen.

Das Verhalten der Operation *getFlowObjects()* ist etwas umfangreicher, da zusätzlich noch die *pred*-Navigation bei der Assoziation *seq* und die weiteren Assoziationen *exceeded* und *Guard* zu berücksichtigen sind. Nicht berücksichtigt werden die Assoziationen *iteration* und *instance*. Diese Assoziationen sind für Prozessmodelle zur Designtime nicht relevant. Sie werden erst zur Runtime wichtig, um die einzelnen Instanzen der Durchläufe bei Iterationen bzw. bei den Multiinstanz-Aktivitäten festzuhalten. Analog verhält es sich mit der Assoziation *archive*, die auch erst zur Runtime eingesetzt wird.

3.2.3.3 Spezifikation temporaler Beziehungen mit OCL

Die operationale Semantik wird der entwickelten Workflowsprache anhand von Invarianten, die den Klassen aus Abbildung 3.2 zugeordnet werden und sich auf die Lebenszyklusspezifikationen von Abbildung 3.3 beziehen, gegeben. Diese Zusicherungen werden dann zur Runtime von USE überprüft, so dass sichergestellt wird, dass ein korrekter Ablauf der Prozessmodelle garantiert ist.

Die Invariante für die vermutlich wichtigste temporale Beziehung (der Sequenz bzw. Workflow Pattern 1) wird in Unterabschnitt 3.2.4.1 vorgestellt. Auch die Abdeckung aller weiteren Workflow Patterns wird dort analysiert und exemplarisch die OCL-Invarianten vorgestellt.

Modellierungselemente, die nicht über die Workflow Patterns abgedeckt werden, werden in diesem Abschnitt vorgestellt. Hiezu gehört z.B. die temporale Beziehung *Parallel*, die als Unterklasse von *Group* im Metamodell von Abbildung 3.2 vorhanden ist. Bei den Workflow Patterns gibt es u.a. das *Interleaved Parallel Routing* Pattern, welches besagt, dass bestimmte Aktivitäten nicht parallel ausgeführt werden dürfen. Das Pattern wird in Unterabschnitt 3.2.4.4 näher behandelt. Eine inverse temporale Beziehung, würde besagen, dass alle Aktivitäten synchron ausgeführt werden müssen. Diese Beziehung gibt es bei den Workflow Patterns nicht, wird aber bei DMWM mit der Gruppe *Parallel* ausgedrückt. Diese temporale Beziehung ist durchaus wichtig für Assistenz- oder Überwachungstätigkeiten. Die Semantik wird mit der Invariante *allInTheSameState* aus Listing 3.6 ausgedrückt. Hier wird sichergestellt, dass alle Aktivitätsobjekte, die zu

einer Gruppe *Parallel* gehören im gleichen Ausführungszustand sein müssen.

Die Entscheidungsmodellierung ist ein wichtiger Teil von Geschäftsprozess- bzw. Workflowmodellierungssprachen (s. Abschnitt 2.5.2). So spielt sie auch für DMWM eine wichtige Rolle. Es wird zwischen expliziten und impliziten Entscheidungen unterschieden. Die implizite Entscheidungsmodellierung wird mit dem *Deferred Choice* Pattern in Abschnitt 3.2.4.4 näher erläutert. Für die explizite Entscheidungsmodellierung sind spezielle Aktivitäten, die Unterklassen von *Decision* im Metamodell von Abbildung 3.2 sind, vorgesehen. Die Kriterien zur Auswahl des korrekten Prozesspfades werden über die *Guards* angegeben.

Die damit direkt oder über Gruppen indirekt verbundenen Aktivitäten können erst dann starten, wenn der Pfad ausgewählt wurde.

Die OCL-Invarianten drücken zunächst nur die deklarative operationale Semantik aus. Das Überspringen der zur Runtime nicht ausgewählten Prozesspfade erfordert eine imperative Spezifikation, welche dann in Abschnitt 4.2.2 näher vorgestellt wird.

```

1 context Parallel inv allInTheSameState:
2   self.activity->forAll(a | self.activity->any(true).state=a.state)
3
4 context Guard inv onlySelectedActivitiesOrGroupsCanRun:
5   if (self.option.oclIsTypeOf(Activity)) then
6     self.option.oclAsType(Activity).state=#running implies self.selected
7   else -- option is a group
8     self.option.oclAsType(Group).activity->exists(state=#running) implies self.selected
9   endif

```

Listing 3.6: Die OCL Invarianten zur Definition der *Parallel*-Beziehung und expliziten Entscheidungsmodellierung

3.2.4 Workflow Patterns

Um die Mächtigkeit der hier eingeführten Workflow-Modellierungssprache zu demonstrieren wird in diesem Abschnitt eine Workflow Pattern Analyse gemacht. Dieses Mittel ist weit verbreitet und allgemein akzeptiert, um Modellierungssprachen für Geschäftsprozesse bzw. Workflow Management Systeme zu analysieren. So wurden u.a. EPKs [MNN05], UML-Aktivitätsdiagramme [RAHW06], BPMN [WAD⁺06], BPEL [WADH03] und das π -Kalkül [PW05] betrachtet.

Die in diesem Abschnitt erfolgende Workflow Pattern Analyse ist nach Kenntnisstand des Autors die erste für einen deklarativen Workflow-Modelliensatz.

Im Folgenden werden nicht alle OCL-Invarianten zur Definition der operationalen Semantik vorgestellt. Nur exemplarisch werden einige erklärt. Es wird lediglich die grobe Semantik umgangssprachlich erläutert. Die OCL-Spezifikationen genauso wie die Beschreibungen geben keine Garantie auf Vollständigkeit. Anhand von Modellausführungen zur Runtime wurde das Metamodell jedoch ausgiebig getestet (s. Kapitel 4). Sollten bei weiteren Ausführungen Unvollständigkeiten im Metamodell auftauchen, können diese anhand von Erweiterungen leicht behoben werden.

3.2.4.1 Basic Control Flow Patterns

Zu den in diesem Abschnitt vorgestellten *Basic Control Flow Patterns* zählen die grundlegenden Modellierungselemente, die von allen Workflowmodellierungssprachen unterstützt werden. In Abbildung 3.4 sind die

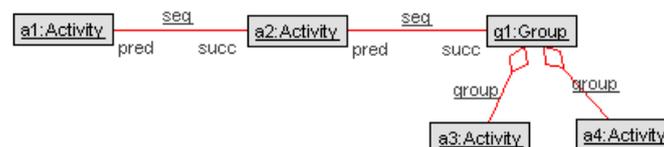
Workflow Control Flow Patterns (WCP) 1-5 modelliert.

In Abbildung 3.4(a) ist angegeben, wie die Sequenz (WCP1) in DMWM ausgedrückt wird. Aktivität *a1* und *a2* sind mit einem *seq*-Link verbunden. Die Rollenbezeichnung *pred* und *succ* geben die Reihenfolge an, in der die Aktivitäten abzuarbeiten sind. In den darauffolgenden Prozessmodellen u.a. in Abbildung 3.4(b) und 3.5 sind die Rollenbezeichnungen für die *seq*-Links weggelassen worden. Die Leserichtung *von links nach rechts* ist dort anzuwenden. D.h. die mit einem *seq*-Link verbundene linke Aktivität ist die vorher auszuführende *pred*-Aktivität und die rechte die darauf folgende *succ*-Aktivität. In Abbildung 3.4(a) ist als letztes in der Sequenzkette die Gruppe *g1* vorhanden. Die enthaltenen Aktivitäten *a3* und *a4* haben keine weiteren Ablaufbeschränkungen. Die erlaubten Ablaufsequenzen sind somit *a1,a2,a3,a4* oder *a1,a2,a4,a3*, wobei *a3* und *a4* auch parallel ausgeführt werden dürfen.

In Listing 3.7 ist dann die operationale Semantik anhand der OCL-Invariante *seqActivity* angegeben. Zunächst werden dort die Vorgängeraktivitäten in der Variable *activities* eingesammelt und die Vorgänger-Gruppen in *ops*. Daraufhin wird in der Invariante folgendes ausgesagt: Wenn das aktuell betrachtete Aktivitätsobjekt im Zustand *running* oder *done* ist, so müssen alle Vorgängeraktivitäten (auch die, die in Gruppen zugeordnet sind) entsprechend der Lebenszyklusspezifikation von Abbildung 3.3 im Zustand *done* oder *skipped* sein.

Die beiden nächsten Workflow Patterns sind mit WCP2&3 in Abbildung 3.4(b) angegeben. Sind Aktivitäten nicht verbunden, so bestehen keine Ausführungseinschränkungen für sie. Ein Beispiel dafür ist mit den Aktivitäten *a3* und *a4* in Abbildung 3.4(a) vorhanden. Die Unabhängigkeit von zwei oder mehreren Aktivitäten kann man aber auch explizit über den *AndOperator* spezifizieren. In Abbildung 3.4(b) sind die Aktivitäten *a2* und *a3* hinter dem *and1*-Operator angegeben. Aktivität *a1* liegt davor und muss damit vorher abgearbeitet werden. Der *and1*-Operator repräsentiert damit das Split-Pattern (WCP2).

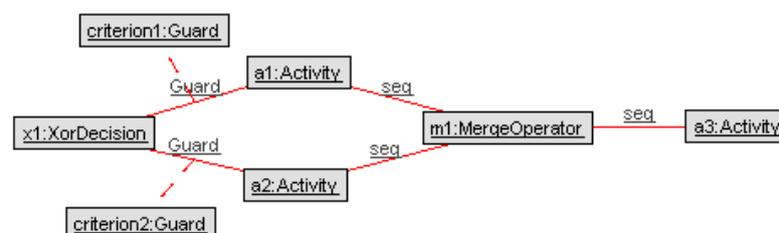
Hinter den Aktivitäten *a1* und *a2* folgt der *and2*-Operator, der den Kontrollfluss wieder zusammenführt und synchronisiert. Dahinter folgt noch Aktivität *a4*, die als letztes erfolgen muss. Entsprechend wird mit dem



(a) WCP1: Sequence



(b) WCP2 & 3: Parallel Split und Synchronization



(c) WCP4 & 5: Exclusive Choice und Simple Merge

Abbildung 3.4: Basic Control Flow Patterns in DMWM-Prozessmodellen

```

1 context Activity inv seqActivity:
2   let activities:Set(Activity) = pred.oclAsType(Activity)->select(ala.isDefined)->asSet in
3   let ops:Set(Group) = pred.oclAsType(Group)->select(o | o.isDefined)->asSet in
4   (self.state = #running or self.state = #done) implies
5     activities->union(ops.activity)->forall(a | a.state=#done or a.state=#skipped)
6
7 context XorDecision inv doneSelectedXor:
8   self.state=#done implies self.guard[decision]->select(glg.selected)->size = 1

```

Listing 3.7: Definition der Sequenz (WCP1) und Exclusive Choice (WCP4)

Objekt *and2* das *Synchronization* Pattern ausgedrückt.

Die deklarative OCL-Invariante zur Klasse *AndOperator* besagt, dass alle Aktivitätsobjekte bzw. Gruppen von Aktivitäten, die mit *pred* verbunden sind, abgearbeitet sein müssen, bevor die succ-Aktivitäten starten können. Die OCL-Invariante wird hier nicht explizit angegeben. Zu dieser Invariante ist zu sagen, dass *FlowOperator*-Objekte nicht über die *seq*-Beziehung verbunden sein dürfen. Diese Eigenschaft ist als Soundness-Eigenschaft in einer OCL-Invariante festgelegt, analog zu denen, die in Abschnitt 3.2.6 näher vorgestellt werden.

Die Workflow Patterns 4&5 werden in Abbildung 3.4(c) näher betrachtet. Für den *choice*-Operator (WCP4) wird eine *XorDecision*-Entscheidungsaktivität gewählt. Während der Ausführung dieser Aktivität muss der Nutzer genau einen Pfad auswählen, der danach auszuführen ist. Die Auswahl findet über die *select()*-Operation der zugehörigen Guard-Objekte (s. Abbildung 3.2) zur Runtime statt. Dass genau ein Pfad auszuwählen ist, wird in der OCL-Invariante *doneSelectedXor* in Listing 3.7 spezifiziert. Die nicht ausgewählten Aktivitäten werden geskippt. Dieses Verhalten betrifft jedoch die imperative Spezifikation der Prozessausführung, welche in Kapitel 4 näher betrachtet wird.

Der *merge*-Operator (WCP5) wird mit dem *MergeOperator*-Objekt in Abbildung 3.4(c) ausgedrückt. Die Prozesspfade werden dort zusammengeführt. Die zugeordnete OCL-Invariante besagt, dass eine Aktivität im Zustand *done* sein muss, damit die Folgeaktivitäten gestartet werden können.

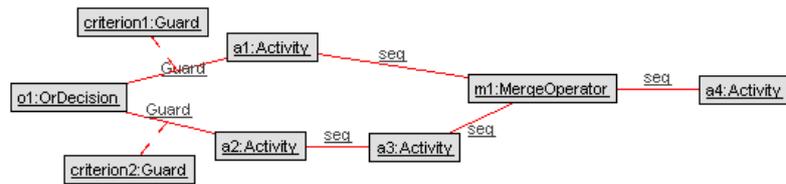
3.2.4.2 Advanced Branching and Synchronization Patterns

In diesem Abschnitt werden mit Abbildung 3.5 weitergehende temporale Beziehungen behandelt, die etwas kompliziertere Verzweigungs- und Synchronisierungsbeziehungen darstellen.

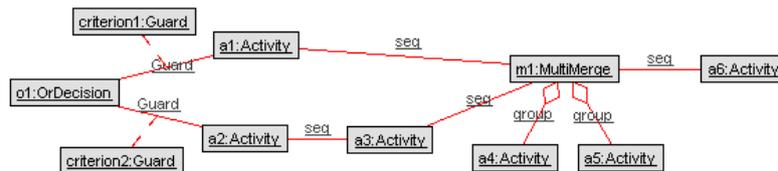
Ein prominentes Beispiel ist mit dem *Multichoice* und dem *Structured Synchronizing Merge* (WCP6&7) in Abbildung 3.5(a) ausgedrückt. Diese Patterns wurden bereits von EPKS mit *OR*-Konnektoren verwendet. Es kann vom Nutzer mit der *OrDecision*-Aktivität mehr als nur eine Folgeaktivitätskette ausgewählt werden. Die nicht ausgewählten Aktivitätsketten werden bis zum zugehörigen *MergeOperator* geskippt. WCP7 besagt mit *structured*-Bezeichnung, dass zu jeder Entscheidung ein zugehöriger *Merge* vorhanden sein muss. Analog zur Invariante *doneSelectedXor* aus Listing 3.7 besagt die Invariante für *OrDecision*, dass mindestens eine Aktivität nach Beendigung der Aktivität ausgewählt sein muss.

In Abbildung 3.5(b) ist dann das *MultiMerge*-Pattern (WCP8) modelliert. Dieses besagt, dass die Anzahl der vorher ausgeführten *pred*-Aktivitäten, die Iterationszyklen der in der Gruppe enthaltenen Aktivitäten bestimmt. Sind also Aktivität *a1* und *a3* ausgeführt worden, müssen genau zwei Iterationszyklen beim *Multimerge*-Objekt *m1* durchgeführt werden. Somit müssten die Aktivitäten *a4* und *a5* zweimal ausgeführt werden.

Das *Structured Discriminator* Pattern (WCP9) wird im DMWM-Metamodell mit der Klasse *Discriminator*



(a) WCP6-7: Multi-Choice und Structured Synchronizing Merge



(b) WCP8: Multi-Merge



(c) WCP9: Structured Discriminator

Abbildung 3.5: Advanced Branching and Synchronization Patterns

ausgedrückt und in Abbildung 3.5(c) verwendet. Die über OCL-Invarianten angegebene operationale Semantik sagt aus, dass mindestens eine *pred*-Aktivität vor dem Operator ausgeführt sein muss bevor die *succ*-Aktivitäten hinter dem *Discriminator* Objekt aktiviert werden. Alle weiteren *pred*-Aktivitäten, die dann beendet werden, haben keinen weiteren Effekt auf die Folgeaktivitäten des Prozessmodells.

Im Workflow Pattern-Katalog folgt nun *Arbitrary Cycle* (WCP10), der in Abschnitt 3.2.4.5 näher behandelt wird und *Implicit Termination* (WCP11). Die implizite Beendigung des Workflows wird von DMWM direkt unterstützt. Mit dem flexiblen, deklarativen Ansatz ist eine explizite Beendigung des Workflows nicht vorgesehen. Das Prozessmodell ist erst dann beendet, wenn alle Aktivitäten nicht mehr im Zustand *waiting* oder *running* befinden. Dieses repräsentiert eine implizite Beendigung des Prozessmodells und damit das WCP11.

3.2.4.3 Multiple Instance Patterns

In diesem Abschnitt werden Aktivitäten behandelt, die mehrfach parallel ausgeführt werden können. Im Metamodell von Abbildung 3.2 sind die vier Unterklassen von der abstrakten Klasse *MultiInstance* dafür vorgesehen, die vier Multiinstanz-Patterns (WCP12-15) umzusetzen.

Alle Multiinstanz-Aktivitäten verwenden die Assoziation *instance* vom DMWM-Metamodell von Abbildung 3.2. Sie verbindet die Multiinstanz-Aktivität mit deren einzelnen Ausführungsinstanzen, die Objekte vom Typ *Activity* sind. Sie besitzen den Lebenszyklus von Abbildung 3.3(a) mit der Ausnahme, dass der Zustand nicht mit *nextIteration()* für die Ausführungsinstanzen zurückgesetzt werden kann. Näheres zur Ausführung der Multiinstanz-Aktivitäten folgt in Abschnitt 4.3.2.3.

Während der Designtime müssen den Patterns WCP12&13 die Anzahl der auszuführenden Instanzen über das Attribut *noOfInst* mitgeteilt werden. Diese Zahl gibt an, wie viele Ausführungsinstanzen beim Starten der Aktivität zur Runtime erzeugt werden sollen. In Abbildung 3.6(a) wird die Klasse *MIWithSync* (WCP12)

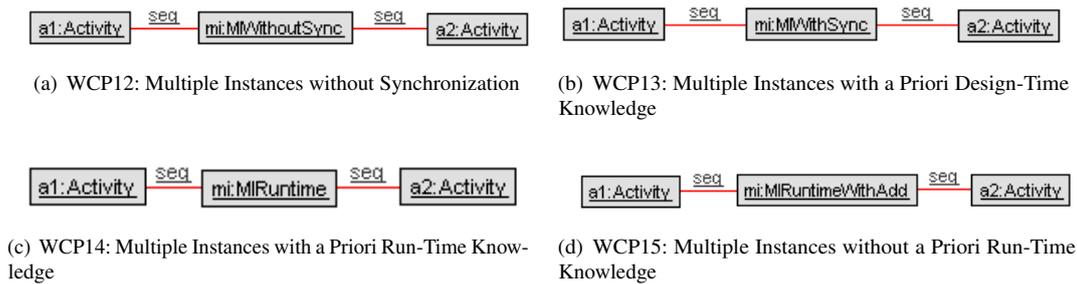


Abbildung 3.6: Multiple Instance Patterns

verwendet. Die operationale Semantik der Klasse ist wie folgt: Zur Runtime müssen alle über die Assoziation *instance* verbundenen Ausführungsinstanzen im Zustand *waiting* sein, damit die übergeordnete *MultiInstance*-Aktivität ebenfalls in den Zustand *waiting* versetzt werden kann. Damit kann die Folgeaktivität *a2* erst dann ausgeführt werden, wenn alle Instanzen von *mi* abgearbeitet, d.h. synchronisiert sind. Im Gegensatz zu WCP12 müssen die einzelnen Ausführungsinstanzen in WCP13 nicht synchronisiert werden. Somit kann in Abbildung 3.6(b) die Aktivität *mi* in den Ausführungszustand *done* versetzt werden, auch wenn noch einzelne Ausführungsinstanzen ausgeführt werden. Folglich darf die Folgeaktivität *a2* ohne Synchronisierung der Ausführungsinstanzen gestartet werden.

Die Klasse *MIRuntime* aus Abbildung 3.2 repräsentiert WCP14. Bei diesem Pattern wird erst zur Runtime beim Starten der Aktivität die Anzahl der zu erzeugenden Instanzen festgelegt. Hierfür wird die Operation *enterNoOfInst()* aus dem Metamodell genutzt. Die einzelnen Ausführungsinstanzen werden analog zum Verhalten von WCP12 synchronisiert. In Abbildung 3.6(c) ist somit die Aktivität *a2* erst dann zu starten, wenn alle Instanzen von *mi* beendet wurden.

Das letzte Multiinstanz-Pattern ist in Abbildung 3.6(d) mit der Klasse *MIRuntimeWithAdd* angewendet worden. Das Verhalten der Objekte dieser Klasse ist analog zum WCP14, wird jedoch um die Operation *addInstance()* erweitert. Dem Nutzer steht über diese Operation die Möglichkeit offen, neue Ausführungsinstanzen während der Ausführung hinzuzufügen. Erst nachdem die Multiinstanz-Aktivität beendet wurde, können auch keine weiteren Instanzen erzeugt werden.

Die oben beschriebene Ausführungssemantiken der verschiedenen *MultiInstance*-Patterns sind anhand der unterschiedlichen OCL-Invarianten im Metamodell spezifiziert. Diese Invarianten werden hier jedoch nicht weiter angegeben.

3.2.4.4 State-based Patterns

Abbildung 3.7 zeigt die drei Patterns, die zu den *State-based Patterns* gehören. Das *Deferred Choice*-Pattern kann sehr einfach modelliert werden (s. Abbildung 3.7(a)). Die OCL-Invariante *allWaitingXorOneRunningOthersSkipped* zur Definition der Semantik ist in Listing 3.8 spezifiziert. Sie besagt, dass zum einen alle Aktivitäten aus der *DeferredChoice*-Gruppe im Zustand *waiting* sein dürfen. Wenn sich zum anderen eine Aktivität im Zustand *running* befindet, müssen alle anderen im Zustand *skipped* überführt worden sein.

Das nächste Pattern ist *Interleaved Parallel Routing*, welches in Abbildung 3.7(b) modelliert wurde. Es ist ähnlich einfach zu modellieren wie der vorher betrachtete *Deferred Choice*. Der Unterschied liegt darin, dass mit *InterleavedParallelRouting* ein anderes Gruppenobjekt verwendet wurde, in dem eine andere OCL-Invariante gilt. Damit gilt eine andere Semantik, die mit der Invariante *atMostOneRunning* in Listing 3.8 spezifiziert wurde. Sie besagt, dass höchstens eine Aktivität der Gruppe zur gleichen Zeit ausgeführt

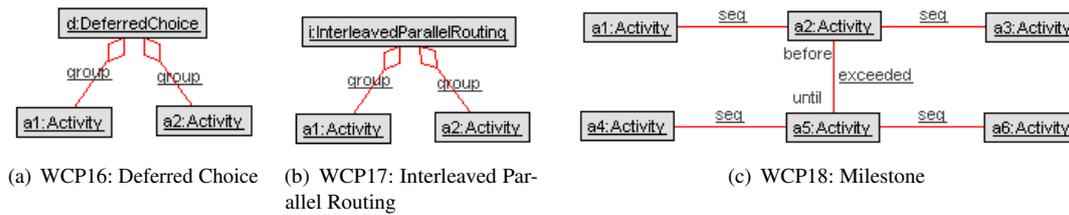


Abbildung 3.7: State-based Patterns

werden darf.

```

1 context DeferredChoice inv allWaitingXorOneRunningOthersSkipped:
2   let act_selected:Set(Activity) = activity->select(a | a.state=#running or a.state=#done) in
3   activity->forall(a | a.state=#waiting) xor
4   ( act_selected->size=1 and (activity - act_selected)->forall(ala.state=#skipped))
5
6 context InterleavedParallelRouting inv atMostOneRunning:
7   self.activity->select(a | a.state=#running)->size<=1
8
9 context Activity::start()
10 pre untilActivitiesStillWaiting:
11   let act:Set(Activity)=until.oclAsType(Group)->select(isDefined()).activity
12     ->union(until.oclAsType(Activity)->select(oclIsTypeOf(Activity)))->asSet()
13   in act->forall(a | a.state=#waiting)

```

Listing 3.8: Definition vom Deferred Choice, Interleaved Parallel Routing und Milestone Pattern

Das letzte der *State-based Patterns* ist das *Milestone Pattern* (WCP18). Es sagt aus, dass wenn ein bestimmter Ausführungspunkt im Prozessmodell erreicht wird, gewisse Aktivitäten nicht weiter ausführbar sind. Bereits gestartete Aktivitäten werden weiterhin ausgeführt.

Dieser Sachverhalt wird mit der Assoziation *exceeded* im Metamodell ausgedrückt. Die Assoziation wird in Abbildung 3.7(c) angewendet und das Pattern damit ausgedrückt. Sobald Aktivität *a5* ausgeführt wird, steht die Ausführung von Aktivität *a2* nicht mehr zur Verfügung. Es wird also Aktivität *a2* geskippt, falls sie sich im Zustand *waiting* befindet. Falls *a2* schon gestartet wurde, hat dies keine Effekte auf deren Ausführung. Jedoch ist ein nochmaliges starten untersagt. Dieser Umstand kann nicht durch eine OCL-Invariante ausgedrückt werden, sondern muss mit der Vorbedingung für die Operation *start()* verknüpft werden. In Listing 3.8 ist die Vorbedingung *untilActivitiesStillWaiting* spezifiziert, die diesen Sachverhalt ausdrückt. Die Vorbedingung ist etwas umfangreicher, weil auch Gruppen mit der *exceeded*-Assoziation verbunden werden können. Die in den Gruppen enthaltenen Aktivitäten müssen erstmal in der Variablen *act* eingesammelt werden. Daraufhin wird sichergestellt, dass sich alle *until*-Aktivitäten im Zustand *waiting* befinden. Die Vorbedingung ist für die Klasse *Activity* in Listing 3.8 angegeben. Sie hat zudem auch für alle deren Unterklassen zu gelten.

Ist die Aktivität *a2* von Abbildung 3.7(c) durch das WCP18 geskippt worden, ist eine weitere Ausführung der Folgeaktivitäten möglich. Im Falle von Abbildung 3.7(c) ist *a3* weiterhin auszuführen.

3.2.4.5 Iteration and Cancellation Patterns

In diesem Abschnitt werden die restlichen aus dem Katalog [AHKB03] verbliebenen Patterns beschrieben. Es gibt noch die Kategorie *Iteration Patterns*, zu der das *Arbitrary Cycle Pattern* (WCP10) gehört. Die

Kategorie wurde in der überarbeiteten Fassung der Workflow Patterns noch erweitert [RHAM06]. Die zusätzlich hinzugefügten Patterns betrachten wir hier jedoch nicht weiter.

Der *Arbitrary Cycle* ist in Abbildung 3.8(a) modelliert. Dieses Pattern beschreibt Zyklen in Prozessmodellen, die unstrukturiert vorhanden sind. D.h. es gibt mehr als einen Entry- bzw. Exit-Punkt für Zyklen in Prozessmodellen. Beispielsweise gibt es im Modell von Abbildung 3.8(a) zwei Einstiegspunkte für die iterative Ausführung von Aktivität *a3*. Sowohl *i1* als auch *i2* kann genutzt werden, um *nextIteration()* und damit einen neuen Ausführungszyklus im Zusammenhang mit *a3* einzuleiten. Somit ist beispielsweise eine Abarbeitungsfolge wie die folgende denkbar, in der zunächst *nextIteration()* auf dem Objekt *i2* und danach auf *i1* aufgerufen wird: *a1,a2,a3,a4,a3,a2,a3,a4,a5*

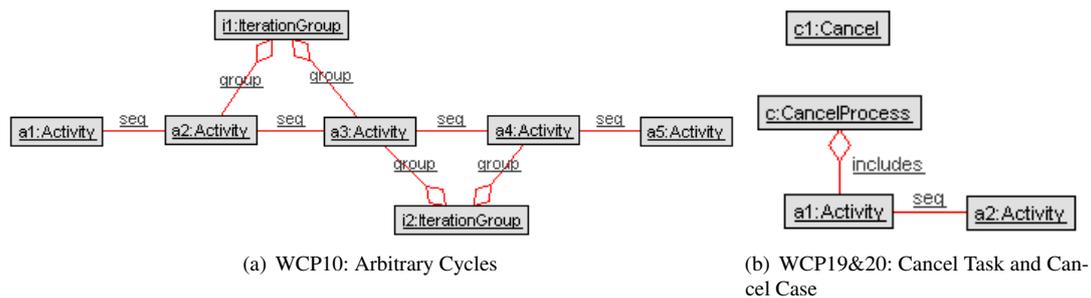


Abbildung 3.8: Iteration, Cancellation and Force Completion Patterns

Zu den *Cancellation and Force Completion Patterns* gehören dann schließlich die in Abbildung 3.8(b) dargestellten *Cancel Task* und *Cancel Case* (WCP19 & 20) Patterns. WCP19 wird durch eine einzelne Aktivität der Klasse *Cancel* repräsentiert. Im Metamodell von Abbildung 3.2 ist zu sehen, dass mit der Operation *cancel()* dem Nutzer für diese Aktivitäten die Möglichkeit bereitgestellt wird, die Aktivität abzubrechen. Zudem ist in der Enumeration *State* der Zustand *canceled* aufgeführt. Die *Cancel*-Aktivitäten können in den Zuständen *waiting* und *running* abgebrochen werden. Dieser Zustand kann nicht wieder verlassen werden. Neue Iterationen sind also mit *nextIteration()* nicht möglich.

Beim *Cancel Case* Pattern (WCP20) kann die komplette Prozessausführung abgebrochen werden. Damit würden alle wartenden und laufenden Aktivitäten in den Zustand *canceled* überführt. Hierzu gehören alle Arten von Aktivitäten, auch wenn sie keine Operation *cancel()* besitzen. Eine weitere Ausführung von Aktivitäten ist damit nicht weiter möglich, womit der komplette Prozess abgebrochen wird.

3.2.5 Ein Beispielprozess in DMWM

In Abbildung 3.9 ist ein fertiges Workflowmodell zu einer Krankenhaus-Notfallaufnahme zu sehen. Der Prozess ist mit *EmergencyProcess* bezeichnet und als oberstes Objekt im Prozessmodell vorhanden. Mit dem Objekt verbunden sind die Objekte *AdjustMedication* und *HeliAmbu*.

AdjustMedication ist eine Iterationsaktivität, die mehrfach ausgeführt werden kann. Sie hat Beziehungen zu anderen Modellelementen und ist somit keiner Ausführungsbeschränkungen unterworfen. *HeliAmbu* repräsentiert eine Deferred Choice Beziehung (s. Abschnitt 3.2.4.4) zwischen *HelicopterDelivery* und *AmbulanceDelivery* Aktivitäten. Die Ausführung einer Aktivität hat das implizite Überspringen der anderen zur Folge.

In einer Sequenz folgt dahinter die *SurgeryCheck* Gruppe, in der zunächst der Zustand des Patienten in der Aktivität *CheckPatientCondition* zu prüfen ist. Diese Aktivität repräsentiert eine *XorDecision*, in der

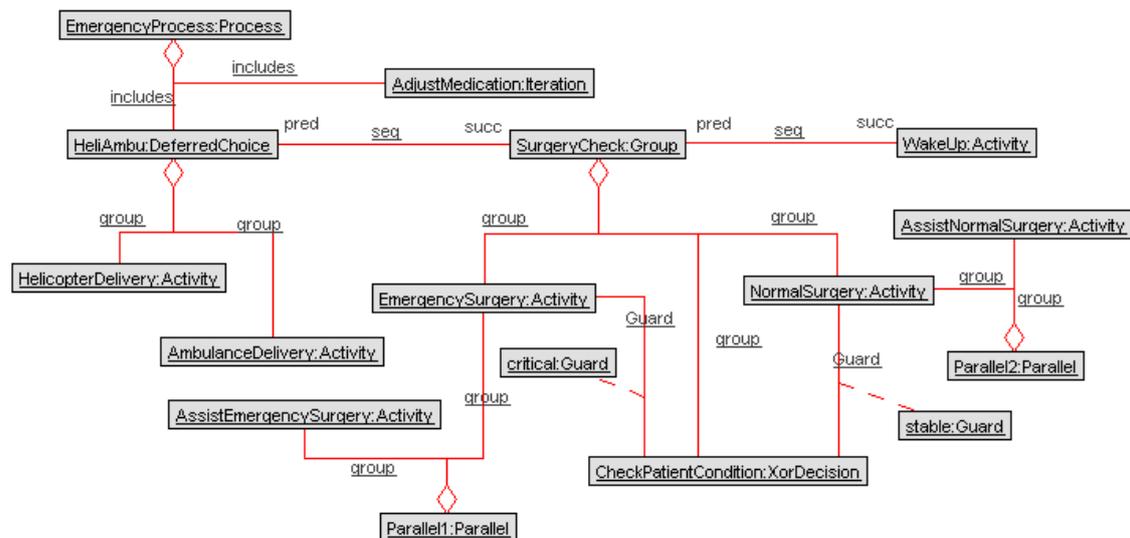


Abbildung 3.9: Ein DMWM-Workflowmodell für einen Krankenhaus-Notfallprozess als Objektdiagramm

der Nutzer basierend auf der *Guard*-Spezifikation den adäquaten Prozesspfad auswählen muss. Mit den Guards *critical* und *stable* wird angegeben, in welchem Zustand sich der Patient befindet. Wählt der Nutzer *critical* aus, so befindet sich der Patient in einem kritischen Zustand und es hat mit *EmergencySurgery* eine Notoperation zu erfolgen. Bei einem stabilen (*stable*) Zustand hat mit der Aktivität *NormalSurgery* eine normale Operation zu erfolgen. Bei den Operations-Aktivitäten wurden jeweils Assistenz-Aktivitäten mit der in Abschnitt 3.2.3.3 erwähnten *Parallel*-Beziehung modelliert. Somit haben diese Aktivitäten gezwungenermaßen parallel zu erfolgen. Als letzte Aktivität ist dann noch *WakeUp* im Prozessmodell vorhanden, die durch die *seq*-Beziehung nach allen Aktivitäten der *SurgeryCheck*-Gruppe zu erfolgen hat. Die Zustände und Attributbelegungen der Aktivitätsobjekte spielen für die in diesem Abschnitt betrachtete Modellierungsphase noch keine Rolle und können hier vernachlässigt werden. Diese werden erst für die Runtime relevant, die in Kapitel 4 im Zusammenhang mit dem Workflow-Runtime-Plugin betrachtet wird. Auch das Designtime-Plugin, das aus den Workflowmodellen ASSL-Instanzierungsprozeduren zur persistenten Speicherung und deren Folgenutzung zur Runtime generiert, wird dort näher erklärt.

3.2.6 Sicherstellung statischer Eigenschaften der Modelle

In diesem Kapitel wurden bisher nur OCL-Constraints vorgestellt, die die operationale Semantik der DMWM-Sprache betreffen. Ein wichtiger zusätzlicher Aspekt von DMWM ist die Soundness-Überprüfung während der Designtime mit OCL. Im Metamodell sind OCL-Invarianten spezifiziert, die potenzielle Deadlocks, die zur Runtime entstehen können, schon während der Designtime identifizieren.

In Listing 3.9 werden zwei Invarianten vorgestellt, die Deadlocks identifizieren. Deadlocks sind dann gegeben, wenn die auf OCL-Constraints basierenden temporale Beziehungen sich widersprechen und damit die Prozessausführung blockieren würden.

Ein solches Problem tritt z.B. dann auf, wenn der Modellierer einen Zyklus von Sequenzen bildet, welches in Abbildung 3.10 angegeben ist. Während der Runtime würde dann folgendes Problem auftauchen. Die OCL-Invariante *seqActivity* von Listing 3.7 fordert, dass die *pred*-Aktivitäten abgearbeitet sein müssen bevor die mit *self* angegebene, aktuell betrachtete Aktivität gestartet werden kann. Diese Zusicherung

```

1 context FlowObject inv noSeqCycles:
2   self.getSuccObjects(Set{ })->excludes(self)
3
4 context Activity inv notParallelAndInterleaved:
5   let parallelAct:Set(Activity)=self.group->select( glg.oclIsTypeOf(Parallel)).activity->excluding(self)->asSet() in
6   let interleavedAct:Set(Activity)=self.group->select( glg.oclIsTypeOf(InterleavedParallelRouting)).activity->
7     excluding(self)->asSet() in
   parallelAct->excludesAll(interleavedAct)

```

Listing 3.9: OCL-Invarianten zur Soundness-Überprüfung zur Designtime

hat offensichtlich ein Problem zur Folge, wenn ein Zyklus auftritt. Um das Problem schon während der Designtime zu identifizieren wird in Listing 3.9 die OCL-Invariante *noSeqCycles* vorgestellt, die Zyklen von *seq*-Links identifiziert. Mit der OCL-Hilfsfunktion *getSuccObjects()*, welche in Abschnitt 3.2.3.2 näher vorgestellt wurde, wird eine transitive Hülle unter Verwendung der *succ*-Aktivitäten gebildet. Nun fordert die Invariante, dass in der vom *self*-Objekt ausgehende transitiven Hülle dasselbige Objekt nicht enthalten darf. Wäre dieses der Fall, so hätte man einen Zyklus im Workflowmodell identifiziert.

In Abbildung 3.10 ist ein Teil des Beispielprozesses von Abbildung 3.9 angegeben und erweitert worden, so dass jetzt die Invarianten von Listing 3.9 verletzt werden. Beispielsweise wird die Invariante *noSeqCycles* durch den Zyklus von *seq*-Links bei den Aktivitäten *HeliAmbu*, *SurgeryCheck* und *WakeUp* verletzt.

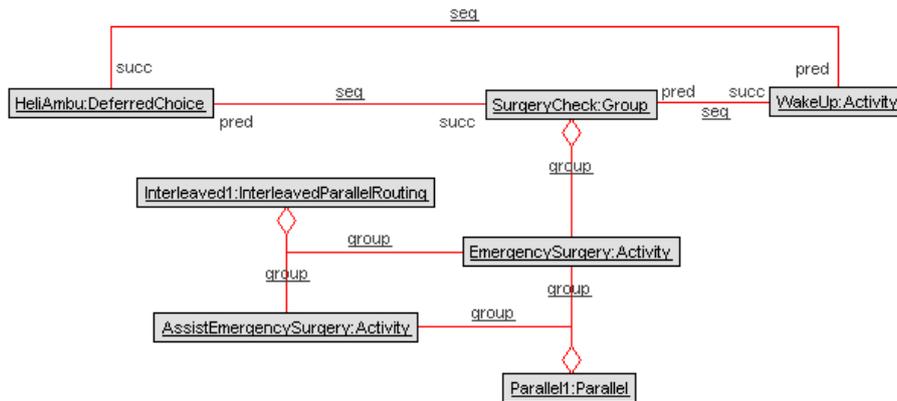


Abbildung 3.10: Modell, das die zwei OCL-Soundness-Invarianten verletzt

Die zweite in Listing 3.9 angegebene Konsistenzeigenschaft ist mit der OCL-Invariante *notParallelAndInterleaved* angegeben. Die beiden Beziehungen *InterleavedParallelRouting* und *Parallel* stehen im Widerspruch zueinander. Während die eine aussagt, dass die in der Gruppe enthaltenen Aktivitäten nicht gleichzeitig ausgeführt werden dürfen, garantiert die andere Beziehung eine zwingende parallele Ausführung der enthaltenen Aktivitäten. Sollten nun zwei Aktivitäten gleichzeitig beiden Beziehungen zugeordnet werden, liegt offenbar ein Widerspruch vor, der zu einem Deadlock zur Runtime führen würde. Um das zu verhindern, ist mit der Invarianten *notParallelAndInterleaved* in Listing 3.2.6 eine OCL-Invariante angegeben, die ein solches Problem identifiziert. Dort werden zunächst in der Variablen *parallelAct* die Aktivitäten gespeichert, die in einer *Parallel*-Beziehung mit der aktuell betrachteten *self*-Aktivität stehen. Analog werden die Aktivitäten in der Variablen *interleavedAct* eingesammelt, die zum aktuellen Objekt in einer *Interleaved Parallel Routing*-Beziehung stehen. Nun sichert die Invariante mit dem OCL-Kommando *excludesAll* zu, dass diese beiden Mengen disjunkt sein müssen. Sollte dies nicht der Fall sein, ist wie in Abbildung 3.10 angegeben, ein Problem aufgetreten. Dort sind die Aktivitäten *EmergencySurgery* und *AssistEmergencySurgery* den

beiden Beziehungen *Parallel* und *InterleavedParallelRouting* zugeordnet.

Es sind noch viele weitere Integritätsbedingungen zur Sicherstellung von Soundness-Eigenschaften denkbar. Beispielsweise stellen Zyklen von *Guard*- und *exceeded*-Assoziationen ebenso ein Problem dar. Die Beispiele von diesem Abschnitt sollen jedoch ausreichen, um exemplarisch zu zeigen, wie OCL-Invarianten im Metamodell eingesetzt werden können, um konsistente Workflowmodelle zu garantieren.

3.3 Datenintegration

In diesem Abschnitt wird die Datenintegration in Workflow-Modellierungssprachen im allgemeinen und in DMWM-Workflowmodellen im speziellen betrachtet. Die Spezifikation von Daten in Geschäftsprozessmodellen ist äußerst wichtig und Bestandteil aller populärer Prozessmodellierungssprachen, wie BPMN, Aktivitätsdiagrammen und eEPKs. Des Weiteren gibt es Datenflussdiagramme, die den Zusammenhang zwischen Funktionen, Schnittstellen und Speichern über Datenflüsse spezifizieren [Bal09]. In Abschnitt 3.3.1 werden diese näher betrachtet und gezeigt, wo Unterschiede und Probleme existieren. Daraufhin wird der Ansatz der Datenmodellierung und die Metamodellerweiterung für die Datenintegration in DMWM-Workflowmodelle in Abschnitt 3.3.2 näher eingeführt. Im DMWM-Ansatz lassen sich außerdem Datenkonsistenzbedingungen mit OCL in den Datenmodellen spezifizieren, welche in Abschnitt 3.3.3 thematisiert werden. Damit können beispielsweise fehlende Dateneinträge zur Runtime identifiziert werden und der Nutzer darauf hingewiesen werden. Außerdem wird dort ein Beispiel gegeben, an dem die Datenintegration gezeigt wird.

3.3.1 Konzepte und Probleme der Datenintegration in etablierten Modellierungssprachen

Die Datenintegration ist in eEPKs vom Kontrollflusskonzept entkoppelt. Ein Beispiel für diese Modelle ist in Abbildung 3.11(a) angegeben. In eEPKs sind die Datenspezifikationen für jede Aktivität neu anzugeben. Datenflüsse, in denen einzelne Objekte von einer Aktivität zur nächsten gegeben werden sind nicht vorgesehen.

Auch bei der in der Softwaretechnik entwickelten Strukturierten Analyse mit Realtime-Erweiterung (SA/RT) und den dort verwendeten Datenflussdiagrammen (DfDs) sind die Ablaufspezifikation (CSpecs) von Datenflüssen entkoppelt [Bal09]. Damit ist das Vorhandensein der Daten nicht Vorbedingung zur Ausführung der Funktion. Es kann z.B. auch passieren, dass während der Ausführung Daten ausgetauscht werden oder Funktionen später ihre Ausführung wieder aufnehmen. In Abbildung 3.11(d) ist ein Datenflussdiagramm zu sehen, das einen Austausch von Daten zwischen zwei Funktionen *Function1* und *Function2* vorgibt. Hier wissen wir nicht in welcher Reihenfolge diese Funktionen abgearbeitet oder ob eine Funktion mehrfach ausgeführt werden muss. Bei DfDs liegt der Fokus der Modellierung auf der Verbindung zwischen Daten, Funktionen, Schnittstellen und Speichern.

Bei Aktivitätsdiagrammen, wofür in Abbildung 3.11(c) ein Beispiel angegeben ist, stellen Objektflüsse gleichzeitig Kontrollflüsse dar. Die Objekte dienen als Eingabeparameter für die Aktivitäten und sind Vorbedingungen zur Ausführung und Konsumierung der Objekte. Die Modellierung eines expliziten Kontrollflusses ist nicht erforderlich, wenn schon ein Objektfluss vorhanden ist. Objektflussspezifikationen können aber Probleme bei der Verknüpfungen zu Flussoperatoren hervorrufen. Beispielsweise kann es Typkonsistenzprobleme geben, wenn diese mit einem *join* synchronisiert oder mit einem *fork* verzweigt werden. Auch semantische Probleme können hier auftreten, die die Effekte auf das Objektmodell bei Ausführung der Flussoperatoren betreffen. Hier stellt sich beispielsweise die Frage, was passiert, wenn zwei Token miteinan-

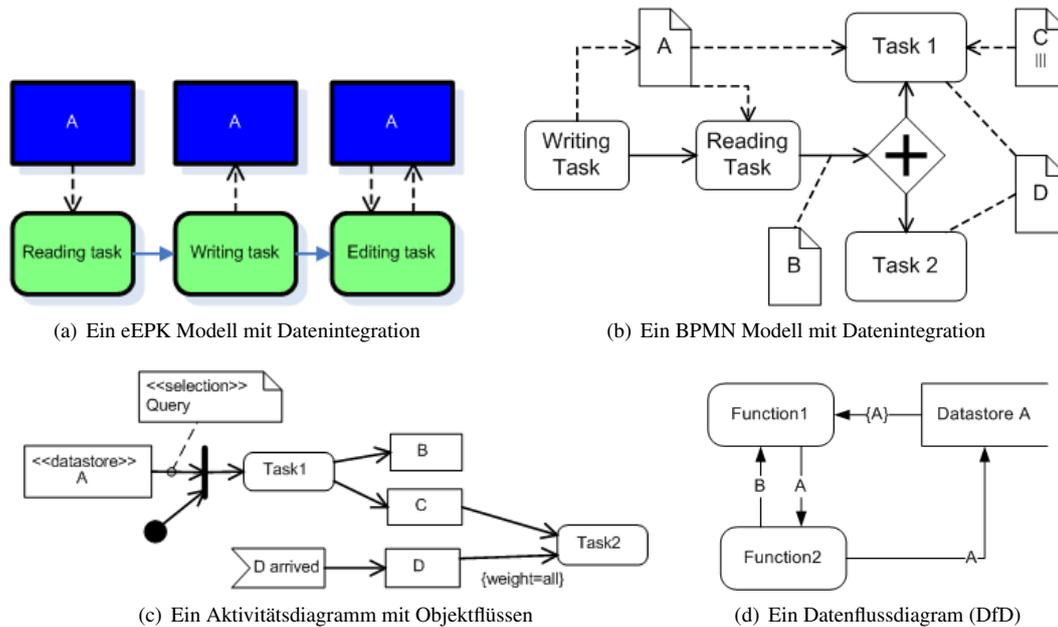


Abbildung 3.11: Datenintegration in den Modellierungssprachen EPK, BPMN, Aktivitäts- und Datenflussdiagrammen

der synchronisiert werden sollen, die auf unterschiedliche Objekte verweisen. Einerseits könnten hier Token solange aufgehalten werden, bis sich die jeweils passenden gefunden haben und die vereinigten Token dann weitergeleitet werden. Andererseits können zwei Token, die auf unterschiedliche Objekte verweisen und am *join* aufeinandertreffen einzeln weitergeleitet werden [UML10, Abs.12.3.34]. In [BF08a] sind einige dieser Probleme angesprochen und Erweiterungen für Aktivitätsdiagramme vorgeschlagen worden.

In BPMN gibt es sowohl die Möglichkeit, die Kontrollflussspezifikationen mit Daten zu koppeln als auch die Datenflüsse separat zu spezifizieren. Das Verbinden der Kontrollflüsse mit Daten ist dadurch möglich, dass an die Kontrollflusskante ein Datenobjekt assoziiert wird. In Abbildung 3.11(b) ist dies der Fall mit dem Datenelement *B* und der Kante, die aus der Aktivität *Reading task* führt. Ob der *And-Gateway* daraufhin Effekte auf dieses Datenobjekt hat, bleibt un spezifiziert. Ein weiteres Problem stellt der explizit angegebene, zu keinem Kontrollfluss assoziierten Datenfluss mit dem Element *A* dar. Hier ist unklar, ob die zwei herausführenden Kanten das gleiche Datenobjekt weitergeben oder zwei verschiedene Instanzen. Es stellt sich die Frage, ob die Bearbeitung des Datenobjekts in *Task1* Effekte auf das Datenobjekt, welches in *Reading Task* bearbeitet wird, hat.

Ein Datenaustausch von parallel ablaufenden Aktivitäten kann bei BPMN über ungerichtete Assoziationen zum Datenelement angegeben werden. In Abbildung 3.11(b) ist eine solche Spezifikation im Zusammenhang mit Datenelement *D* zu sehen. Aktivität *Task1* und *Task2* können parallel auf das Datenelement zugreifen. Eine solche Spezifikation ist mit Objektflüssen in Aktivitätsdiagrammen nicht möglich. Wie schon vorher erwähnt, ist der Kontrollfluss von der Datenintegration bei BPMN entkoppelt, so dass der Zugriff nicht gezwungenermaßen parallel geschehen muss.

Daten können des Weiteren in BPMN als Eingabe- bzw. Ausgabeparameter klassifiziert werden. Damit stellen diese Daten analog zu Objektflüssen in Aktivitätsdiagrammen Eingabeparameter bzw. Vorbedingungen zur Ausführung der Aktivität dar. Bei Aktivitätsdiagrammen, DfDs und BPMN können explizit *Datastores* verwendet werden, um eine persistente Speicherung der Daten anzugeben. In EPKs wird das Vorhalten von Daten in Datenbanken direkt implizit angenommen.

Bei UML-Aktivitätsdiagrammen können durch «*selection*» Klauseln, die an Objektfluss-Kanten annotiert werden, Kriterien zur Objektauswahl spezifiziert werden (s. Abbildung 3.11(c)). Ist der Objektfluss mit einem *Datastore*-Knoten verbunden repräsentiert diese Klausel eine Datenbankanfrage. Die Möglichkeit der Anfragespezifikation ist bei den anderen Modellierungssprachen nicht vorgesehen. Bei DfDs sind Speicher zwar integraler Bestandteil der Modellierungssprache, bei den ausgehenden Datenflüssen kann man anhand der Data-Dictionary Spezifikation jedoch nur Datentypen annotieren. Im Fall von Abbildung 3.11(d) ist mit *{A}* angegeben, dass eine Menge von Objekten des entsprechenden Typs angefragt wird. Eine Menge von den entsprechenden Objekten wird bei BPMN mit *|||* und bei Aktivitätsdiagrammen mit *Set of <Type>* in den Objektknoten angegeben. Die Spezifikation von einer Menge von Daten ist bei EPKs nicht vorgesehen.

In Aktivitätsdiagrammen können außerdem ankommende Objekte mit *AcceptEventActions* modelliert werden, um dann in den zugeordneten Objektknoten gesammelt zu werden. Zudem lassen sich die Objektflüsse gewichten, so dass nur eine gewisse Anzahl von Objekten an diesen Kanten langlaufen darf. Diese Modellierungselemente sind in Abbildung 3.11(c) angegeben und werden nur für Aktivitätsdiagramme bereitgestellt. Den anderen hier betrachteten Sprachen fehlen solche Ausdrucksmöglichkeiten.

Nach [UML10, S.411] und [Sch01, S.118f] gibt es vier Aktionen, die Aktivitäten auf Daten auslösen können:

- **read:** Lesen eines/mehrerer Datenelemente
- **create:** Anlegen eines/mehrerer Datenelemente
- **update:** Ändern eines/mehrerer Datenelemente
- **delete:** Löschen eines/mehrerer Datenelemente

Die in Abbildung 3.11 angegebenen Diagramme zeigen, dass die drei ersten oben aufgeführten Aktionen ausgedrückt werden können. Die im Folgenden aufgeführte Modellierungsweise ist bei allen angegebenen Modellierungssprachen im Prinzip gleich. Es gibt im Wesentlichen drei Verbindungen von Datenelementen zu Aktivitäten. Ein gerichteter Pfeil vom Datenobjekt zur Aktivität repräsentiert eine Leseaktion. Ein Pfeil in die umgekehrte Richtung von der Aktivität zum Datenobjekt kann eine *create* bzw. *write* Aktion ausdrücken. *Update*-Operationen können in zwei Weisen dargestellt werden. Zum einen zeigt das EPK-Beispiel von Abbildung 3.11(a), dass ein Pfeil vom Datenobjekt zur Aktivität und zurück modelliert werden kann. Zum anderen ist beim BPMN-Beispiel von Abbildung 3.11(b) eine ungerichtete Assoziation, die das Datenelement *D* mit Aktivitäten verknüpft, zu sehen. Die letzte oben aufgeführte Aktion *Löschen von Datenelementen* kann visuell in den Diagrammen jedoch nicht ausgedrückt werden.

Alle hier vorgestellten Datenintegrationen in die Workflowmodelle sind darstellbar, aber nicht ausführbar. Die in [BF08a] und oben angesprochenen Probleme bei der Verknüpfung von Objektflüssen mit Flussoperatoren auf das Objektmodell bleiben offen.

Eine eindeutigere Semantik der Objektintegration beinhaltet der DMWM-Ansatz, der in den folgenden Abschnitten näher eingeführt wird. DMWM ist auch mit der Verbindung zu den Datenspezifikationen ausführbar und die Effekte der im Metamodell definierten Elemente werden klar definiert.

3.3.2 Metamodell für Datenintegration

Zur Datenmodellierung werden UML-Klassendiagramme genutzt. Das Workflow-Metamodell liegt ebenfalls als UML-Klassendiagramm vor. Für eine Integration von Workflow- und Datenmodellen kann man die UML-Klassendiagramme vereinigen. Voraussetzung hierfür ist jedoch, dass es keine Namenskonflikte bei den Klassenbezeichnungen zwischen Datenmodell und Workflow-Metamodell gibt. Die Workflowmodellierung

(inkl. die Verbindung zu den Daten) erfolgt dann im UML-Objektdiagramm (s. Abbildung 3.13(b)). Eine Übersicht über die Verbindung von Modellen, deren Instanzen und die verwendeten Diagrammart wird in Abschnitt 4.2.3 im Zusammenhang mit den zur Runtime verwendeten Diagrammen gegeben.

Folgende Operationen werden von DMWM-Workflowmodellen unterstützt. Dabei ist die Verbindung zwischen den in Abschnitt 3.3.1 aufgeführten Aktionen *read* und *create* durch die gleiche Bezeichnungen offensichtlich. Des Weiteren wird mit *edit* das in Abschnitt 3.3.1 angegebene *update* ausgedrückt. *Delete* kann bei DMWM ebenso wie bei den etablierten Modellierungssprachen nicht modelliert werden.

- **read:** Anfragen von Daten aus Datenbestand (im Objektdiagramm)
- **edit:** Anfragen von Daten aus Datenbestand (im Objektdiagramm), die danach editiert werden können
- **create:** Erzeugung eines speziellen Objekts
- **flow:** Angabe eines Objekts, das von mehreren Aktivitäten genutzt wird

In Abbildung 3.12 ist die Metamodellerweiterung für die oben aufgeführten Datenoperationen für DMWM zu sehen. Die Klasse *Activity* ist schon aus dem DMWM-Metamodell von Abbildung 3.2 bekannt. Die drei ersten Operationen *read*, *create* und *edit* werden mit speziellen Klassen ausgedrückt, die mit der Assoziation *use* mittels der abstrakten Klasse *DataObject* mit Aktivitäten verbunden sind. In der abstrakten Klasse *DataObject* ist mit dem Attribut *classname* die Bezeichnung der betreffenden Klasse aus dem Datenmodell anzugeben. Wird beispielsweise ein *CreateData*-Objekt mit der Attributbelegung *B* angegeben, so wird zur Runtime ein Datenobjekt vom entsprechenden Typ erstellt, sobald die assoziierte Aktivität gestartet wird. Die Interpretation dieser Datenobjekte und die Präsentation der Daten für den Nutzer ist relevant für die Runtime und damit Gegenstand der Diskussion in Abschnitt 4.3.3.

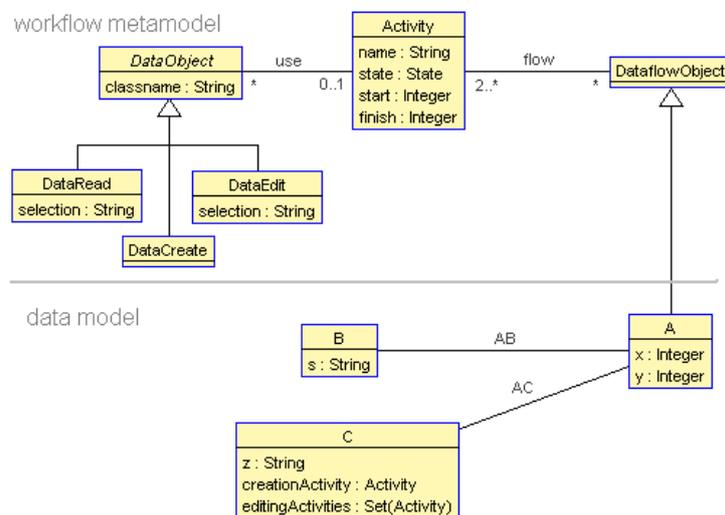


Abbildung 3.12: Metamodellerweiterung für Datenintegration in DMWM

Objekte vom Typ *DataRead* und *DataEdit* repräsentieren Datenanfragen, die in OCL-Anfragerterme überführt werden und vom USE-Workflow-Plugin zur Runtime interpretiert werden. Zur Überführung in den OCL-Anfragerterm sind die Attribute *classname* und *selection* notwendig. Anhand dieser Attribute wird folgender Anfragerterm gebildet: `<classname>.allInstances()->select(<selection>)` Es werden also alle Objekte des entsprechenden Typs mit dem Befehl *allInstances()* zusammengesammelt und darauf ein *select* ausgeführt.

Die OCL-Anfrage wird dann von USE interpretiert und die relevanten Daten in Tabellenform angezeigt. Bei Daten, die mit *DataEdit* angefragt wurden, gibt es zusätzlich für den Nutzer die Möglichkeit, Datensätze zu editieren.

Über die Assoziation *flow* werden von mehreren Aktivitäten genutzte Objekte direkt mit Aktivitäten verbunden. Diese Objekte werden als Daten gesehen, die von mehreren Aktivitäten evtl. auch parallel genutzt werden und somit Datenflüsse im Workflowmodell repräsentieren. Klassen aus dem Datenmodell, die Datenflussobjekte darstellen, werden mit einer Vererbungsbeziehung zur abstrakten Klasse *DataflowObject* aus dem Workflowmetamodell ausgedrückt.

Das Datenmodell im Klassendiagramm von Abbildung 3.12 umfasst die Klassen *A*, *B* und *C*. Die Klasse *A* repräsentiert Objekte, die gleichzeitig für Objektflüsse im Workflowmodell genutzt werden. Im Datenmodell hat diese Klasse Assoziationen zu den Klassen *B* und *C*. Des Weiteren hat die Klasse *C* die Attribute *creationActivity* und *editingActivities*. Diese Attribute stellen Verbindungen zu Aktivitäten im Workflowmodell her, die von der Workflow-Engine zur Runtime eingetragen werden. Die erzeugenden bzw. editierenden Aktivitäten werden in diesen Attributen für Objekte vom Typ *C* eingetragen. OCL-Integritätsbedingungen, die diese Informationen nutzen, werden in Abschnitt 3.3.3 erläutert. Die Anwendung der Datenobjekte zur Runtime wird Thema in Abschnitt 4.3.3 sein.

3.3.3 Datenmodellierung und Spezifikation der Datenkonsistenz in Workflowmodellen

In diesem Abschnitt wird eine Datenmodellierung und die Integration der Dateninformationen in die Workflowmodelle an einem Beispiel vorgenommen. Auf Seite des Datenmodells werden OCL-Integritätsbedingungen spezifiziert, die die Datenintegrität in Verbindung mit der Workflow-Modellausführung sicherstellen. Als Beispiel wird das in Abschnitt 3.5.3 vorgestellte Workflowmodell zur Krankenhaus-Notfallaufnahme erweitert. In Abbildung 3.13(a) ist dazu ein UML-Klassendiagramm als Datenmodell angegeben. Das Workflowmodell, das Elemente aus dem Datenmodell verwendet ist in Abbildung 3.13(b) im UML-Objektdiagramm zu sehen. Es wurde ein Ausschnitt vom Notfallprozess von Abbildung 3.9 mit einer Datenintegration angereichert.

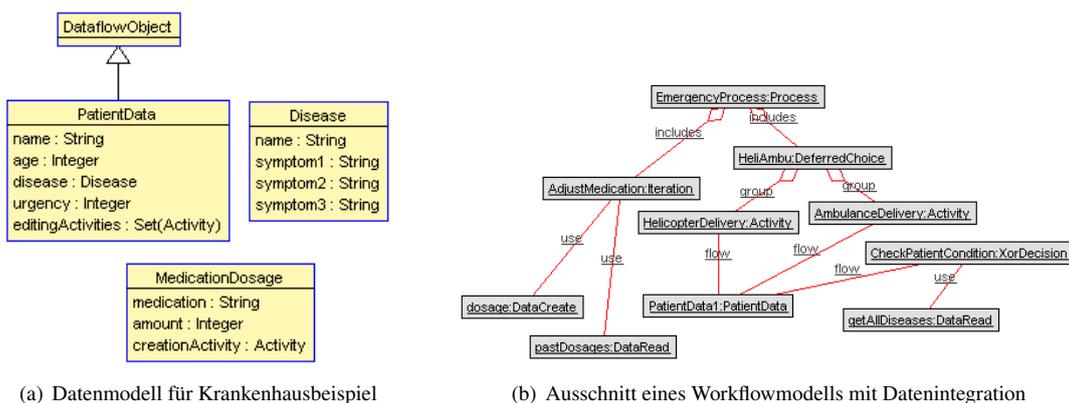


Abbildung 3.13: Daten- und Workflowmodell mit Datenintegration

Das Datenmodell von Abbildung 3.13(a) umfasst die Klassen *PatientData*, *Disease* und *MedicationDosage*. Mit *PatientData* werden Datensätze zum Patienten erfasst. Diese Daten beinhalten den Namen, das Alter des Patienten, die festgestellte Krankheit und die Dringlichkeit, mit der der Patient zu behandeln ist. Diese

Patientendaten sind zentraler Bestandteil eines Notfallprozesses und werden von vielen Aktivitäten benötigt. Das Workflowmodell von Abbildung 3.13(b) zeigt die Verbindungen des *PatientData*-Objekts zu den Aktivitäten *AmbulanceDelivery*, *HelicopterDelivery* und *CheckPatientCondition*.

Die Klasse *Disease* modelliert im Datenmodell Krankheiten und ihre Symptome, die vom Krankenhaus-Informationssystem zu speichern sind. Im Attribut *name* wird die Bezeichnung der Krankheit festgehalten und in den drei *symptom*-Attributen werden drei zugehörige Symptome angegeben. Im Objektmodell von USE wird zur Runtime eine Menge von *Disease*-Objekten vorhanden sein, die die Datensätze aus dem Krankenhaus-Informationssystem repräsentieren. Anhand dieser Informationen soll der Arzt vom Informationssystem unterstützt werden. Die Zuordnung der Krankheit findet über das Attribut *disease* und die Dringlichkeit über das Attribut *urgency* in der Klasse *PatientData* statt. Die Aktivität *CheckPatientCondition* ist eine Entscheidungsaktivität, in der auf Grundlage der verbundenen Datenelemente entschieden werden muss, ob der Patient sich im Zustand *stable* oder *critical* befindet. Die Aktivität ist mit dem *PatientData*-Objekt verbunden und somit ist dieses Objekt während der Ausführung der Aktivität zu editieren bzw. zu lesen. Neben den Daten zum Patienten ist das *DataRead*-Objekt *getAllDiseases* mit der Aktivität verbunden. Dieses Objekt repräsentiert die Datenbankabfrage an das Informationssystem. Das Attribut *classname* (s. Abbildung 3.12) ist mit *'Disease'* und das *selection*-Attribut mit *'true'* belegt, so dass alle Krankheits-Datensätze abgefragt werden. Das Anfrageobjekt *getAllDiseases* wird dafür in den OCL-Anfragerterm *Diseases.allInstances()->select(true)* umgewandelt und zur Runtime von USE interpretiert.

In der *Iteration*-Aktivität *AdjustMedication* ist die Verabreichung von Medikamenten anhand von *MedicationDosage*-Objekten zu dokumentieren. Wenn die Aktivität gestartet wird, wird anhand des Objekts *dosage* vom Typ *CreateData* ein *MedicationDosage*-Objekt erstellt. Die Aktivität, die das *MedicationDosage*-Objekt erstellt hat wird als Referenz in das Attribut *creationActivity* von der Workflow-Engine eingetragen. Anhand dessen können Integritätsbedingungen (s. Listing 3.10) und Datenanfragen gestellt werden. Mit *getPastMedications*, die im Workflowmodell mit der Aktivität *AdjustMedication* verbunden ist, wird eine solche Anfrage gestellt. Der OCL-Anfragerterm *Medication.allInstances()->select(creationActivity=self)* wird aus den Attributbelegungen des *DataRead*-Objekts erstellt. Die Regeln dafür wurden in Abschnitt 3.3.2 angegeben. Somit werden alle bisherigen Medikationen des Patienten abgefragt und dem Anwender während der Ausführung der Aktivität präsentiert. Eine mögliche Überdosierung oder Doppelmedikation kann dadurch verhindert werden.

Auf Seite des Datenmodells sind OCL-Constraints spezifiziert, um gegenseitige Workflow- und Datenmodell-Konsistenzeigenschaften sicherzustellen. In Listing 3.10 sind zwei OCL-Invarianten angegeben, die den Nutzer während der Workflow-Ausführung auf fehlende Dateneinträge aufmerksam machen sollen. Für die Constraints werden die Attribute *editingActivities* bei der Klasse *PatientData* und das Attribut *creationActivity* bei der Klasse *MedicationDosage* im Datenmodell benötigt.

```

1 context MedicationDosage inv MedicationEntriesMade:
2   creationActivity.state = #done implies medication.isDefined() and amount.isDefined()
3
4 context PatientData inv DiseaseAndUrgencyEnteredAfterDelivery:
5   ( editingActivities->isDefined() and editingActivities->exists((name='AmbulanceDelivery'
6     or name='HelicopterDelivery') and state=#done)) implies
7     disease.isDefined() and urgency.isDefined()

```

Listing 3.10: OCL-Invarianten zur Datenintegritätsprüfung im Zusammenhang mit Workflowausführung

Die Invariante *MedicationEntriesMade* drückt aus, dass die Attribute *medication* und *amount* eingetragen sein müssen, wenn die erzeugende Aktivität abgeschlossen ist. Während der Ausführung des Workflowmo-

dells von Abbildung 3.13(b) wird bei jedem Starten der Aktivität *AdjustMedication* ein *MedicationDose*-Objekt erstellt und die Rückreferenz zur erstellenden Aktivität im Attribut *creationActivity* hinterlegt. Somit kann der Zustand der erzeugenden Aktivität abgefragt und anhand dessen überprüft werden, ob die entsprechenden Einträge gemacht wurden.

Die Invariante *DiseaseAndUrgencyEnteredAfterDelivery* ist der Klasse *PatientData* zugeordnet und stellt sicher, dass nach dem Transport des Patienten die Datenfelder *disease* und *urgency* eingetragen sein müssen. In dem Attribut *editingActivities* werden alle Aktivitäten eingetragen, bei denen das Objekt über *DataflowObject* oder *DataEdit*-Objekte angefragt und Attribute während der Ausführung editiert wurden. Betrachtet man das Workflowmodell von Abbildung 3.13(b), so sind die beiden Transport-Aktivitäten *AmbulanceDelivery* und *HelicopterDelivery* über eine *DeferredChoice*-Beziehung verbunden. Wird eine von diesen Aktivitäten ausgeführt, so wird das verbundene *PatientData*-Objekt zum editieren freigegeben. Nun ist in der Invariante zu prüfen, wenn eine Transportaktivität im Attribut *editingActivities* auftaucht und diese beendet wurde, so müssen die angegebenen Attribute im *PatientData*-Objekt eingetragen sein.

Listing 3.10 zeigt, dass die Invarianten zur Spezifikation der Konsistenz zwischen Workflow- und Datenmodell nicht ganz intuitiv zu lesen und verstehen sind. Im Gegensatz zum Erstellen des Workflowmodells muss der Modellierer beim Erstellen des Datenmodells gleichzeitig textuelle OCL-Invarianten spezifizieren. Hier können recht schnell Fehler passieren und eine Validation der Modelle ist daher notwendig. Genau für solche Zwecke stellt die in Abschnitt 4.3 vorgestellte Validierungsfunktionalität eine Möglichkeit bereit, die Spezifikation zu prüfen. Im Speziellen lassen sich bei DMWM die Korrektheitseigenschaften anhand der Workflow-Ausführung zur Runtime mit dem USE-Workflowplugin validieren, was in Kapitel 4 näher erläutert wird.

3.4 Organisationsmodellierung bei DMWM

Neben der Datenmodellierung ist die Organisationsmodellierung ein zweiter wichtiger Aspekt der Workflow- bzw. Geschäftsprozessmodellierung. Entsprechend spielt beim ARIS-Ansatz (s. in Abschnitt 2.5.2.1) das Organisationsmodell eine wichtige Rolle. Dort wird ein hierarchischer Ansatz gewählt. Ein Beispiel ist in der Abbildung 2.2(b) angegeben. Das Unternehmen repräsentiert die Wurzel und die Hierarchieebenen darunter geben die Abteilungen an. Die Abteilungen können rekursiv aufgegliedert werden, bis in den Blättern des Organisationsbaums, den Abteilungen Rollen zugeordnet werden.

Im Folgenden wird die metamodellbasierte Organisationsmodellierung in DMWM vorgestellt. Dafür wird in Abschnitt 3.4.1 das Metamodell zur Organisationsmodellierung für DMWM eingeführt. Außerdem wird die Verbindung vom Organisationsmodell zum Workflowmodell über das DMWM-Metamodell hergestellt. In Abschnitt 3.4.2 wird dann das Metamodell angewendet, um ein Organisationsmodell an einem Beispiel zu erstellen. Auch die Zuordnung von Organisationseinheiten bzw. Rollen zu Aktivitäten wird dort geschehen.

3.4.1 Metamodell für die Organisationsmodellierung

Für DMWM wird ähnlich zum ARIS-Ansatz eine hierarchische Organisationsmodellierung verfolgt. Ebenso wie beim Metamodell für ARIS [Sch01] wurde also ein Organisationsmetamodell entwickelt, das eine solche (hierarchische) Modellierung ermöglicht. In Abbildung 3.14 ist das DMWM-Metamodell zur Organisationsmodellierung angegeben.

Neben dem ARIS-Metamodell für die Organisationsmodellierung [Sch01] wurden auch bei den objektorientierten Analyse-Patterns [FCJ96] ähnliche Klassendiagramme für eine hierarchische Organisationsmodel-

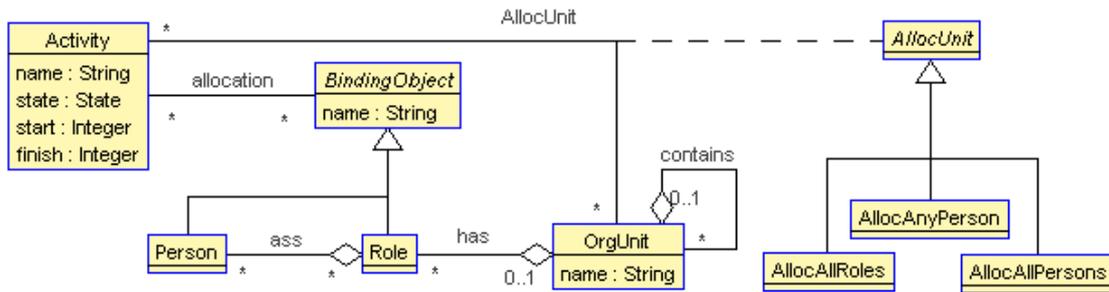


Abbildung 3.14: DMWM-Metamodel für die Organisationsmodellierung

lierung vorgeschlagen. Es wurden für beide Ansätze reflexive Assoziationen in den Klassendiagrammen verwendet. Analog wurde auch beim DMWM-Metamodel zur Organisationsmodellierung eine reflexive Assoziation namens *contains* in Abbildung 3.14 eingesetzt. Die Zuordnung von Abteilungen zur Rollen wird über die Assoziation *has* ausgedrückt. Über die Assoziation *ass* wird des Weiteren angegeben, welche Personen welche Rollen haben.

Während der Designtime werden Aktivitäten zu Rollen über die Assoziation *allocation* zugeordnet. Zur Runtime wird dann die Allokation von Personen vorgenommen, die die entsprechenden Rollen vertreten. Die Funktionalität wird über eine Nachbedingung für die *start()*-Operation von der Klasse *Activity* und deren Unterklassen im DMWM-Metamodel deklarativ beschrieben. Sie ist in Listing 3.11 angegeben. In den Variablen *assignedRoles* und *assignedPersons* werden die zur aktuell betrachteten Aktivität zugeordneten Rollen und Personen gespeichert. Nun sichert die Nachbedingung zu, dass nach dem Starten der Aktivität für alle zur Designtime zugeordneten Rollen eine Person zur Ausführung assoziiert sein muss. Eine imperative Allokationsprozedur, die beim Aufrufen der *start()*-Operation ausgeführt wird, muss dann die angegebene Nachbedingung erfüllen. Sie ist mit ASSL umgesetzt und wird für die Runtime benötigt, die in Kapitel 4 Thema der Diskussion sein wird.

```

1 context Activity::start()
2 post allocationDone:
3   let assignedRoles:Set(Role)=self@pre.bindingObject.oclAsType(Role)->select(isDefined())->asSet() in
4   let assignedPersons:Set(Person)=self.bindingObject.oclAsType(Person)->select(isDefined())->asSet() in
5   assignedRoles->forall(r|assignedPersons->exists(plr.person->includes(p)))

```

Listing 3.11: OCL-Invariante zur Allokation von Personen für die Aktivitätsausführung

In eEPK-Modellen können Aktivitäten auch ganzen Abteilungen zugeordnet werden (s. [STA05]). Da EPKs rein zur Veranschaulichung der Geschäftsprozessmodelle geeignet sind und nicht zur Ausführung, bleibt es dort unterspezifiziert, welche und wie viele Personen der Abteilung für die Durchführung der Aktivität benötigt werden. Für die Ausführung der Workflowmodelle ist diese Information jedoch erforderlich. Im Metamodel von Abbildung 3.14 ist ersichtlich, dass Aktivitäten mit Organisationseinheiten anhand der Assoziationsklasse *AllocUnit* bei DMWM verbunden werden können. Die Assoziationsklasse ist noch abstrakt. Verbindungen zwischen Aktivitäten und Organisationseinheiten können nur anhand der konkreten Unterklassen vorgenommen werden. Hier unterscheidet DMWM drei Arten der Allokation anhand der drei folgenden Unterklassen.

- *AllocAllRoles* gibt an, dass zur Ausführung der verbundenen Aktivität eine Person der assoziierten

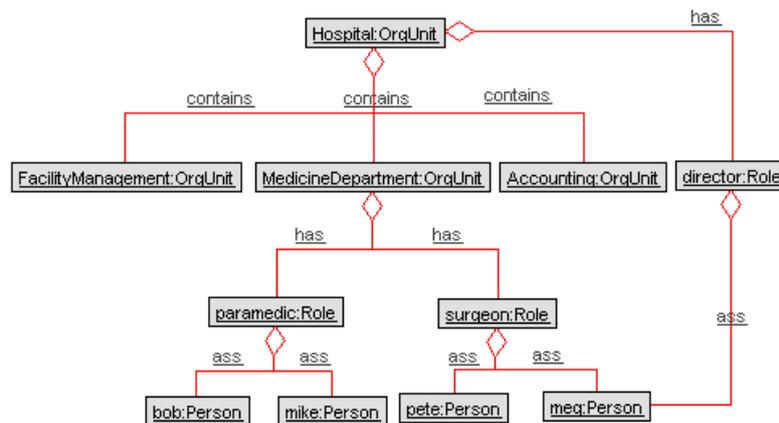
Abteilung benötigt wird.

- Mit *AllocAnyPerson* wird modelliert, dass genau eine Person der Abteilung zur Ausführung der Aktivität benötigt wird. Jede Person der Abteilung ist dabei berechtigt, die Aktivität auszuführen.
- Bei *AllocAllPersons* wird angegeben, dass alle Personen der Abteilung zur Ausführung der Aktivität vorgesehen sind.

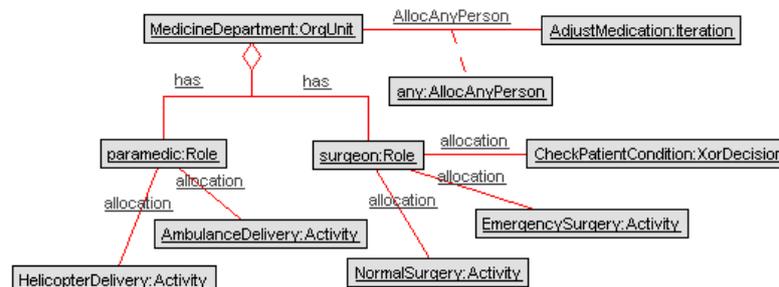
Analog zur Nachbedingung von Listing 3.11 sind für die oben aufgeführten Organisationseinheit-zu-Personen-Allokationen ebenfalls Nachbedingungen für die *start()*-Operation im DMWM-Metamodell spezifiziert, die hier aber nicht weiter angegeben werden.

3.4.2 Organisationsmodellierung und Zusammenhang zu Workflowmodellen

In diesem Abschnitt wird in Abbildung 3.15 das Metamodell angewendet, um für das in Abschnitt 3.2.5 eingeführte Notfall-Krankenhaus-Beispiel eine Organisationsmodellierung für ein Krankenhaus zu erstellen. Auch die Zuordnungen von Aktivitäten zu Rollen und Organisationseinheiten zur Designtime wird vorgestellt.



(a) Ein Organisationsmodell für ein Krankenhaus



(b) Allokationsmodell für einen Teil des Notfallprozess

Abbildung 3.15: Organisationsmodell für Krankenhaus und Aktivitäts-Allokationsmodell

In Abbildung 3.15(a) ist ein hierarchisches Organisationsmodell als UML-Objektdiagramm zu sehen, das aus dem Organisationsmetamodell von Abbildung 3.14 erzeugt wurde. Als Wurzel des Organisationsbaums ist das gesamte Krankenhaus mit *Hospital* bezeichnet. Dieses gliedert sich in drei Abteilungen

Gerätemanagement (*FacilityManagement*), medizinische Abteilung (*MedicineDepartment*) und Verwaltung (*Accounting*). Im Organisationsbaum sind nun zwei Rollen der medizinischen Abteilung zugeordnet. Die Rolle des Rettungsassistenten (*paramedic*) und die des Chirurgen (*surgeon*) sind hierfür im Modell vorhanden. Weiterhin sind jeweils zwei Personen vorhanden, die die beiden obigen Rollen übernehmen: *Bob* und *Mike* sind Rettungsassistenten und *Pete* und *Meg* übernehmen die Rolle des Chirurgen. Außerdem hat *Meg* die Rolle der Krankenhausdirektorin inne. Damit ist es nach dem Metamodell also erlaubt, dass Personen mehrere Rollen übernehmen können. Es ist anzumerken, dass das hier eingeführte Organisationsmodell nur ein Ausschnitt eines wirklichen Organisationsmodells ist und als Beispiel zu verstehen ist.

In Abbildung 3.15(b) sind Aktivitäten vom Notfallprozess von Abbildung 3.9 zu sehen. Zum einen sind die Transportaktivitäten *HelicopterDelivery* und *AmbulanceDelivery* jeweils mit der Rolle des Rettungsassistenten verbunden. Zum anderen sind die ärztlichen Aktivitäten *NormalSurgery*, *EmergencySurgery* und *CheckPatientCondition* dem Chirurgen zugeordnet. Eine weitere Zuordnung ist bei der Verabreichung von Medikamenten bei der Aktivität *AdjustMedication* vorhanden. Sie ist über ein Objekt der Assoziationsklasse *AllocAnyPerson* mit der medizinischen Abteilung verbunden. Das bedeutet, dass jede Rolle bzw. jede Person dieser Abteilung zur Ausführung dieser Aktivität berechtigt ist. Im Falle des hier betrachteten Modells sind das alle vier Personen.

Die Ausführung der Workflowmodelle inklusive imperativer Allokationsprozeduren, die unterschiedliche Strategien implementieren können, betreffen die Runtime, die in Kapitel 4 näher betrachtet wird.

3.5 Realisieren von DMWM mit USE

Mit der in den vorherigen Abschnitten vorgestellten UML- und OCL-Spezifikation für DMWM ist die Basis gegeben, ein Modellierungswerkzeug für die Designzeit und eine Workflow-Engine für die Runtime zu entwickeln. Dagegen kann man auch bestehende UML-Werkzeuge wie z.B. das *Eclipse Modeling Framework* [Gro09] oder die *UML-based Specification Environment* (USE) [GBR07] zur Umsetzung von DMWM verwenden. Bei einer solchen modellbasierten Umsetzung kann man sich viel Implementierungsarbeit sparen. Für diese Dissertation wurde das UML-Werkzeug USE gewählt, da dort u.a. mit ASSL eine OCL-basierte imperative Sprache bereitsteht, mit der die operationale Semantik für DMWM umgesetzt werden kann. Die Modellausführung mit ASSL ist Thema in Kapitel 4.

Im Folgenden wird in Abschnitt 3.5.1 das Werkzeug USE näher vorgestellt. Daraufhin wird die Benutzung des USE-Tools für DMWM mit Anwendungsfällen in Abschnitt 3.5.2 verdeutlicht. Die Benutzung des Tools zur Modellierung eines Workflows ist Thema in Abschnitt 3.5.3. Schließlich beschreibt Abschnitt 3.5.4 die automatische Soundness-Analyse der Workflowmodelle. Die dafür bereitgestellten Funktionalitäten und Werkzeuge von USE werden dafür näher beleuchtet.

3.5.1 UML-Entwicklungsumgebung USE

Die UML-based Specification Environment (USE) ist ursprünglich dazu entwickelt worden, UML-Klassendiagramme zusammen mit OCL-Constraints zu validieren. USE wird im Zusammenhang mit den in Abschnitt 2.1 vorgestellten leichtgewichtigen formalen Methoden in der Softwareentwicklung genutzt. Hier werden die Modelle während der Entwurfsphase auf Eigenschaften getestet.

Der Entwickler spezifiziert ein Modell als UML-Klassendiagramm zusammen mit OCL-Constraints. Nun wird anhand des Modells ein Abbild eines laufenden Systems in Form eines UML-Objektdiagramms durch Manipulation des Systemzustands erzeugt. Daraufhin prüft das USE-System OCL-Constraints und der

Entwickler das Ergebnis. Bei festgestellten Inkonsistenzen können zwei Arten von Fehlern festgestellt werden. Entweder das Modell oder der erstellte Snapshot vom laufenden System ist nicht zulässig.

Mit leichtgewichtigen formalen Methoden zur Softwareentwicklung [GBR07, Jac03] können Entwurfsfehler schon während der Analysephase festgestellt und behoben werden. Die dadurch validierten Modelle stellen einen Qualitätsgewinn dar, der verhindern soll, dass sich Modellierungsfehler bei der Softwareentwicklung in die Implementierungsphase fortpflanzen. Je später ein Fehler entdeckt wird, desto schwieriger ist es, ihn zu beheben. Das USE-Tool unterstützt den Entwicklungsprozess mit den im Folgenden aufgeführten UML-Diagrammart.

- **UML-Klassendiagramm:** Diese Diagramme werden anhand einer textuellen Spezifikation von USE eingelesen. Die Modelle werden syntaktisch validiert. Treten Fehler (beispielsweise Typfehler, doppelte Bezeichnungen o.ä.) auf, wird der Nutzer darauf hingewiesen. Zusätzlich sind in der gleichen Datei die OCL-Invarianten und OCL-Vor- und Nachbedingungen zu formulieren. Auch hierfür findet eine Syntaxanalyse statt. Zusätzlich wird eine semantische Analyse vollzogen, indem die Modelle auf Eigenschaften, die von der UML-Spezifikation her verboten sind, untersucht werden. Darunter zählen z.B. Zyklen in Vererbungshierarchien (vgl. [RG00]). Ist das Modell schließlich korrekt eingelesen worden, wird es im Tool grafisch dargestellt.
- **UML-Objektdiagramm:** Auf Grundlage des vorher eingelesenen Klassendiagramms lassen sich Objektdiagramme erzeugen. *create/destroy object*, *insert/remove links* und *set attribute values* sind imperative Befehle dafür. Die damit entstehenden Diagramme stellen sogenannte *Snapshots* eines (laufenden) Systems dar. Dies kann z.B. der Zustand eines laufenden Programmes zu einem bestimmten Zeitpunkt sein.

Objektdiagramme können einerseits grafisch vom Nutzer erzeugt werden. Andererseits können sie auch halbautomatisch anhand der ASSL-Sprache generiert werden [GBR05]. Näheres zur Verwendung von ASSL im DMWM-Ansatz wird in Abschnitt 4.2 folgen.

Während der Erstellung der Objektzustände wird der Zustand permanent von USE auf OCL-Constraintverletzungen überwacht. Hier können zum einen modellinhärente Inkonsistenzen, wie z.B. Kardinalitätsverletzungen bei Assoziationen geschehen. Zum anderen können explizit spezifizierte OCL-Zusicherungen fehlschlagen. Hier liegt die Aufgabe des Entwicklers den Fehler im Systemzustand oder in der Spezifikation zu finden. Dafür stellt USE zur Unterstützung des Nutzers Funktionalitäten und Werkzeuge bereit.

- **UML-Sequenzdiagramm:** Dynamisches Verhalten wird anhand von OCL-Vor- und Nachbedingungen deklarativ im UML-Klassendiagramm spezifiziert. Daraufhin können diese Spezifikationen durch Folgen von Snapshots validiert werden. Der Programmablauf wird anhand dieser Diagramme in USE protokolliert. USE unterstützt dabei ausschließlich synchrone Operationsaufrufe. Bei Sendebotschaften wird die OCL-Vorbedingung und bei Antwortbotschaften die Nachbedingung geprüft. Eine Bedingungsverletzung wird durch einen roten Pfeil im Sequenzdiagramm visuell hervorgehoben.

USE lässt sich zusätzlich zur UML-modellbasierten Softwareentwicklung auch als Metamodellierungswerkzeug nutzen. Metamodelle definieren dabei häufig domänenspezifische Sprachen, so wie es schon in Abschnitt 2.2 erwähnt wurde. Klassendiagramme repräsentieren dabei Metamodelle. OCL-Invarianten legen im Metamodell die Integritätsbedingungen für die zu entwickelnde Sprache fest. Die Objektdiagramme stellen daraufhin eine abstrakte Syntax der vom Metamodell beschriebenen Sprache dar. Mit der oben beschriebenen Validierungsmethode wurden bereits Teile des UML-Metamodells mit USE getestet und dortige Fehler aufgedeckt [RG00].

Bei *DMWM*-Ansatz wird USE ebenfalls metamodelbasiert zur Workflowmodellierung eingesetzt. Im folgenden Abschnitt wird näheres dazu beschrieben.

3.5.2 Anwendungsfälle zur Modellierung von DMWM mit USE

Im hiesigen Abschnitt soll nun konzeptionell erörtert werden, wie UML, OCL und die Funktionalität des USE-Tools genutzt werden, um Workflowmodelle zu erstellen.

Eine nähere Betrachtung, wie der Entwickler mit *DMWM* Workflowmodelle erstellt, ist im Use Case-Diagramm in Abbildung 3.16 angegeben. Dort wird zusätzlich mit den Anwendungsfällen *create data model* und *create organisational model* verdeutlicht, dass zu den Workflowmodellen zusätzliche Modelle zur Unternehmensmodellierung im *DMWM*-Ansatz verwendet werden können.

Die Spezifikation der Verbindungen dieser Modelle wird durch die Anwendungsfälle *model data-* und *organisational dependencies* ausgedrückt und ist Thema in den Abschnitten 3.3 und 3.4 gewesen. Die Spezifikation der Verbindungen ist optional. Beim Weglassen würden die Modelle separat ohne gegenseitigen Zusammenhang existieren. Spezifiziert man jedoch die Verbindungen, erweitern diese Anwendungsfälle die entsprechenden Use Cases zur Erstellung der Workflow-, Daten- und Organisationsmodelle. Daher sind die Use Cases zur Angabe der Verbindungen mit der *extend*-Beziehung zu den relevanten Use Cases zur Erstellung der eigentlichen Modelle verbunden.

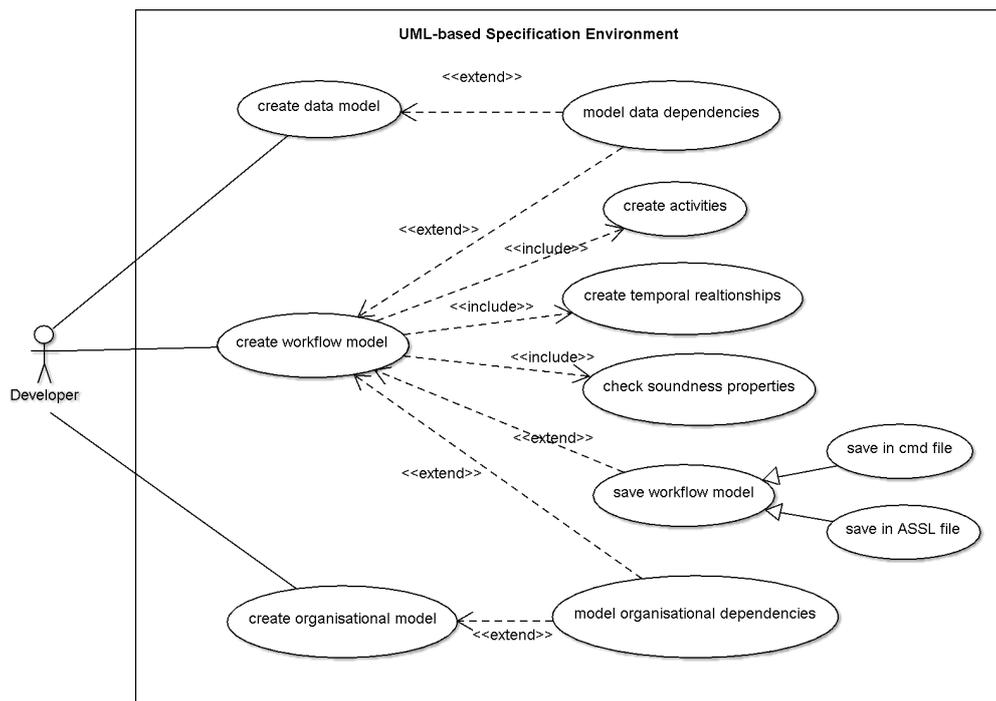


Abbildung 3.16: Einsatz des USE Tools zur Workflowmodellierung

Zunächst ist der Kern der Workflowmodellierung jedoch das Modellieren der Aktivitäten und der temporalen Beziehungen zwischen diesen. Dieses ist in Abbildung 3.16 über den Anwendungsfall *create activities* und *create temporal relationships* ausgedrückt, die über die *includes*-Beziehung zwangsläufig zu jeder Erstellung eines Workflowmodells (Use Case *create workflow model*) gehört. Dieses Thema ist in den Abschnitten 3.2 und 3.2.4 behandelt worden.

Während der Modellierung werden Soundness-Eigenschaften der Workflowmodelle überprüft. Dieses wird in Abbildung 3.16 mit dem Use Case *check soundness properties* ausgedrückt. Für den leichtgewichtigen formalen DMWM-Ansatz werden die Workflowmodelle als *sound* betrachtet solange sie keine Inkonsistenzen wie Deadlocks oder Livelocks beinhalten. Näheres dazu wurde bereits in Abschnitt 3.2.6 behandelt.

Ist das Workflowmodell erstellt, gibt es zwei Möglichkeiten dieses abzuspeichern und für die Runtime nutzbar zu machen. Zum einen kann mit dem Anwendungsfall *save to cmd file* das Workflowmodell einer USE-Kommandodatei gespeichert werden, die den Zustand des Workflowmodells reproduzieren kann. Damit ist lediglich eine Prozessinstanz vom Workflowmodell reproduzierbar. Zum anderen wurde ein Designtime-Plugin entwickelt, das die Workflowmodelle persistent in ASSL-Dateien speichert. Ruft man die ASSL-Instanzierungsprozedur mehrfach während der Runtime auf, sind mehrere Prozessinstanzen des modellierten Workflows erzeugbar. Das Designtime-Plugin wird im Zusammenhang mit der ASSL-Sprache in den Abschnitten 4.1 und 4.2 erläutert.

3.5.3 Modellierung eines Workflows

Das USE-Tool wird zunächst mit dem vorgestellten Workflow Metamodell geladen. Daraufhin wird die Oberfläche von USE angezeigt, welche in Abbildung 3.17 zu sehen ist.

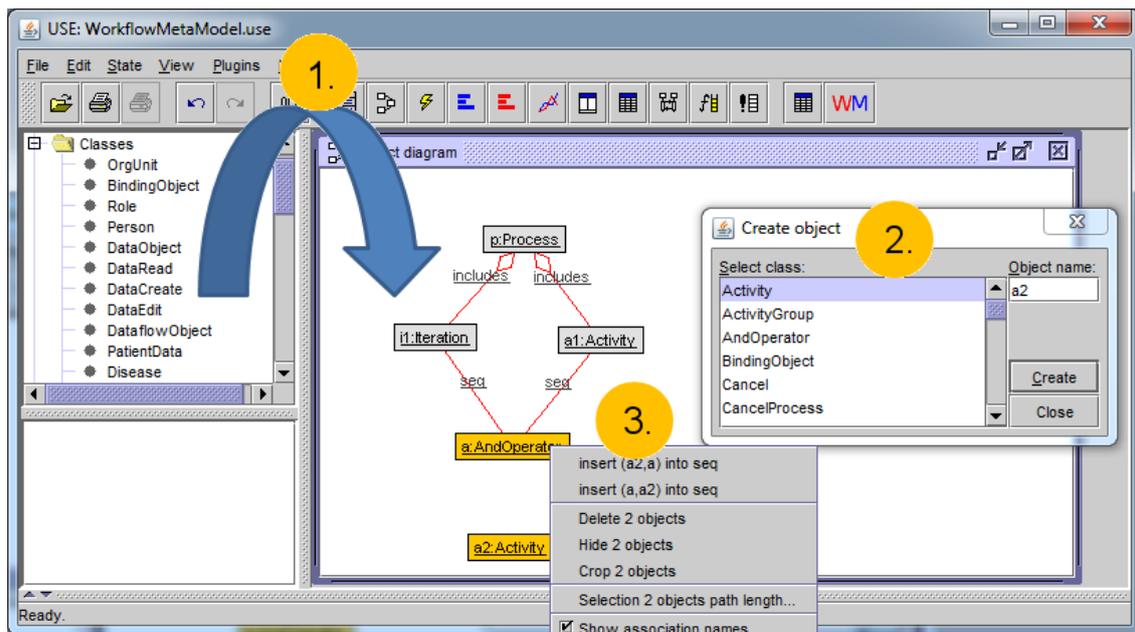


Abbildung 3.17: Modellierungsoberfläche vom UML-Werkzeug USE

Es gibt zwei Möglichkeiten, Aktivitätsobjekte zu erzeugen, die mit den Verweisen 1. und 2. in Abbildung 3.17 angegeben sind. Zum einen werden links oben im Fenster die Metamodellelemente bereitgestellt. Diese kann der Modellierer per *Drag and Drop* nutzen, um Aktivitäten zu erzeugen (s. Verweis 1. in der Abbildung). Dabei wird die Objekt-ID für das Aktivitätsobjekt automatisch von USE vergeben. Zur Bezeichnung der Aktivität ist es dann notwendig, dass der Modellierer die Aktivität anhand des Attributes *name* benennt.

Die zweite Möglichkeit, Aktivitätsobjekte zu erzeugen, ist über den *Create object view* in USE gegeben (s. Fenster in Abbildung 3.17, das mit 2. gekennzeichnet ist). Hierbei kann der Modellierer die Aktivitäten

anhand der Objekt-ID bezeichnen. Ein fertiges Workflowmodell, das anhand dieser Methode erstellt wurde, ist in Abbildung 3.9 zu sehen.

Die Beziehungen zwischen den Aktivitäten werden durch die Links zwischen den Objekten modelliert. Hier können zwei Objekte markiert werden und ein Rechtsklick ausgeführt werden. Dem Modellierer werden dann die aus dem Metamodell möglichen Verbindungen vorgeschlagen. In Abbildung 3.17 kann eine *seq*-Beziehung zwischen Objekt *a* und *a2* hergestellt werden (mit Nummer 3 in der Abbildung angegeben). Dabei sind zwei Richtungen möglich, den Link zu erzeugen, die die Abarbeitungsreihenfolge der Aktivitäten angibt. Gruppen, Flussoperatoren, andere Elemente aus dem Workflowmetamodell von Abbildung 3.2 und die Verbindung zwischen ihnen werden analog dazu im USE-Objektdiagramm erzeugt.

3.5.4 Unterstützung des Nutzers bei Modellerstellung

In diesem Abschnitt wird gezeigt, dass auch ungeübte Nutzer Fehler und die verursachenden Objekte recht einfach identifizieren und dann beheben können. Der Nutzer muss nicht zwangsläufig mit OCL vertraut sein, um Hinweise auf Probleme im Prozessmodell zu bekommen. Hierfür wird das Workflowmodell und die Konsistenzbedingungen von Abschnitt 3.2.6 herangezogen und ausgewertet.

Der *Class invariant view* aus Abbildung 3.10 gibt schon einen groben Überblick über die Invarianten und deren Auswertungen. Es sind dort alle Invarianten aufgelistet. Mit *true* wird angegeben, dass die entsprechende Invariante erfüllt ist. Dagegen wird mit *false* eine Invariantenverletzung angezeigt. Es fehlt bei dieser Ansicht jedoch die wichtige Information, welche Objekte die Invariantenverletzungen hervorrufen.

Ein wichtiges Analysewerkzeug wird mit dem *Class extent view* in USE bereitgestellt, welches in Abbildung 3.18(a) zu sehen ist. Dort kann eine Klasse ausgewählt werden, deren Objekte mit Attributbelegungen und Invariantenauswertungen dargestellt werden.

Am Beispiel von Abbildung 3.18(a) wurde die Klasse *Activity* ausgewählt. Die Attributbelegungen für *finish*, *start* und *state* sind zu vernachlässigen. Interessant sind die Invariantenauswertungen für jedes Objekt. Mit einem grünen Häkchen wird ausgedrückt, dass das Objekt die Invariante erfüllt, bei einem roten Kreuz verletzt das Objekt die Invariante. Anhand der Bezeichnung der Invariante und der Zuordnung der verletzenden Modellelemente kann der Modellierer das Problem im Modell identifizieren und beheben. Beispielsweise besagt der *Class extent view* in Abbildung 3.18(a), dass die Objekte *AssistEmergencySurgery* und *EmergencySurgery* die Invariante *notParallelAndInterleaved* verletzen. Außerdem ist ersichtlich, dass die Aktivität *WakeUp* die Invariante *noSeqCycles* verletzt. Auf Basis dieser Informationen kann der Modellierer das Prozessmodell im Objektdiagramm aus Abbildung 3.10 analysieren und die Fehler beheben.

Für eine intensivere Untersuchung der Invariantenverletzung kann der *Evaluation browser* in USE verwendet werden. Diese Analyse ist vor allem für Nutzer geeignet, die mit OCL vertraut sind. In Abbildung 3.18(b) ist ein Ausschnitt zur Auswertung der Invariante *noSeqCycles* angegeben. Der OCL-Term wird in einen abstrakten Syntaxbaum aufgegliedert und dargestellt. Terme werden in Unterterme rekursiv aufgegliedert und als Kinder im Baum hinzugefügt. Die Aufspaltung endet in den Blättern im Baum mit atomaren nicht weiter aufteilbaren Termen, wie z.B. Attribut-, Variablenbezeichner oder beliebigen Werten. Attribut- bzw. Variablenbezeichner werden mit Werten belegt, die aus dem USE-Objektdiagramm stammen. Die Auswertung wird dann ausgehend von den Blättern wieder bis zur Wurzel zurückgeführt und damit der Gesamtterm ausgewertet.

Es gibt mehrere Darstellungsfunktionen im *Evaluation browser*, um die OCL-Auswertungen für den Endnutzer verständlicher zu machen. Beispielsweise ist in Abbildung 3.18(b) auf der rechten Seite im oberen Fenster eine Variablenbelegung des im Baum aktuell markierten Terms zu sehen. Im Baum ist die vierte Zeile markiert, in der die Variable *self* gerade mit dem Objekt *@WakeUp* belegt ist. Des Weiteren ist

Activity	finish	name	start	state	notParallelAndInterleaved	seqActivity	noSeqCycles
AmbulanceDelivery	Undefined	'AmbulanceDelivery'	Undefined	State::waiting	✓	✓	✓
AssistEmergencySurgery	Undefined	'AssistEmergencySurgery'	Undefined	State::waiting	✗	✓	✓
AssistNormalSurgery	Undefined	'AssistNormalSurgery'	Undefined	State::waiting	✓	✓	✓
EmergencySurgery	Undefined	'EmergencySurgery'	Undefined	State::waiting	✗	✓	✓
HelicopterDelivery	Undefined	'HelicopterDelivery'	Undefined	State::waiting	✓	✓	✓
NormalSurgery	Undefined	'NormalSurgery'	Undefined	State::waiting	✓	✓	✓
WakeUp	Undefined	'WakeUp'	Undefined	State::waiting	✓	✓	✗

(a) Class extent view

The screenshot shows the 'Evaluation browser' window. The main area displays a tree view of evaluation results. The root node is 'FlowObject.allInstances->forAll(self: FlowObject | self.getSuccObjects(Set {})->excludes(self)) = false'. Below it, several sub-terms are listed, with their evaluation results highlighted in bold and color-coded: 'self.getSuccObjects(Set {})->excludes(self) = true' (bold black), 'self.getSuccObjects(Set {})->excludes(self) = false' (white on black), and 'newElems->isEmpty = false' (bold black). The right-hand pane shows the current context: 'self : Activity = @WakeUp' and 'Set{@HeliAmbu,@SurgeryCheck,@WakeUp}->excludes(@WakeUp) = false'. At the bottom, there are 'Display options' and 'Close' buttons.

(b) Evaluation browser

Abbildung 3.18: Toolunterstützung zur Identifizierung von Soundness-Problemen

im unteren Fenster eine aktuelle Auswertung der Subterme zu sehen, in der die Subterme mit ihren Werten substituiert worden sind.

Außerdem gibt es die Möglichkeit zur visuellen Hervorhebung von Boolean-Termen im Auswertungsbaum. Mit *true* ausgewertete Terme werden in Abbildung 3.18(b) mit einer **fetten** Schrift hervorgehoben und mit *false* ausgewertete Terme werden mit inverser Farbe (weiße Schrift auf schwarzem Hintergrund) dargestellt. Zusätzlich gibt es hierfür auch die Möglichkeit zur visuellen Hervorhebung der Subbäume. Es gibt noch diverse weitere Funktionalitäten, die hier nicht weiter aufgezählt, aber in [Dat07] erklärt werden.

Kapitel 4

Metamodellbasierte Workflow-Modellausführung

Die bisherige Betrachtung des DMWM-Ansatzes betraf vorwiegend die Modellierung von Workflows zur Designtime. Wobei im Metamodell aber durchaus schon die Grundlagen zur Ausführung der Modelle gelegt wurden, indem die operationale Semantik anhand von OCL-Bedingungen hinterlegt wurde. Diese Spezifikation repräsentiert eine plattformunabhängige Beschreibung für die neu entwickelte Workflowsprache.

Die Runtime für den DMWM-Ansatz wird in diesem Kapitel behandelt. Diese ist nicht weiter plattformunabhängig, da sie mit der OCL-basierten imperativen Sprache ASSL verbunden ist, die vom UML-Tool USE interpretiert werden kann. Ein großer Teil der Implementierungsarbeit für DMWM bestand darin, die Sprache ASSL zu erweitern, um die imperativen Ausführungsprozeduren zu realisieren, welches in Abschnitt 4.2.1 näher beschrieben wird. Ein weiterer großer Teil lag darin, eine grafische Schnittstelle mit dem USE-Workflow-Runtime-Plugin zu schaffen, um dem Entwickler eine komfortable Möglichkeit für die Ausführung der Modelle zu geben und diese damit zu validieren.

Dieses Kapitel gliedert sich in vier Abschnitte. In Abschnitt 4.1 wird die Einleitung und Motivation des entwickelten DMWM-Tools anhand von Anwendungsfällen gegeben. Daraufhin wird in Abschnitt 4.2 die Sprache ASSL mit den für DMWM entwickelten Erweiterungen vorgestellt. Der damit realisierte imperative Teil des Workflow-Metamodells ist ebenfalls Gegenstand der Betrachtung. Auf das Workflow-Runtime-Plugin mit allen seinen Funktionen wird in Abschnitt 4.3 näher eingegangen. Die Analyse der ausgeführten Prozesse lässt sich mit dem Tool USE anhand von UML-Sequenzdiagrammen, Log-Windows und OCL-Processmining-Anfragen durchführen, was Thema in Abschnitt 4.4 sein wird. Schließlich werden noch verwandte Arbeiten zur Modellierung und Ausführung von Prozessmodellen im Abschnitt 4.5 betrachtet.

4.1 Einführung

Ein wichtiger Aspekt bei DMWM ist die Ausführung und Validation der Workflowmodelle. Gleichzeitig wird mit der Ausführung das Metamodell und die dort integrierten Constraints validiert. Im Laufe der Entwicklung wurden mit dieser Methode nicht nur Fehler in Workflowmodellen sondern auch Fehler im Workflow-Metamodell herausgefunden. In Abbildung 4.1 sind die Anwendungsfälle aufgelistet, für die das Runtime-Plugin von DMWM entwickelt wurde.

Anhand des USE-Tools und des Workflow-Plugins lassen sich die Workflowmodelle instanzieren und

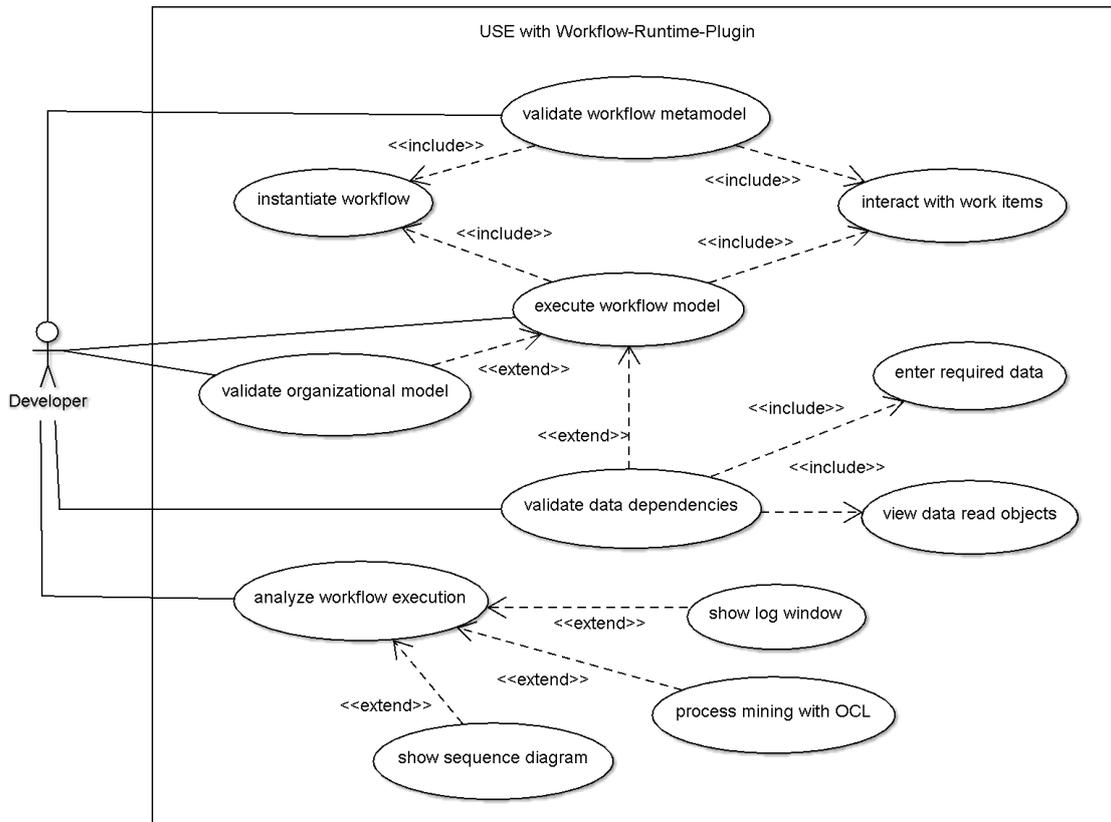


Abbildung 4.1: Anwendungsfälle für DMWM zur Runtime und das entwickelte Workflow-Plugin

ausführen. Dieses ist mit den beiden Anwendungsfällen *instantiate workflow* und *interact with work items* ausgedrückt. Für das Instanzieren stellt das Runtime-Plugin eine Funktionalität bereit, um ASSL-Instanzierungsprozeduren aufzurufen und damit Prozessinstanzen zu erzeugen. Das persistente Speichern von Workflowmodellen in ASSL-Instanzierungsprozeduren übernimmt das Designtime-Plugin, welches in Abschnitt 4.2.4 vorgestellt wird. Der Anwendungsfall *interact with work item* beinhaltet das Starten, Beenden oder Überspringen von Aktivitäten. Auch weitere Operationen aus dem Workflow-Metamodell von Abschnitt 3.2 werden zur Interaktion bereitgestellt. Diese werden realisiert mit dem Runtime-Plugin, basierend auf den ASSL-Prozeduren und OCL-Constraints aus dem Workflow-Metamodell, die in Abschnitt 3.2.4 vorgestellt wurden. Anhand der Ausführung der Workflowmodelle wird sowohl das Workflow-Metamodell als auch das Workflowmodell validiert. Bei Fehlern ist entweder das Verhalten des Workflowmodells auf Basis eines falschen Metamodells nicht korrekt oder das Workflowmodell ansich wurde zur Designtime falsch spezifiziert. Die Gründe für ungewolltes Verhalten lassen sich über diverse Funktionalitäten von USE, welche u.a. in Abschnitt 3.5.4 vorgestellt wurden, herausfinden.

Während der Ausführung der Workflowmodelle können zusätzlich Abhängigkeiten vom Workflow- zum Organisationsmodell abgetestet werden. Der Anwendungsfall *validate organizational model* kann also *execute workflow model* erweitern. Dafür verwendete ASSL-Allokationsprozeduren ordnen den Aktivitäten Personen zu. Diese Funktionalität ist Thema in Abschnitt 4.2.2.

Ein weiterer Aspekt, der während der Ausführung der Workflow-Modelle relevant ist, ist die Einbeziehung der Datenobjekte und die damit verbundene Validierung des Datenmodells. Notwendige Dateneinträge

sind über OCL-Constraints im Datenmodell spezifiziert, was bereits in Abschnitt 3.3.3 thematisiert wurde. Im Runtime-Plugin werden die Objekte angezeigt und der Nutzer hat die notwendigen Daten einzugeben (Anwendungsfall *enter required data*). Während der Ausführung der Aktivitäten werden angefragte Datenobjekte ebenfalls dargestellt (Anwendungsfall *view data read objects*). Die Funktionalität des Runtime-Plugins zur Interaktion mit Datenobjekten wird in Abschnitt 4.3.3 beleuchtet.

Schließlich können die ausgeführten Workflowinstanzen auf mehrere Weisen analysiert werden. Dazu gehört das Log-Window im Workflow-Plugin, welches die Aktionen vom Nutzer während der Workflow-Ausführung inkl. Timestamps protokolliert. USE stellt ein OCL-Anfragefenster bereit, das eine Analyse auf Basis von OCL Process-Mining-Anfragen ermöglicht. Eine visuelle Variante, den Ablauf der Workflowinstanz zu erfassen, ist mit dem UML-Sequenzdiagramm in USE gegeben. In diesem Diagramm werden die Operationsaufrufe bei den Aktivitäten geloggt und dargestellt. Die Aufrufe können ausgehend vom Nutzer über das Workflow-Plugin oder durch Seiteneffekte ausgehend von anderen Aktivitätsausführungen auf Basis temporalen Beziehungen geschehen. Die Analysemöglichkeiten von ausgeführten Workflows werden in Abschnitt 4.4 näher erklärt.

4.2 Instanziierung und Grundlagen zur Ausführung der Modelle

USE stellt nicht nur die Modellierungsumgebung bereit, sondern auch die Ausführungsumgebung für die Workflowmodelle. Von USE können zwei Sprachen interpretieren werden, die Teile der *UML Action Semantics* umsetzen. Eine dieser Sprachen ist *A Snapshot Sequence Language (ASSL)* [GBR07] und die andere *A Simple OCL-based Imperative Language (SOIL)* [Büt11]. ASSL ist schon länger in USE integriert, wohingegen SOIL recht aktuell parallel zum DMWM-Ansatz entwickelt wurde. Für DMWM war die Sprache zu neu, so dass ASSL für die imperative Umsetzung Verwendung fand.

4.2.1 ASSL Sprachvorstellung und Erweiterung

A Snapshot Sequence Language (ASSL) wurde zur Validierung und Zertifizierung von UML-Modellen implementiert [GBR05]. Anhand von ASSL-Prozeduren lassen sich in USE halbautomatisch Objektdiagramme aus Klassendiagrammen erzeugen. Dafür implementiert ASSL große Teile der UML-Action Semantics, wie z.B. Erstellen, Löschen von Objekten oder Links und Setzen von Attributwerten von Objekten. In Tabelle 4.1 sind die ASSL Befehle inkl. Erklärung dazu angegeben. Die Befehle werden in ASSL-Prozeduren in einer Pascal-ähnlichen Syntax geschrieben [GBR07]. Analog zu Pascal können auch Variablen deklariert und dann in OCL-Termen genutzt werden, nachdem ihnen Objekte bzw. Werte zugeordnet wurden. Es stehen zudem *for*-Schleifen und *if*-Bedingungen zur Verfügung.

Die grundlegenden Befehle zur Erzeugung von Objekten, Links und Setzen von Attributwerten ist essenziell für die DMWM-Workflowausführung. Zusätzlich musste ASSL erweitert werden, indem die drei Befehle *ASSLCall*, *OpEnter* und *OpExit* hinzugekommen sind. Diese werden umfangreich genutzt, was in Abschnitt 4.2.2 näher vorgestellt wird. Für die Erweiterungen musste der bereits umfangreiche auf ANTLR [Par07] basierende Interpretationsmechanismus des ASSL-Interpreters erweitert werden. In [BHW11] wurden die entsprechenden Erweiterungen vorgestellt.

Eine Auffälligkeit bei ASSL ist das explizite Kennzeichnen von OCL-Termen in den ASSL Kommandos mit eckigen Klammern. In Tabelle 4.1 ist das zu sehen beim *Zuweisen eines Attributwertes*. Auch in den folgenden Listings 4.1 und 4.2 ist diese Eigenart auffällig. Bei der Sprache SOIL wurde diese wieder beseitigt, so dass die OCL dort direkter integriert ist.

ASSL-Befehl	Erklärung zum Befehl
Create(<Classname>)	Erzeugung eines Objekts
Insert(<Association>,<Objectlist>)	Erzeugung eines Links zwischen mehreren Objekten (i.d.R. zwei)
[<Object>].<attr>:=[OCLTerm]	Zuweisen eines Attributwertes
ASSLCall <procname>(<parameters>)	Aufruf einer weiteren ASSL-Prozedur. Es kann auch rekursive Aufrufe damit realisiert werden
OpEnter <objectID> <Operation>(<parameters>)	Eine Operation aufrufen und die OCL-Vorbedingungen überprüfen
OpExit	Eine Operation verlassen und die OCL-Nachbedingungen überprüfen

Tabelle 4.1: Die für DMWM verwendeten ASSL-Befehle und Erklärungen dazu

Das Überschreiben von Operationen lässt sich mit ASSL sehr gut lösen, wenn man einige Eigenschaften der Interpretationsweise der ASSL-Dateien beachtet. Das Überschreiben von Operationen ist dabei für die Umsetzung von DMWM äußerst wichtig. Beispielsweise Verhält sich die *finish()*-Operation der Klasse *Activity* anders als die der Klasse *Iteration* oder *Decision* vom Workflow-Metamodell aus Abbildung 3.2. Also muss diese Operation in den entsprechenden Unterklasse überschrieben werden.

Die Interpretationsweise der ASSL-Dateien ist wie folgt. USE parst die ASSL-Datei und die erste ASSL-Prozedur wird zur Ausführung genommen, zu der der Aufruf und die übergebenen Objekte zur Signatur passen. Beispielsweise sind somit die Operationen *procedure finish(d:Decision)* und *procedure finish(i:Iteration)* vor *procedure finish(a:Activity)* zu schreiben. Steht die Prozedur mit einer Signatur mit allgemeineren Typen davor, würde diese Prozedur zur Ausführung genommen werden. Im obigen Fall würde *procedure finish(a:Activity)* für alle übergebenen Aktivitäts-Objekte genommen werden, auch für die vom Typ *Iteration* und *Decision*. Die Operationen der spezielleren Klassen sind also vor den der generelleren Klassen in der ASSL-Datei anzuordnen.

Zwei spezifische Kommandos der ASSL Sprache sind *Try* und *Any*. Anhand dieser Befehle kann die schon erwähnte halbautomatische Erzeugung der Objektdiagramme erfolgen. Beim *Try* Befehl wird eine Sequenz von Objekten übergeben. Hier wird versucht, ein Objekt aus der Sequenz zu finden, das im Zusammenhang mit der weiteren ASSL Prozedurausführung ein Objektdiagramm findet, das keine OCL-Bedingungen verletzt. Bei *Any* wird ein beliebiges Objekt aus der Menge gewählt. Die Überprüfung, ob OCL-Bedingungen verletzt sind, wird immer nach kompletter Abarbeitung der ASSL-Prozedur vorgenommen. Für intern über *ASSLCall* aufgerufene Prozeduren gilt dieses nicht. Es wird der Zustand des Objektdiagramms betrachtet, der nach kompletter Abarbeitung der Ausgangsprozedur vorhanden ist.

4.2.2 Imperative Teil des Metamodells

Die verschiedenen Operationen vom Workflow-Metamodell von Abbildung 3.2 werden imperativ mit der Sprache ASSL implementiert. Dies bildet die Grundlage zur Ausführung der Workflowmodelle mit dem Workflow-Plugin. In Listing 4.1 ist ein Auszug der ASSL-Prozedur angegeben, die die *start()*-Operation der Klasse *Activity* realisiert. Diese Prozedur soll hier als Beispiel genügen. In analoger Weise wurden alle weiteren Operationen des DMWM-Metamodells umgesetzt.

Der ASSL *start()*-Prozedur wird die zu startende Aktivität als Parameter übergeben. Daraufhin werden mit dem Befehl *OpEnter* die OCL-Vorbedingungen der *start()*-Operation getestet, die im DMWM-Metamodell hinterlegt sind (s. Abschnitt 3.2.3.1). Schlägt hier die Bedingung fehl, wird die komplette Ausführung der ASSL-Prozedur zu keinem Ergebnis führen. In Zeile 5 wird dann der Zustand der Aktivität auf *running*

```

1 procedure start(a:Activity)
2 var setA:Set(Activity);
3 begin
4   OpEnter [a] start();
5   [a].state:=[#running];
6   for gr:Group in [a.group->asSequence] begin
7     -- Deferred Choice Group
8     if [gr.oclIsTypeOf(DeferredChoice)] then begin
9       setA := [gr.activity->select(a2|a2.state=#waiting)];
10      for a2:Activity in [setA->asSequence] begin
11        ASSLCall skip([a2]);
12      end;
13    end;
14    -- Parallel Group
15    if [gr.oclIsTypeOf(Parallel)] then begin
16      setA := [gr.activity->select(a2|a2.<a and a2.state<>#running)];
17      for a2:Activity in [setA->asSequence] begin
18        ASSLCall start([a2]);
19      end;
20    end;
21  end;
22  -- calculate and skip all exceeded activities
23  setA := [...];
24  for a1:Activity in [setA->asSequence] begin
25    ASSLCall skip([a1]);
26  end;
27  -- allocate resources
28  ASSLCall allocation([a]);
29  OpExit;
30 end;

```

Listing 4.1: *start()*-Operation der Klasse *Activity* umgesetzt mit ASSL

gesetzt.

In den darauf folgenden Zeilen 6-26 werden die Seiteneffekte, die das Starten der aktuellen Aktivität auf andere Aktivitäten hat, umgesetzt. Zunächst werden in der *for*-Schleife die *Deferred Choice* und *Parallel* Gruppen geprüft. Je nachdem, ob Aktivitäten vorhanden sind, die in einer dieser Beziehungen stehen, werden diese mittels Aufruf durch *ASSLCall* geskippt oder gestartet. Daraufhin werden die Aktivitäten berechnet, die durch die *exceeded*-Beziehung übersprungen werden müssen. Der OCL-Term ist recht umfangreich und wurde in Zeile 23 weggelassen.

Die Prozedur ruft, bevor mit *OpExit* die Nachbedingungen überprüft werden, eine ASSL-Allokationsprozedur auf, die eine auszuführende Person aus dem Organisationsmodell zuordnet. Diese Prozedur ist in Listing 4.2 angegeben und verwendet das Organisations-Metamodell von Abbildung 3.14. Zur Runtime ist hierfür das Organisationmodell ebenfalls notwendig.

Die Allokationsprozedur beinhaltet ein Mapping von Rollenzuordnungen zu Personen. Die Person darf hier keiner zur Zeit laufenden Aktivität zugeordnet sein. Sie wird anhand des OCL-Terms in Zeile 6 ausgewählt. Eine beliebige Person, die diese Kriterien erfüllt, wird mit dem ASSL-Befehl *Any* der Variablen *p* zugewiesen. Dann wird diese mit der Aktivität anhand der Assoziation *allocation* verbunden.

Daraufhin werden alle Verbindungen der Aktivität zu Organisationseinheiten betrachtet. Hier können nach dem Metamodell von Abbildung 3.14 potenziell mehrere Verbindungen bestehen. Das bedeutet, dass evtl. mehrere Organisationseinheiten für die Ausführung zuständig sind. Es wird für jede verbundene

```

1 procedure allocation(a:Activity)
2 var p:Person;
3 begin
4   -- Role–Person allocation
5   for r:Role in [a.bindingObject.oclAsType(Role)–>select(isDefined())–>asSequence()] begin
6     p:=Any([r.person–>select(activity–>forall(state<>#running))–>asSequence()]);
7     Insert(allocation, [a], [p]);
8   end;
9   -- OrgUnit–Person allocation
10  for u:AllocUnit in [a.allocUnit] begin
11    ASSLCall allocation([u]);
12  end;
13 end;

```

Listing 4.2: Allokationsprozedur umgesetzt mit ASSL

Organisationseinheit eine ASSL-Allokationsprozedur aufgerufen. Für jede der drei Klassen *AllocAllRoles*, *AllocAnyPerson* und *AllocAllPersons* ist eine ASSL-Allokationsprozedur definiert. Anhand des dynamischen Bindens wird die gewünschte Prozedur automatisch aufgerufen und die Verbindung von der Aktivität zu Personen hergestellt. Die Umsetzung dieser Prozeduren wird hier jedoch nicht weiter betrachtet.

4.2.3 Verhältnis Metamodelle, Modelle und Instanzen

Bei dem DMWM-Ansatz muss zwischen Metamodell, Modell und Modellinstanzen unterschieden werden. Um einen Überblick über die verschiedenen Modelle zur Workflow-, Daten- und Organisationsmodellierung und die dafür verwendeten Diagrammart zu bekommen, wird in Tabelle 4.2 eine Übersicht gegeben.

DMWM-Diagrammart	Metamodell	Modell (Designtime)	Modellinstanz (Runtime)
Workflow	UML-Klassendiagramm	UML-Objektdiagramm	UML-Objektdiagramm angereichert mit Ausführungsdaten
Datenmodell	bereits mit USE implementiert	UML-Klassendiagramm	UML-Objektdiagramm
Organisationsmodell	UML-Klassendiagramm	UML-Objektdiagramm mit Aktivitätszuordnungen zu Rollen oder Organisationseinheiten	UML-Objektdiagramm mit Personenzuordnungen

Tabelle 4.2: Die verwendeten UML-Diagramme für DMWM

Das Workflow-Metamodell liegt in einem UML-Klassendiagramm angereichert um OCL-Constraints vor, welches in Kapitel 3 sehr ausführlich vorgestellt wurde. Anhand des USE-Tools lassen sich die Workflowmodelle zur Designtime mit UML-Objektdiagrammen erstellen. In diesem Objektmodell sind die Attribute für die Ausführungsdaten, wie Aktivitätszustände oder Zeitstempel, noch nicht relevant und daher nicht belegt.

Das Designtime-Plugin nutzt das Workflowmodell, um eine ASSL-Prozedur zu generieren, die das Modell persistent speichert. Das Vorgehen wird in Abschnitt 4.2.4 näher vorgestellt. Ruft man die generierte Instanziierungsprozedur zur Runtime auf, wird das Prozessmodell reproduziert und die Zustände aller Aktivitäten auf *waiting* gesetzt. Wird die Instanziierungsprozedur mehrfach aufgerufen, so können mehrere Instanzen des gleichen Prozesses erzeugt werden.

Bei der Datenmodellierung werden die von USE unterstützten UML-Klassendiagramme genutzt. Eine Metamodellierung ist hierfür nicht notwendig. USE implementiert bereits einen großen Teil des UML-Metamodells für Klassendiagramme. Dieses wird zur Design-time zur Datenmodellierung für DMWM eingesetzt. Vor Ausführung der Prozessmodelle zur Runtime können die Datenobjekte erzeugt werden, die die Datenbasis eines Informationssystems repräsentieren. Die Workflowmodelle können dann beispielsweise mit *DataRead*-Objekten die gewünschten Daten anfragen. Während der Ausführung können des Weiteren Objekte mit *DataCreate* erzeugt werden, die damit dem Datenbestand des Informationssystems hinzugefügt werden.

Das Organisations-Metamodell liegt so wie das Workflow-Metamodell im Klassendiagramm vor. Für DMWM sind im UML-Klassendiagramm damit das Datenmodell zusammen mit dem Workflow- und Organisations-Metamodell vorhanden. Anhand des Organisations-Metamodells wird die hierarchische Organisationsmodellierung zur Design-time im UML-Objektdiagramm durchgeführt, so wie es in Abschnitt 3.4 gezeigt wurde. Hier werden die Verbindungen von Aktivitäten zu zuständigen Rollen und Organisationseinheiten hergestellt. Durch die ASSL-Allokationsprozeduren (s. Listing 4.2) werden die ausführenden Personen festgelegt. Zur Design-time können unterschiedliche Organisationsmodelle erstellt werden, wovon ein bestimmtes zur Runtime auszuwählen ist. Damit können Workflowmodelle in unterschiedlichen Kontexten ausgeführt und evtl. Ressourcenengpässe identifiziert werden. Das gewünschte Organisationsmodell muss dann vor Ausführung der Workflowmodelle in das Objektdiagramm von USE geladen und damit für die Runtime zur Verfügung gestellt werden. Zur Runtime kommt dann für das Organisationsmodell die Zuordnungsinformation zur ausführenden Person in Form von *allocation*-Links hinzu.

4.2.4 Workflow Design-time-Plugin und Toolchain

Das Design-time-Plugin für USE hat die Aufgabe, aus dem Workflowmodell, welches im USE-Objektdiagramm erstellt wurde, ASSL-Kommandos zu generieren, die das Workflowmodell für die Runtime reproduzieren können. Das Plugin realisiert den im Use Case-Diagramm von Abbildung 3.16 mit *save in ASSL file* bezeichneten Anwendungsfall. Anhand der ASSL-basierten Speicherung des Workflowmodells können Instanzen des Workflowmodells zur Runtime erzeugt werden.

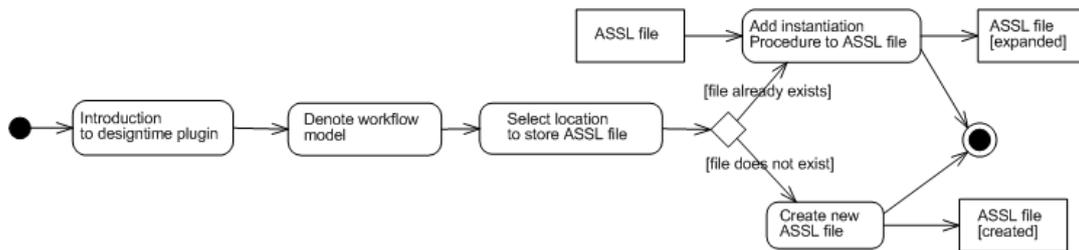
Abbildung 4.2(b) zeigt den Anfangsdialog des Design-time-Plugins und Listing 4.3 einen Auszug einer generierte ASSL-Instanzierungsprozedur. Die Funktionsweise des Plugins und die benötigten Eingaben des Nutzers für das Design-time-Plugin werden im Aktivitätsdiagramm von Abbildung 4.2(a) verdeutlicht.

Zunächst wird das Plugin über USE mit einem Icon in der Menüleiste, welches mit 1. in Abbildung 4.2(b)) markiert ist, aufgerufen. Des Weiteren kann auch das Runtime-Plugin über das daneben stehende Icon, welches mit 2. markiert ist, gestartet werden.

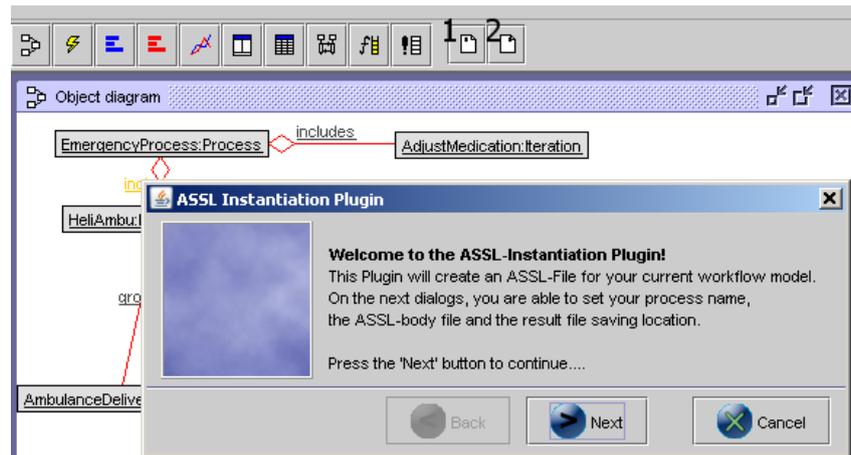
Hat man das Design-time-Plugin aufgerufen, wird das Anfangsdialogfenster geöffnet (s. Abbildung 4.2(b)). Im Hintergrund der Abbildung ist das zu speichernde Workflowmodell im Objektdiagramm zu sehen. Der weitere Ablauf zur Generierung der ASSL-Instanzierungsprozedur ist im Aktivitätsdiagramm von Abbildung 4.2(a) angegeben.

Zunächst muss der Nutzer den Prozess bezeichnen, wonach die Instanzierungsprozedur benannt wird. Beispielsweise ist bei der Generierung der Prozedur von Listing 4.3 die Bezeichnung *EmergencyProcess* gewählt worden. Daraufhin muss der Nutzer eine ASSL-Datei wählen, welche entweder schon existiert oder noch nicht vorhanden ist. Ist die ASSL-Datei schon vorhanden, so wird die neu generierte Prozedur der bestehenden Datei hinzugefügt. Wenn dagegen die ASSL-Datei noch nicht vorhanden ist, so wird eine neue Datei inkl. generierter Instanzierungsprozedur erzeugt.

Das Runtime-Plugin kann ASSL-Dateien entgegennehmen. Diese Dateien stellen eine ASSL-Runtime-



(a) Ablauf des Designtime-Plugin Dialogs



(b) Das Designtime-Plugin in USE aufgerufen

Abbildung 4.2: DMWM-Designtime-Plugin

Environment dar, die die imperativen Prozeduren zur Ausführung der Workflowmodelle beinhaltet, von denen Beispiele in Abschnitt 4.2.2 vorgestellt wurden. Falls Fehler in den ASL-Prozeduren vorhanden sind, so können diese editiert werden und die veränderte ASL-Datei lässt sich zur Laufzeit austauschen. Die dort enthaltenen Instanzierungsprozeduren können des Weiteren aufgerufen werden. Mit dieser Funktionalität lassen sich unterschiedliche Workflowmodelle auch mehrfach instanzieren.

Bei der Erzeugung einer neuen ASL-Datei (s. Aktivität *Create new ASL file* in Abbildung 4.2(a)) fügt das Designtime-Plugin zusätzlich die Standard-Prozeduren zur Workflowausführung der ASL-Datei hinzu. Wenn dagegen in eine bestehende ASL-Datei die generierte Instanzierungsprozedur geschrieben werden soll (s. Aktivität *add instantiation procedure to ASL file*), ist vom Nutzer darauf zu achten, dass die Prozeduren zur Ausführung der Workflowmodelle dort vorhanden sind. Die generierte Instanzierungsprozedur wird der Datei dann hinzugefügt. In Listing 4.3 ist ein Auszug einer Instanzierungsprozedur zu sehen, die aus dem Workflowmodell von Abbildung 3.9 erzeugt wurde.

Mit dem ASL-*Create* Befehl werden Objekte erzeugt. In Zeile 4 wird eine Aktivität erstellt. Daraufhin wird die Bezeichnung der Aktivität in dem Attribut *name* auf die Bezeichnung aus dem ursprünglichen Workflowmodell gesetzt. In Abbildung 3.9 wurden *ObjectIDs* und nicht die *name*-Attribute zur Bezeichnung der Modellelemente genutzt. Das Designtime-Plugin kann mit beiden Arten der Bezeichnung von Modellelementen umgehen. Falls das *name*-Attribut den Wert *Undefined* hat, nimmt das Designtime Plugin die *ObjectID* aus dem Workflowmodell.

In der ASL-Prozedur wird nach der Bezeichnung der Aktivität diese auf den Ausgangszustand *waiting* gesetzt. In Zeile 10 ist außerdem noch zu sehen, dass eine *DeferredChoice*-Gruppe erstellt wird und die beiden vorher erzeugten Aktivitäten über die Aggregation *group* mit dem *DeferredChoice* verbunden werden.

```

1 procedure instantiateEmergencyProcess()
2 var a1:Activity, a2:Activity, d1:DeferredChoice ...;
3 begin
4   a1 := Create(Activity);
5   [a1].name := ['HelicopterDelivery'];
6   [a1].state := [#waiting];
7   a2 := Create(Activity);
8   [a2].name := ['AmbulanceDelivery'];
9   [a2].state := [#waiting];
10  d1 := Create(DeferredChoice);
11  Insert(group, [d1], [a1]);
12  Insert(group, [d1], [a2]);
13  ...
14 end;

```

Listing 4.3: Ausschnitt einer ASSL-Instanzierungsprozedur generiert vom Designtime-Plugin

Die weiteren Elemente aus Abbildung 3.9 werden analog dazu erstellt und sind in Listing 4.3 nicht weiter angegeben.

Die Verbindungen vom Workflowmodell zum Organisationsmodell werden z.Zt. noch nicht vom Designtime Plugin zur Reproduktion in den ASSL-Instanzierungsprozeduren gespeichert. Sie müssen vom Modellierer vor der Ausführung der Workflowinstanzen nochmal hergestellt werden. Hier ist eine Erweiterung der Funktionalität des Designtime Plugins wünschenswert, die die Verbindungen zum Organisationsmodell erfasst und zur Runtime reproduziert.

Mit dem Designtime- und Runtime-Plugin wurden für DMWM zwei Werkzeuge entwickelt, die sehr hilfreich sind. Des Weiteren können weitere Werkzeuge von USE eingesetzt werden, um z.B. die Workflowabläufe zu analysieren. In Abbildung 4.3 ist dafür eine Entwicklungskette als Aktivitätsdiagramm, eine Tool-Kette und die genutzten Artefakte abgebildet, um die Nutzung der Tools und Diagramme in DMWM zu verdeutlichen.

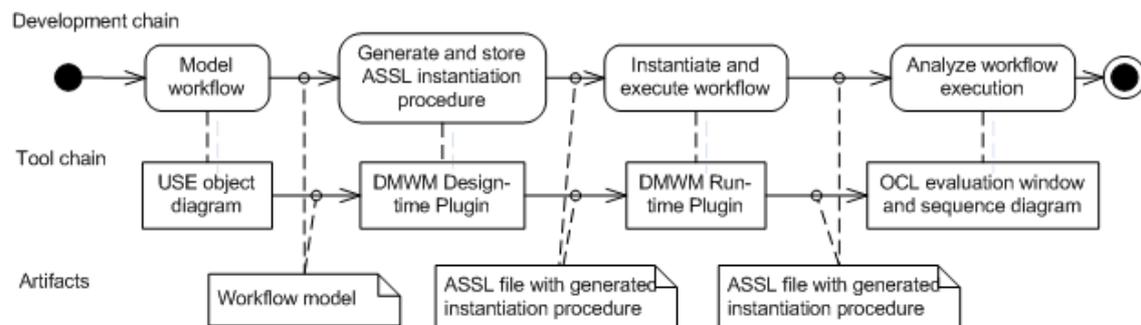


Abbildung 4.3: Übersicht über Entwicklungsprozess von DMWM-Modellen und Nutzung der unterschiedlichen Werkzeuge

Zunächst wird der Workflow mit dem Objektdiagramm modelliert, so wie es in Abschnitt 3.5.3 gezeigt wurde. Daraufhin wird das in diesem Abschnitt beschriebene Designtime-Plugin eingesetzt, um ASSL-Instanzierungsprozeduren zu generieren. Die werden dann wiederum von dem in Abschnitt 4.3 beschriebenen Runtime-Plugin genutzt, um Workflows zu instanzieren. Der Entwickler führt die Modelle anhand des Plugins aus und validiert sie damit. Nach der Ausführung liegen die Workflowinstanzen zusammen mit den Ausführungsdaten im Objektdiagramm vor, die dann zur Analyse genutzt werden können. Hier ist z.B. Processmining mit dem von USE bereitgestellten *OCL evaluation window* möglich. Außerdem loggt USE

die Workflowausführung anhand eines Sequenzdiagramms. Die Analyse von ausgeführten Workflows wird in Abschnitt 4.4 thematisiert.

4.3 Workflow Runtime-Plugin

Das Runtime-Plugin ist mit all seinen bereitgestellten Funktionalitäten notwendig für die Validierung der dynamischen Modelleigenschaften. Die ASSL-Prozeduraufrufe zur Ausführung der Workflowmodelle hätten zwar auch per Hand über die Konsole analog zu denen aus [BGF10] gemacht werden können. Eine solche Bedienung wäre für den Nutzer aber zu umständlich und nicht gut benutzbar. Eine grafische Nutzerschnittstelle wird daher zur praktikablen Validierung der Workflowmodelle benötigt. Auch weitere Funktionalitäten wie die Darstellung der Datenelemente während der Workflow-Ausführung sind notwendig, um diese Aspekte der Workflowmodelle zu testen. Die Funktionalitäten des Runtime-Plugins dienen dazu und werden im Folgenden näher beschrieben.

4.3.1 Sichten auf Prozessinstanz

Ist das USE-Tool mit dem in Abschnitt 3.2 beschriebenen Workflow-Metamodell und dem Runtime-Plugin geladen, so kann dies gestartet werden. Das Workflow-Runtime-Plugin lässt sich anhand des Buttons *WM* in der Menüleiste von Abbildung 3.17 oder mit dem in Abbildung 4.2(b) mit „2“ referenzierten Button aufrufen. Die unterschiedliche Darstellungsweise der Buttons in den beiden Abbildungen ist unterschiedlichen Versionen von USE geschuldet. In Abbildung 4.4 sieht man das Runtime-Plugin mit verschiedenen Sichten auf die aktuelle Prozessinstanz. Diese sind Instanzen des Workflowmodells von Abbildung 3.9.

Das Plugin stellt drei verschiedene Sichten auf die Prozessinstanz bereit, die der Nutzer im Optionsauswahlmenü von Abbildung 4.4(d) auswählen kann. Das Menü kann mit einem Rechtsklick auf das Runtime-Plugin aufgerufen werden. Dieser Aufruf ist beim USE-Tool auch bei den anderen Views so möglich [Dat07]. Die verschiedenen Sichten auf die Workflowinstanz sind mit *Worklist View*, *Extended View* und *Structured View* angegeben.

In Abbildung 4.4(a) ist die Darstellung des *Extended view* angegeben. Alle zugehörigen Aktivitäten dieser Prozessinstanzen werden als Kinder im Baum hinzugefügt. Es sind dort alle Aktivitäten von der Prozessinstanz von Abbildung 3.9 zu sehen. Die Aktivitäten befinden sich alle im Zustand *waiting* und sind damit grün dargestellt. Die Zuordnung der Ausführungszustände zu den Farben ist in Tabelle 4.3 angegeben. Das Plugin unterscheidet den Zustand *waiting* in startbare und nicht startbare Aktivitäten. Hier wird für die *start()*-Operation für jede Aktivität im voraus geprüft, ob keine OCL-Constraints verletzt werden und die Aktivität damit startbar ist.

Farbe	Zustand
Grün (mit schwarzer Schrift)	Waiting (nicht startbar)
Hellgrün (mit grüner Schrift)	Waiting (startbar)
Blau	Running
Grau	Skipped
Schwarz	Done
Rot	Failed
Gelb	Canceled
Magenta	Undefined

Tabelle 4.3: Die Farbzusordnungen zu den Zuständen



(a) Extended View

(b) Worklist View

(c) Structured View

(d) Optionsauswahlmenü für Runtime-Plugin

Abbildung 4.4: Verschiedene Sichten auf Workflow im Runtime-Plugin und Optionsauswahlmenü

Die *enabled*-Prüfung ist nicht nur für die Farbdarstellung der Aktivitäten im Baum notwendig, sondern auch für die Buttons, die in Abbildung 4.4(a) unten im Fenster zu sehen sind. Diese Buttons repräsentieren die Schnittstelle der Aktivitäten, die im Workflow-Metamodell von Abbildung 3.2 als Operationen modelliert sind. Der Nutzer kann damit Aktivitäten starten, beenden, überspringen oder abbrechen.

Es kann eine *enabled*-Prüfung für alle Operationen im Workflow-Metamodell an- bzw. wieder abgestellt werden. Hierfür dient das Untermenü *enabled check for...* vom Optionsauswahlmenü, das in Abbildung 4.4(d) zu sehen ist. Für die Operationen, für die die *enabled*-Prüfung angestellt ist, werden die Operationen im voraus getestet, ob die Ausführung möglich ist oder ob die Ausführung OCL-Bedingungen verletzt. Diese Probeausführung hat noch keine Effekte auf die Workflowinstanz. Die Buttons werden je nach Ergebnis des Tests für den Nutzer aktiviert oder deaktiviert. Im Falle von Abbildung 4.4(a) ist die Aktivität *AmbulanceDelivery* ausgewählt. Die Aktivität wird hellgrün dargestellt, was zeigt, dass sie startbar ist. Dieses zeigt sich auch am aktivierten *start* Button. Alle weiteren Buttons sind deaktiviert und können damit vom Nutzer für diese Aktivität nicht aufgerufen werden.

Für Abbildung 4.4(b) wurde die *Worklist View* ausgewählt. Hier sind ausschließlich die startbaren und laufenden Aktivitäten dargestellt. Alle beendeten oder abgebrochenen Aktivitäten werden herausgefiltert und nicht dargestellt. Des Weiteren ist dort eine weitere Workflowinstanz mit dem *instantiateEmergencyProcess* aus dem Optionsauswahlmenü von Abbildung 4.4(d) erzeugt worden. Das dafür erstellte Prozessobjekt und die dazugehörigen Aktivitäten haben andere Objekt-IDs erhalten als die für die erste Workflowinstanz, in

dem die Objekt-IDs die gleiche wie die Bezeichnung der Aktivitäten ist. Die Objekt-IDs sind in Klammern hinter der Aktivitätsbezeichnung in den drei Sichten von Abbildung 4.4 zu sehen.

Die dritte und letzte Sicht auf Workflowinstanzen ist mit *Structured View* bezeichnet und wird in Abbildung 4.4(c) verwendet. Dort sind zusätzlich zu den Aktivitäten Gruppen von Aktivitäten zu sehen. Die Gruppen sind vom Workflow-Metamodell von Abbildung 3.2 bekannt. Im Workflowmodell von Abbildung 3.9 werden vier Gruppen verwendet, die während der Ausführung vom Runtime-Plugin in Abbildung 4.4(c) dargestellt werden. Hier kann es vorkommen, dass Aktivitäten mehreren Gruppen zugeordnet sind und somit auch mehrfach in dieser Sicht angezeigt werden.

Neben der Darstellungsauswahl der Workflowinstanz stellt das Operationsauswahlmenü von Abbildung 4.4(d) noch weitere Funktionalitäten bereit. Als erster Eintrag ist mit *Show Log* die Möglichkeit gegeben, ein Log-Fenster aufzurufen, in der die bisher getätigten Interaktionen im Runtime-Plugin angezeigt werden. In Abschnitt 4.4.1 wird dieses Fenster näher vorgestellt. Die vier darauf folgenden Einträge im Optionsauswahlmenü betreffen die Darstellung der Zeit. *Elapsed Time* und *Elapsed Seconds* zeigen die abgelaufene Zeit nach dem Aufrufen des Runtime Plugins an. *Real Time* gibt die aktuelle Systemzeit und *Ticks* geben die Anzahl der Aktionen des Nutzers an. Bei der Workflowsausführung werden in den *start* und *finish* Attributen der Aktivitäten (s. Abbildung 3.2) die *Elapsed Seconds* eingetragen.

Mit *Run ASSL Procedure* gibt es die Möglichkeit, eine beliebige ASSL Prozedur aus der aktuell ausgewählten ASSL-Runtime-Datei aufzurufen. Dagegen wird mit *Peek ASSL Procedure* nur getestet, ob die entsprechende ASSL Prozedur ohne OCL-Constraint-Verletzungen ausführbar ist. Tritt eine Verletzung auf, wird diese angezeigt. Mit dieser Funktionalität kann z.B. geprüft werden, warum der *start*- oder *finish* Button für eine bestimmte Aktivität nicht aktiviert ist.

Im Optionsauswahlmenü folgt die Aktivierung bzw. Deaktivierung der *enabled* Prüfungen für bestimmte Operationen. Für große Prozesse ist es evtl. ratsam, bestimmte Checks zu deaktivieren, da diese recht rechenaufwändig und zeitintensiv sind. Mit *Instantiate...* folgt die Möglichkeit, aus der ASSL-Runtime-Datei Prozesse zu instanziiieren. Zudem kann mit *Load ASSL-Runtime* die ASSL-Runtime-Datei gewechselt werden. Dieser Wechsel wird wie auch alle anderen Einstellungen am Optionsauswahlmenü persistent vorgenommen. D.h. beim nächsten Programmstart bleiben die Änderungen erhalten. Es besteht mit *Revert to default ASSL-Runtime* immer die Möglichkeit, zur ursprünglichen ASSL-Runtime-Datei zurückzukehren und die vorgenommenen Änderungen rückgängig zu machen. Als letztes kann der Nutzer die Datensicht während der Workflow-Ausführung an- bzw. wieder abstellen. Die Datensicht beim Runtime-Plugin wird Thema in Abschnitt 4.3.3 sein.

4.3.2 Ausführung spezieller Aktivitäten

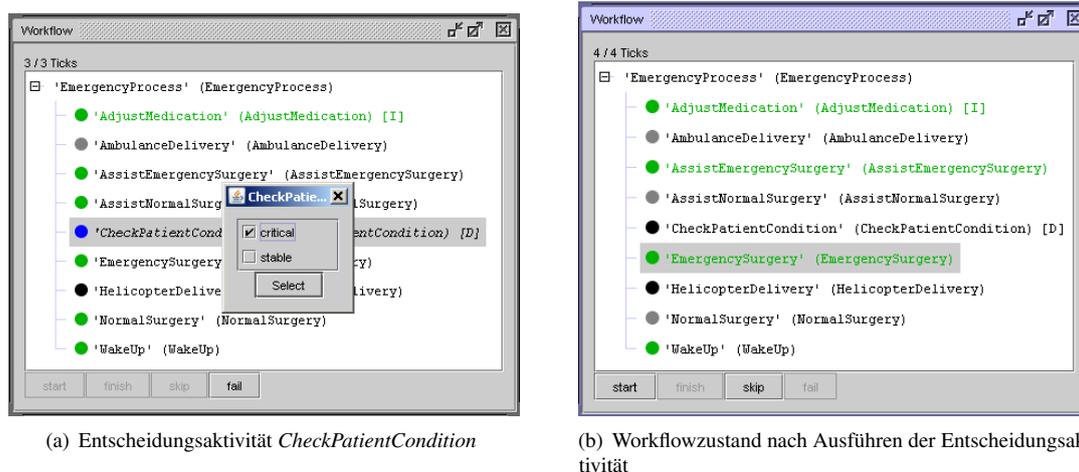
Im Workflow-Metamodell von Abbildung 3.2 ist ersichtlich, dass spezielle Aktivitäten als Unterklassen von *Activity* existieren, die unterschiedliches Verhalten für den Nutzer darstellen. Die unterschiedliche Darstellung wird vom Runtime-Plugin und das unterschiedliche Verhalten mit den ASSL-Prozeduren, die exemplarisch in Abschnitt 4.2.2 vorgestellt wurden, realisiert. Dieses betrifft die im Folgenden aufgelisteten *Decision*, *Iteration* und Unterklassen von *MultiInstance*.

4.3.2.1 Entscheidungsaktivitäten

Die Semantik der Entscheidungsmodellierung bei DMWM ist ähnlich zu der, die bei EPKs verwendet wird und in Abschnitt 2.5.2.1 beleuchtet wurde. Bei dem Ansatz wird angenommen, dass der Nutzer die Entscheidung während der Bearbeitung der Entscheidungsaktivität machen muss. Die Aktivität, die zur

Entscheidung führt, wird mit der Aktivität bezeichnet und die Kriterien zur Pfadauswahl wird bei DMWM in den zugeordneten *Guards* angegeben. Während der Ausführung der Aktivität werden dem Nutzer die *Guards* zur Auswahl dargestellt. Dieser hat dann die Aufgabe, den zutreffenden Pfad zu wählen.

Im Folgenden wird das Konzept an einem Beispiel erläutert. In Abbildung 4.5(a) wird die Aktivität *CheckPatientCondition* zur Runtime dargestellt. Während der Ausführung der Aktivität hat die ausführende Person die Verfassung des Patienten zu prüfen. Es stehen laut Workflowmodell von Abbildung 3.9 hier zwei Alternativen bereit. Entweder ist der Patient in einer stabilen oder in einer kritischen Verfassung. Diese Alternativen werden dem Nutzer zur Runtime präsentiert und eine davon hat er per Checkbox auszuwählen.



(a) Entscheidungsaktivität *CheckPatientCondition*

(b) Workflowzustand nach Ausführen der Entscheidungsaktivität

Abbildung 4.5: Anzeigen der Entscheidungsaktivitäten während der Runtime

Nach Auswahl und Beendigung der Entscheidungsaktivität werden die nicht ausgewählten Aktivitäten mit *skip()* übersprungen. Dieses ist in der ASSL-Datei für die *finish()*-Operation der Klasse *Decision* umgesetzt. Aktivitätsketten, die mit *seq*-Links entstehen können, werden bis zum nächsten *merge*-Operator übersprungen, um das *Structured Synchronization Merge Pattern* (WCP7) umzusetzen, welches in Abschnitt 3.2.4.2 vorgestellt wurde. In dem Falle, dass die *seq*-Kette ohne *FlowOperator* endet, werden die Aktivitäten bis dahin übersprungen. Im Notfallprozess-Beispiel von Abbildung 3.9 sind solche Ketten nicht vorhanden.

In Abbildung 4.5(b) ist die Workflowinstanz nach Beendigung der Entscheidungsaktivität zu sehen. Da der Guard *critical* in Abbildung 4.5(a) ausgewählt wurde, wurde die Aktivität *NormalSurgery* übersprungen. Da diese Aktivität wiederum in einer parallelen Beziehung zur Aktivität *AssistNormalSurgery* steht, wurde damit auch diese übersprungen. Übersprungene Aktivitäten werden grau (als *skipped*) gekennzeichnet. Das Verhalten ist in einem UML-Sequenzdiagramm in Abschnitt 4.4.2 angegeben.

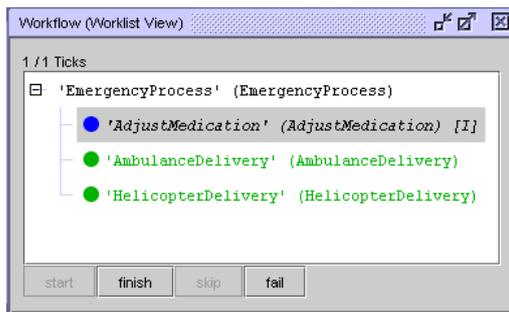
In Abbildung 4.5(b) ist des Weiteren zu sehen, dass die Aktivitäten *EmergencySurgery* und *AssistEmergencySurgery* startbar geworden sind, was mit hellgrün für *enabled* gekennzeichnet wurde. Diese Konsequenz ist auch an dem aktivierten *start*-Button zu sehen, der die ausgewählte *EmergencySurgery* Aktivität betrifft.

4.3.2.2 Iterationen

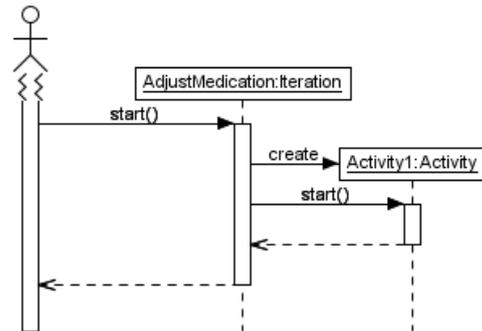
Iterationen stellen ein anderes Verhalten dar als normale Aktivitäten, weil sie einen anderen Objekt-Lebenszyklus aufweisen. Normale Aktivitäten haben den in Abbildung 3.3(a) dargestellten Lebenszyklus, wohingegen *Iteration*-Aktivitäten den von Abbildung 3.3(b) besitzen. Die Darstellung von Iterationsaktivitäten im Runtime-Plugin ist mit einem *[I]* gekennzeichnet, welches in Abbildung 4.6(a) an der gestarteten

Aktivität *AdjustMedication* zu sehen ist.

Wird die Iterationsaktivität gestartet, so wird eine neue *Activity*-Instanz erzeugt und über einen Link der Assoziation *iteration* verbunden. Die Erzeugung und das Verhalten des *start()*-Aufrufs ist im Sequenzdiagramm in Abbildung 4.6(b) zu sehen. Der *start()*-Aufruf hat eine Zustandsänderung der Iterationsaktivität zufolge und zudem wird der Aufruf an das neu erzeugte Objekt, das die Ausführungsinstanz repräsentiert, weiterdelegiert. Dieses Aktivitätsobjekt hat den Lebenszyklus einer normalen Aktivität von Abbildung 3.3(a). Der *finish()* oder *fail()*-Aufruf wird daraufhin ebenfalls weitergeleitet.



(a) Darstellung der Iterationsaktivität *AdjustMedication* im Runtime Plugin



(b) Starten eines Iterationszykluses im Sequenzdiagramm dargestellt

Abbildung 4.6: Ausführung der Iterationaktivität

4.3.2.3 Multiinstanzaktivitäten

In diesem Abschnitt werden exemplarisch zwei der vier Multiinstanz-Aktivitäten des Workflow-Metamodells vorgestellt. Dafür und für die Elemente, die in Abschnitt 4.3.2.4 vorgestellt werden, ist ein neues Workflowmodell notwendig, welches in Abbildung 4.7(a) angegeben ist. Dort wird ein weiterer Notfallprozess gezeigt, der die Multiinstanz-Aktivität *NotifyAffiliated* beinhaltet. In der Aktivität sollen die Angehörigen benachrichtigt werden. Die Reihenfolge der Benachrichtigung ist nicht spezifiziert, so dass ihnen auch parallel Bescheid gegeben werden kann. Zudem können, während die Multiinstanz-Aktivität läuft, noch weitere parallel laufende Instanzen der Aktivität erzeugt und gestartet werden, womit zusätzliche Angehörige zu benachrichtigen sind. Damit wird das Workflow Pattern 15 (s. Abschnitt 3.2.4.3) ausgedrückt.

Der Notfallprozess von Abbildung 4.7(a) umfasst des Weiteren eine Iterationsgruppe *Exploration* zur Untersuchung des Patienten und eine Operationsaktivität *Surgery*, die in einer sequenziellen temporalen Beziehung stehen. D.h. die Untersuchung ist vor der Operation durchzuführen. Zudem ist die Gruppe *Exploration* eine *IterationGroup*, so dass die enthaltenen Aktivitäten nochmal wiederholt ausgeführt werden können. Die Gruppe umfasst die Multiinstanz-Aktivität *DoubleDiagnosis*, in der zwei Diagnosen unabhängig voneinander durchgeführt werden sollen. Daraufhin sollen Vorerkrankungen in der Aktivität *CheckPreExistingIllnesses* festgestellt werden. Sind die Ergebnisse nicht zufriedenstellend, so können die enthaltenen Aktivitäten durch einen *nextIteration()*-Aufruf beim *Exploration*-Objekt nochmal durchgeführt werden. Ein weiteres Beispiel dafür wird in Abschnitt 4.3.2.4 gegeben.

In Abbildung 4.7(b) ist das Workflowmodell im Runtime-Plugin dargestellt. Es ist zu sehen, dass die beiden Multiinstanz-Aktivitäten startbar sind. In Abbildung 4.7(c) wird daraufhin die Aktivität *NotifyAffiliated* gestartet. Es wird damit die Anzahl der zu startenden Instanzen abgefragt, woraufhin im angegebenen Beispiel 2 eingetragen wird. Unten rechts in der Abbildung ist zusätzlich zu den üblichen Buttons zur



Abbildung 4.7: Workflow-Modell mit Multiinstanz-Aktivitäten und Ausführung im Runtime-Plugin

Interaktion mit Aktivitäten auch ein *add*-Button zu sehen. Dieser repräsentiert die gleichnamige Operation der Klasse *MIRuntimeWithAdd* vom Workflow-Metamodell aus Abbildung 3.2. Es können damit während die Multiinstanz-Aktivität läuft Ausführungsinstanzen hinzugefügt werden.

In Abbildung 4.7(d) sind daraufhin die zwei Ausführungsinstanzen als Kinder in den Baum hinzugefügt worden. Diese können unabhängig voneinander ausgeführt werden. Die Multiinstanzaktivität *NotifyAffiliated* kann jedoch erst dann beendet werden, wenn alle Ausführungsinstanzen nicht mehr laufen. Möchte man hier ein unsynchrones Verhalten erlauben, sollte eine Aktivität der Klasse *MIWithoutSynch* verwendet werden, die das Workflow Pattern 12 ausdrückt (s. Abschnitt 3.2.4.3).

4.3.2.4 Darstellung und Ausführung eines Cancel-Prozesses und einer Iterationsgruppe

In Abbildung 4.7(a) ist bereits ein Cancel-Prozess modelliert. Workflowinstanzen davon können jeder Zeit mit der *cancel()*-Operation abgebrochen werden. Im Runtime-Plugin ist das Prozessobjekt und der *cancel*-Button in Abbildung 4.7(b) zu sehen, mit dem der Nutzer die Operation ausführen kann.

Im Workflowmodell von Abbildung 4.7(a) ist außerdem noch ein Objekt vom Typ *IterationGroup* angegeben. Die Benutzung von Iterationsgruppen im Runtime-Plugin wird in Abbildung 4.8 angegeben.

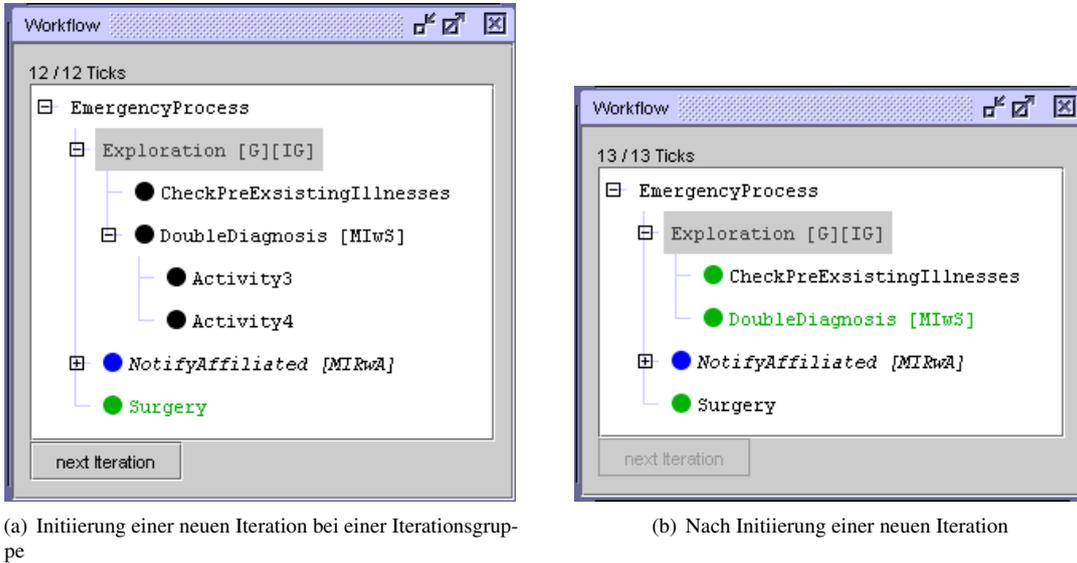


Abbildung 4.8: Ausführung der Iterationsgruppe

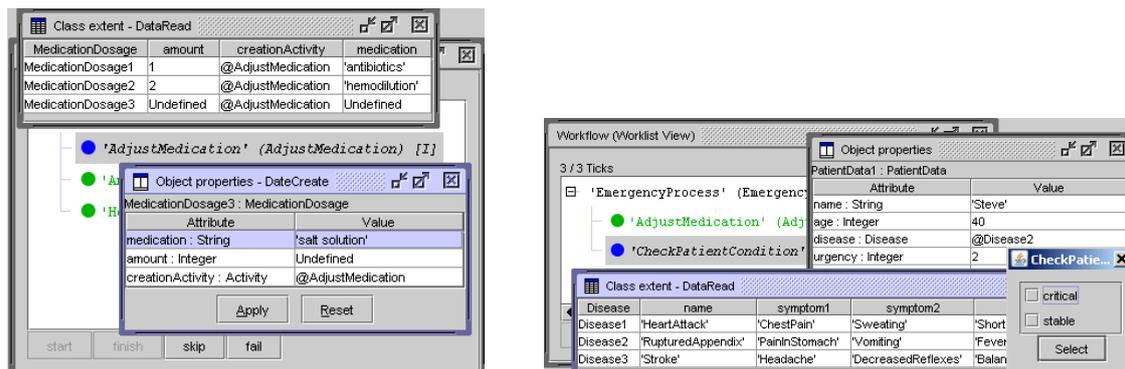
Zunächst ist in der Abbildung 4.8(a) ein durchgeführter Iterationszyklus der Gruppe *Exploration* zu sehen, in dem die Aktivitäten *DoubleDiagnostics* und *CheckPreExistingIllnesses* durchgeführt sind. Die Aktivität *Surgery* ist aktiviert und kann damit durchgeführt werden. Es kann aber auch eine weitere Iteration von *Exploration* gestartet werden, indem der Button *nextIteration* gedrückt wird und damit die enthaltenen Aktivitäten neu durchgeführt werden müssen. Das ist in Abbildung 4.8(b) geschehen. Dort ist zu sehen, dass die Aktivität *Surgery* nicht weiter aktiviert ist, da die in der Gruppe *Exploration* enthaltenen Aktivitäten vorher noch einmal ausgeführt werden müssen.

Der vorhergehenden Iterationszyklus mit den zugehörigen Aktivitätsinstanzen zusammen mit den Zeitstempeln bleiben im Objektdiagramm vorhanden und werden mit einem *archive*-Link verbunden. Im Runtime-Plugin wird diese Information nicht dargestellt, da sie für die aktuelle Ausführung i.d.R. keine Relevanz hat.

4.3.3 Anzeige der Datenabhängigkeiten

Im Workflowmodell von Abbildung 3.13(b) sind die Datenabhängigkeiten von diversen Aktivitäten angegeben. In diesem Abschnitt wird mit Abbildung 4.9 die Darstellung dieser Datenverbindungen zur Runtime näher betrachtet. Zur Veranschaulichung dient das Anwendungsszenario aus Abschnitt 4.3.1 und das Workflowmodell aus Abschnitt 3.3.3.

Die Aktivität *AdjustMedication* soll die Verabreichung von Medikamenten und ihre Dosierung dokumentieren und wurde in Abbildung 4.9(a) zur Ausführung ausgewählt. Das *DataRead*-Objekt *pastDosages* vom Workflowmodell von Abbildung 3.9 wird vom Runtime-Plugin ausgewertet und in der *DataRead*-Tabelle in der Abbildung oben angezeigt. Es ist zu sehen, dass zwei vorhergehende Medikamente verabreicht wurden und damit die Iterationsaktivität *MedicationDosage* bereits zweimal ausgeführt wurde. Die aktuelle Medikation ist zwar schon in der Tabelle aufgelistet, die Daten dafür sind aber noch einzutragen. Dafür wird das *DataCreate*-Fenster darunter eingeblendet, das vom Runtime-Plugin mit der Interpretation des



(a) Darstellung der Datenabhängigkeiten von *AdjustMedication* im Runtime Plugin

(b) Darstellung der Datenabhängigkeiten von *CheckPatientCondition*

Abbildung 4.9: Darstellung von Daten während der Workflowausführung

DataCreate-Objekts *dosage* angezeigt wird. Das Attribut *creationActivity* vom neu erzeugten Datenobjekt *MedicationDosage3* ist bereits vom Runtime Plugin auf die erzeugende Aktivität *AdjustMedication* gesetzt worden. Während der Abarbeitung der Aktivität hat der Nutzer die Aufgabe, die Attribute *medication* und *amount* einzutragen.

In Abbildung 4.9(a) ist des Weiteren zu sehen, dass die aktuell ausgewählte Aktivität *AdjustMedication* nicht beendet werden kann, da der entsprechende *finish*-Button deaktiviert ist. Um den Grund dafür herauszufinden, kann der Nutzer *peek ASSL procedure* aus dem Optionsmenü des Runtime-Plugins (s. Abbildung 4.4(d)) aufrufen. Der ASSL-Prozeduraufruf *finish(AdjustMedication)* versucht diese Prozedur auszuführen und liefert als Ergebnis, dass diese entweder erfolgreich ausgeführt werden kann oder eine Constraint-Verletzung hervorgerufen wird. In Listing 4.4 ist die Konsolenausgabe des *peek*-Aufrufs der *finish()*-Operation von Abbildung 4.9(a) zu sehen, die angibt, dass die Invariante *MedicationEntriesMade* verletzt wurde. Mit dieser Information sieht der Nutzer, dass die Einträge für die aktuelle Medikation noch fehlen. Erst nachdem diese getätigt wurden, kann die Aktivität beendet werden.

```

1 checks      valid      invalid      mul. viol.      time (ms)      Invariant
2 ...         ...         ...         ...         ...         ...
3 1           1           0           0           0           MIWithSync::instanceDoneSelfDone
4 1           0           1           0           0           MedicationDosage::MedicationEntriesMade
5 0           0           0           0           0           MultiInstance::instanceCreated
6 ...         ...         ...         ...         ...         ...
7 Total checks: 12 Overhead (checks - states checked): 11
8 Sorted 0 times.

```

Listing 4.4: Konsolenausgabe bei Peek-Aufruf der *finish()* Operation bei *AdjustMedication*

In Abbildung 4.9(b) ist des Weiteren die Aktivität *CheckPatientCondition* ausgewählt worden. Deren Datenabhängigkeiten *getAllDiseases* und *PatientData* des Workflowmodells von Abbildung 3.13(b) werden vom Runtime-Plugin dort angezeigt. Die *DataRead*-Tabelle repräsentiert die Daten des Informationssystems, die über das *getAllDiseases* Objekt angefragt wurden. Des Weiteren ist das *PatientData*-Objekt im *Object properties*-Fenster zu sehen. Da *CheckPatientCondition* eine Entscheidungsaktivität ist, werden dem Nutzer zudem die zugehörigen Entscheidungsalternativen analog zur Abbildung 4.5(a) zur Auswahl gestellt.

4.3.4 Adaptive Aspekte zur Workflow Modellierung

Sowohl die Workflowmodelle zur Design-time als auch die Workflowinstanzen zur Runtime werden in einem UML-Objektdiagramm repräsentiert (s. Tabelle 4.2). Damit ergibt sich die Möglichkeit, Workflowinstanzen zur Runtime im Objektdiagramm von USE zu editieren. Modellierungselemente können wie zur Design-time hinzugefügt, wieder gelöscht oder Attribute verändert werden.

Das hier vorgestellte Runtime-Plugin reagiert unmittelbar auf Änderungen im Objektdiagramm und aktualisiert die Darstellung der Workflowinstanz entsprechend. Erzeugt der Nutzer beispielsweise eine neue Aktivität im Prozessmodell und fügt sie einem Prozess hinzu, so taucht sie dann im Runtime-Plugin mit auf. Wurde die Aktivität noch nicht auf den initialen Zustand *waiting* gesetzt wird die neu hinzugefügte Aktivität entsprechend der Tabelle 4.3 mit der auffälligen Farbe Magenta hervorgehoben. Ein Korrigieren des Zustandes durch den Nutzer ist dann noch möglich.

Die Adaptivität bringt Vorteile, indem das Workflowmodell zur Design-time nicht komplett spezifiziert sein muss. Elemente können zur Runtime hinzugefügt und die Workflowmodelle an nicht bedachte Situationen angepasst werden. Es bedarf keines Transformationsschrittes für die Ausführbarkeit der Prozessmodelle. Zur Runtime steht die volle Funktionalität des Modellierungstools zur Verfügung und es bestehen keine Einschränkungen der Modellierung. Es sind hierbei jedoch ein paar Dinge zu beachten, um eine konsistente Ausführung der Modelle zu gewährleisten. Bestimmte Teile der Prozessinstanz sollten geschützt und nicht editiert werden. Als Richtlinie, um diese Probleme zu umgehen, kann man folgende Grundsätze heranziehen:

- Es sollten keine Aktivitäten aus dem Workflowmodell entfernt werden. Falls eine Aktivität nicht auszuführen ist, sollte sie über das Runtime-Plugin mit *skip()* übersprungen werden.
- Zustandsänderungen und allgemeine Attributänderungen sollten bei Aktivitäten nur über das Runtime-Plugin geschehen.
- Beim Hinzufügen von Aktivitäten sind diese vom Nutzer auf den Ausgangszustand *waiting* zu setzen.
- Beim Hinzufügen von temporalen Beziehungen ist darauf zu achten, dass danach keine OCL-Invariante verletzt ist. Sollte dieses der Fall sein, ist eine weitere Ausführung des Prozessmodells im Runtime Plugin nicht möglich.

4.4 Analyse ausgeführter Prozesse und Process Mining

In UML sind diverse Diagramme vorhanden, um Abläufe bzw. Szenarios zu beschreiben. Bei DMWM wird das Sequenzdiagramm dafür genutzt, um durchgeführte Workflowabläufe festzuhalten. Mit OCL stehen mächtige Möglichkeiten bereit, um Mininganfragen zu stellen. Die Prozessinstanzen sind zwar auch im UML-Objektdiagramm vorhanden, die die relevanten Ausführungsdaten beinhalten, diese Diagramme sind aber nicht gut dafür geeignet, spezifische Ausführungsdaten zu ergründen. Mit dem Runtime-Plugin für DMWM ist ein Protokollfenster bereitgestellt worden, das die Nutzerinteraktion protokolliert.

4.4.1 Ausführungsprotokoll und Analyse im Runtime-Plugin

Das Runtime-Plugin stellt ein Ausführungsprotokoll bereit, das die bisher getätigten ASSL-Operationsaufrufe zusammen mit der zugehörigen Aktivität und Zeitstempel auflistet. In Abbildung 4.10 ist dieses Fenster zu sehen.

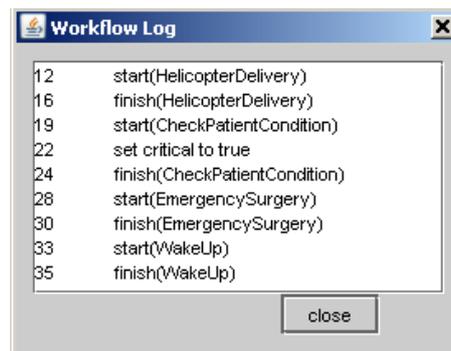


Abbildung 4.10: Das Log-Fenster vom Runtime-Plugin

In dem Protokoll werden ausschließlich die vom Nutzer getätigten Aufrufe protokolliert. Seiteneffekte, die durch temporale Verbindungen zu anderen Aktivitäten auftreten, werden nicht berücksichtigt. Bei dem in Abbildung 4.10 protokollierten Szenario wurde zunächst die Aktivität *HelicopterDelivery* nach 12 Sekunden gestartet. Die Zeit läuft nach dem erstmaligen Starten des Runtime-Plugins los. Das indirekte Überspringen der *AmbulanceDelivery* Aktivität wird hier nicht, dafür aber im Sequenzdiagramm von Abbildung 4.11 mit dem *skip()*-Aufruf dargestellt. Daraufhin wurde die gestartete Aktivität nach 16 Sekunden wieder beendet. Als nächstes wurde *CheckPatientCondition* gestartet, der Guard *critical* gesetzt und die Aktivität wieder beendet. Die Aktivitäten *EmergencySurgery* und *WakeUp* sind daraufhin durchgeführt worden, womit das hier betrachtete Szenario geendet hat.

4.4.2 UML-Sequenzdiagramme

In USE werden die Operationsaufrufe nicht nur mit dem Log-Fenster von Abbildung 4.10 sondern auch mit einem UML-Sequenzdiagramm protokolliert. In Abbildung 4.11 ist ein Szenario vom Workflowmodell von Abbildung 3.9 durchgeführt worden. Es wird hier das gleiche Szenario wie in Abschnitt 4.4.1 betrachtet. Im Gegensatz zum Log-Fenster werden im Sequenzdiagramm auch die vom Nutzer nicht direkt aufgerufenen Operationen aufgeführt. Die direkt aufgerufenen Kommandos gehen, so wie es bei Sequenzdiagrammen üblich ist, vom UML-Strichmännchen aus.

Die Ausführung des Prozesses von Abbildung 4.11 ist wie folgt. Das Szenario beginnt mit dem Transport des Patienten durch das Starten der Aktivität *HelicopterDelivery*. Diese steht in der *DeferredChoice*-Beziehung zu *AmbulanceDelivery* (s. Prozessmodell von Abbildung 3.9), die damit geskippt wurde. Während der Durchführung der Aktivität *CheckPatientCondition*, hat der Nutzer den Guard *critical* ausgewählt, so wie es im Protokoll von Abbildung 4.10 zu sehen ist. Beim Beenden der Aktivität wird der nicht ausgewählte Zweig und damit *NormalSurgery* geskippt. Diese Aktivität steht wiederum in einer *Parallel* Beziehung zu *AssistNormalSurgery*, die damit ebenfalls geskippt wurde.

Die parallele Ausführung von *EmergencySurgery* und *AssistEmergencySurgery* ist daraufhin mit den entsprechenden *start()* und *finish()*-Aufrufen im Sequenzdiagramm abgebildet. Im Szenario ist als letztes dann die Durchführung der *WakeUp*-Aktivität zu sehen.

4.4.3 Process Mining mit OCL

Process Mining ist ein Verfahren, um bestimmte Eigenschaften bei abgearbeiteten Prozessen herauszufinden [Aal11]. Die Daten werden von Workflow- bzw. Informationssystemen aus real abgelaufenen Ge-

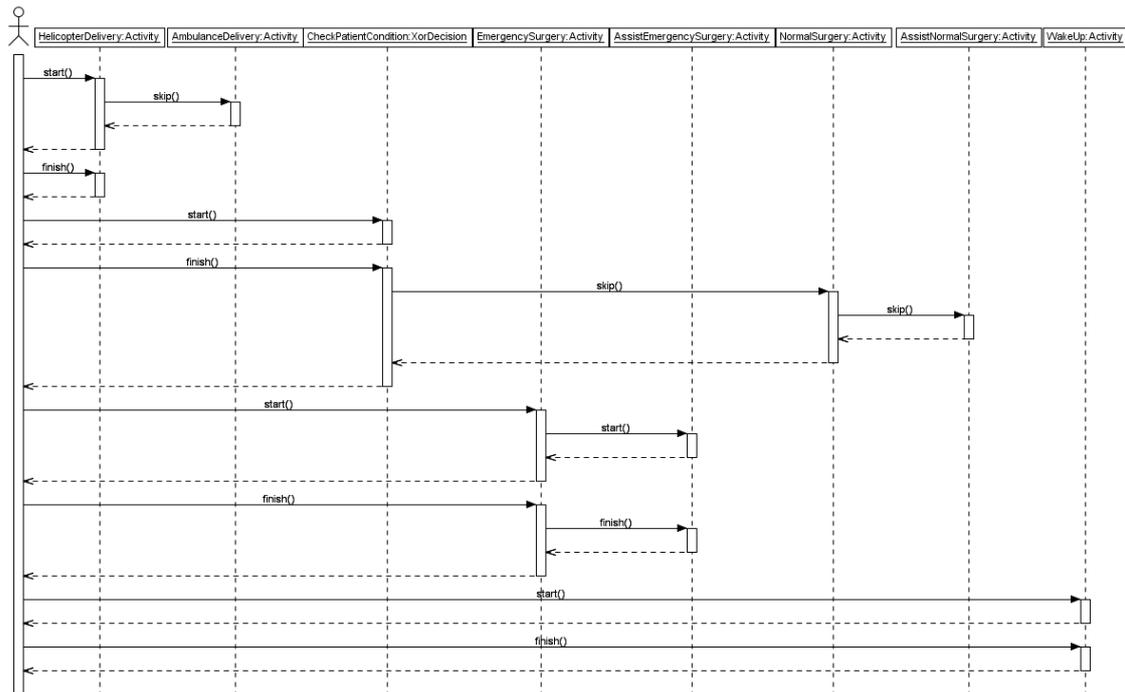


Abbildung 4.11: Ein Szenario des Notfallprozesses in einem Sequenzdiagramm dargestellt

schäftsprozessen erzeugt und in Form von Event-Logs gespeichert. Daraufhin können diese in das MXML-Austauschformat [DA05] oder XES [AAM⁺11] transformiert werden, so dass abgearbeitete Prozessinstanzen ausgehend von Workflow- oder Informationssystemen zur Analyse- bzw. Miningsoftware übertragen werden können. Es wird derzeit an einem weiteren Plugin für USE gearbeitet, das real durchgeführte Prozesse über das MXML-Format importieren kann.

Für MXML wurde ein Metamodell entwickelt [DA05], was für eine Erweiterung des DMWM-Metamodells dienen soll. Insbesondere ist die Aufnahme von Events im DMWM-Metamodell vorgesehen. Auf dieser Basis können die MXML-Daten vom Event-Log in das UML-Objektdiagramm von USE übertragen werden. Hat man die Daten im UML-Objektdiagramm, können die abgearbeiteten Prozesse über OCL-Anfragen wie im Folgenden betrachtet, analysiert werden.

Für die Mininganfragen können OCL-Hilfsfunktionen aus dem Workflow-Metamodell, wie z.B. die *getActivities()*-Operation, die in Abschnitt 3.2.3.2 vorgestellt wurde, eingesetzt werden. Diese Operation wird beispielsweise auch in der OCL-Anfrage von Listing 4.5 eingesetzt. Mit ihr sollen die Aktivitäten und zugehörigen Ausführungsdauern sortiert ausgegeben werden. Die Aktivität, die am kürzesten gebraucht hat, wird in der Sequenz, die zurückgegeben wird, zuerst ausgegeben. Darauf folgen die Aktivitäten mit einer aufsteigenden Ausführungsdauer.

Die Anfrage in der ersten Zeile von Listing 4.5 wurde in der Konsole von USE aufgerufen. In der zweiten Zeile ist dann das Ergebnis zu sehen, das USE nach Interpretation der OCL-Anfrage geliefert hat. Die Anfrage in der Konsole hätte genauso im *OCL Evaluation Window* über die GUI gestellt werden können. Diese Möglichkeit ist in der Toolchain von Abbildung 4.3 angegeben.

Zum Process-Mining gehören insbesondere auch Verfahren, um aus den abgearbeiteten Prozessinstanzdaten Prozessmodelle wiederzugewinnen [Aal11]. Hierfür wurde im Zusammenhang mit dem ProM-

```

1 use> ?EmergencyProcess.getActivities()->iterate(a:Activity;acc:Bag(Tuple(n:String, t:Integer))=Bag{}
  |acc->including(Tuple{n:a.name, t:a.finish-a.start}))->select(t.isDefined())->sortedBy(t)
2 -> Sequence{Tuple{n='AssistEmergencySurgery', t=2}, Tuple{n='EmergencySurgery', t=2}, Tuple{n='
  HelicopterDelivery', t=3}, Tuple{n='WakeUp', t=4}, Tuple{n='CheckPatientCondition', t=7}} : Sequence(
  Tuple(n:String, t:Integer))

```

Listing 4.5: Konsolenausgabe für OCL-Mininganfrage

Tool [ADG⁺09] u.a. der α -Algorithmus entwickelt, der aus einer Liste von Events ein Petri-Netz als Prozessmodell generieren kann [DMW09].

Die Möglichkeit, aus den Abarbeitungsdaten ein Prozessmodell zu entwickeln, ist mit DMWM und USE ebenfalls denkbar. Die Invarianten, die die operationale Semantik für die Runtime festlegen und auf den Ausführungszuständen der Aktivitäten basieren (s. Abschnitt 3.2.3.3) können auf Zeitstempel umformuliert werden. Aktivitäten und zugehörige Events inklusive Zeitstempel könnten über das MXML- oder XES-Format in das Objektdiagramm überführt werden. Die temporalen Beziehungen könnte der Modellierer nachmodellieren und damit Prozessmodelle erzeugen. Die OCL-Invarianten aus dem Workflow-Metamodell für das Process-Mining würden dann angeben, ob auf Basis der Zeitstempel temporale Beziehungen verletzt sind. Dieser Ansatz wird hier jedoch nicht weiter vertieft.

4.5 Verwandte Arbeiten

Der in diesem und Kapitel 3 vorgestellte DMWM-Ansatz tangiert zwei Bereiche der (Wirtschafts-) Informatik. Zum einen wird das Gebiet der UML-modellbasierten und -getriebenen Softwareentwicklung und zum anderen die Workflow- und Geschäftsprozessmodellierung berührt. Das UML-Werkzeug USE, das vorwiegend zur Validierung von UML-Modellen zur Softwareentwicklung genutzt wird, wird mit DMWM auf dem Gebiet der Workflowmodellierung eingesetzt. Im Folgenden werden unterschiedliche Ansätze aus den beiden Bereichen und damit zu DMWM verwandte Arbeiten kurz beschrieben.

Die modellgetriebene Softwareentwicklung verwendet häufig Metamodelle, um auf deren Basis Modelltransformationen durchzuführen. U.a. wurde für solche die Sprache QVT (Query View Transformation) [QVT08] entwickelt. Sie beinhaltet eine imperative OCL-basierte Sprache *imperativeOCL* zum implementieren der Transformationen. Wie in Abschnitt 2.2 schon erwähnt, werden die Transformationen bei QVT dafür genutzt, um Modelle von einer Modellierungssprache in eine andere zu übersetzen. In diesem Zusammenhang ist auch die Atlas Transformation Language (ATL) zu sehen [JABK08], die mit einem Eclipse-Plugin zur Entwicklung und Ausführung von Modelltransformationen umgesetzt wurde.

Kermeta wurde entwickelt, um auf Metamodellebene Transformationen zu definieren, um die darauf basierenden Modelle ausführen zu können. Für diesen Ansatz gibt es ebenfalls eine OCL-basierte imperative Sprache [MFJ05].

Die mit dem UML-Tool USE umgesetzte *A Snapshot Sequence Language* (ASSL) wird ebenfalls dazu genutzt, UML-Modelle zu validieren. Dieses wird erreicht, indem semi-automatisch Objektdiagramme von Klassendiagrammen erzeugt werden, die dann Testfälle darstellen. Eine ähnliche Validierungsmethode für Modelle wird mit Alloy verfolgt, das im Gegensatz zu OCL eine eigene Sprache zum Modelchecking verwendet [Jac03].

Die von USE bereitgestellte Sprache ASSL setzt weite Teile der UML-Action Semantics um, so dass diese Sprache analog zu Kermeta als imperative Sprache zur Ausführung von Modellen genutzt werden kann. Die ebenfalls von USE interpretierte Sprache SOIL [Büt11] stand zum Zeitpunkt der Entwicklung von DMWM noch nicht bereit. Diese Sprache hat wie Kermeta zum Ziel, die Modelle ausführbar zu machen. Sie wird für

den in Abschnitt 6.2 vorgestellten Ansatz zur metamodellbasierten Ausführung von CTT-Aufgabenmodellen genutzt.

Bezogen auf die Workflowmodellierung und -simulation gibt es diverse Ansätze. Im Bereich der Workflow- und Unternehmensmodellierung ist die ARIS Methode und Toolset weit verbreitet [Sch01, Sch02]. ARIS unterstützt neben der EPK nun auch BPMN als Geschäftsprozessmodellierungssprache. Die Modelle können in dem Tool modelliert, simuliert und in andere Sprachen wie BPEL transformiert werden. Mit der Transformation können die Modelle für den operativen Einsatz im Unternehmen weiterentwickelt werden.

Obwohl BPMN in der Spezifikation eine informelle Beschreibung der Ausführungssemantik hat [BPM11] und damit zwangsläufig noch Mehrdeutigkeiten existieren, gibt es Werkzeuge, die die Modelle simulieren und damit validieren können. IYOPRO ist beispielsweise ein solches Tool [IYO].

Mehrdeutigkeiten werden bei Petri-Netzen, für die es weitreichende Ansätze zur Workflowmodellierung und -simulation gibt, durch den formalen Charakter vermieden. Van der Aalst hat auf Basis von Petri-Netzen den Soundness-Begriff für Workflow-Netze entwickelt [AS11]. Auch die Workflow Patterns wurden mit Hilfe der farbigen Petri-Netze formalisiert [RHAM06]. Zur Ausführung von farbigen Petri-Netzen gibt es das Werkzeug *CPN-Tools*, das damit Modelleigenschaften testen kann. Über die Simulation sollen mögliche Optimierungen bei den Prozessen identifiziert werden, wie z.B. eine bessere Ressourcenauslastung erreicht werden kann. Die Ausführung dient hierbei nicht der Validierung der Modelle durch den Nutzer. Hierfür ist hingegen auf Basis von Petri-Netzen ein Werkzeug *Executable Use Cases* entstanden. Die Petri-Netz-Modelle werden mit Informationen für den Nutzer angereichert, wie z.B. aussagekräftige Fotos. Diese Informationen werden dem Nutzer während der Ausführung adäquat dargestellt [JTF09].

Ein deklarativer Ansatz zur Workflowmodellierung existiert mit DECLARE, das auf Basis von LTL Geschäftsabläufe beschreibt [PSSA10]. DECLARE nutzt genauso wie DMWM einen Mechanismus, um die nicht lesbaren textuellen Formeln hinter grafischen Modellierungselementen zu verbergen. Beide Ansätze sind deklarativ und besitzen eine grafische Syntax. DMWM hat im Gegensatz zu DECLARE umfassendere Möglichkeiten zur Daten- und Organisationsmodellierung. Dagegen bietet DECLARE eine Schnittstelle zum Workflowmanagement System YAWL, so dass die Workflow-Engine von DECLARE in einem WfMS genutzt werden kann.

Um hierarchische Modelle zur Workflowmodellierung zu nutzen, ist es möglich, mit CTTE [MPS02] Aufgabenmodelle zu erstellen. Aufgabenmodelle werden vorwiegend im HCI-Bereich u.a. zur modellbasierten User Interface-Modellierung, zur Usability-Analyse und zum Requirements-Engineering genutzt. CTTE kann eine Ausführung der Aktivitätsmodelle simulieren, um sie vom Nutzer zu validieren. Die integrierte Modellierung und Ausführung in einem Tool ist ähnlich zum DMWM-Ansatz. Bei DMWM sind die Modelle jedoch zusätzlich adaptiv, so dass sie zur Runtime verändert werden können. Aufgabenmodelle zur Workflowmodellierung werden intensiver in Kapitel 5 und 6 behandelt.

OCL wurde im Zusammenhang mit Workflowmodellierung bereits in diversen weiteren Arbeiten verwendet. So wurde in [BCC07] das Daten- bzw. Domänenmodell um Workflow-Daten angereichert. Außerdem wurde in [AK01] OCL verwendet, um das Organisationsmodell an das Workflowmodell zu binden. Diese Ansätze wurden aber im Gegensatz zu DMWM nicht UML-Tool-basiert ausführbar gemacht.

Mit DMWM und OCL ist eine Analyse der ausgeführten Prozesse möglich, so dass dieser Ansatz auch zum Process-Mining [Aal11] genutzt werden kann, so wie es in Abschnitt 4.4 erläutert wurde. In diesem Gebiet wird das ProM-Tool [DMV⁺05] eingesetzt, um Analysen von Event-Logs von bereits abgearbeiteten Prozessen durchzuführen und ggf. Prozessmodelle daraus zu erzeugen. Schnittstellen für Tools zur Analyse der Daten aus Workflow Management Systemen waren schon im WfMC-Referenzmodell [Hol98] vorgesehen. Hier sollen z.B. Monitoring-Werkzeuge zur visuellen Aufbereitung der abgelaufenen und aktuell laufenden

Prozesse den Betrieb im Unternehmen anzeigen. Dieses Gebiet wird auch als *Business Process Intelligence* bezeichnet [MR06].

Kapitel 5

Hierarchieorientierte Workflowmodellierung

Dieses Kapitel widmet sich der hierarchieorientierten Workflowmodellierung mit Aufgabenmodellen. Bisher werden zur Beschreibung von Geschäftsprozessen vornehmlich flache Modellierungssprachen wie EPKs, UML-Aktivitätsdiagramme und BPMN verwendet. Auch DMWM und Declare-Modellen [PSSA10] zählen zu dieser Kategorie von Sprachen.

Abschnitt 5.1 gibt eine Einführung in bereits vorhandene hierarchische Modellierungssprachen, die zur Geschäftsprozessmodellierung eingesetzt werden. Zudem werden Aufgabenmodelle und insbesondere die derzeit populärste Sprache ConcurTaskTrees (CTT) vorgestellt. Die Transformation von CTT-Aufgabenmodellen zu Aktivitätsdiagrammen ist Thema in Abschnitt 5.2. Dort werden spezifische Eigenheiten der temporalen CTT-Operatoren deutlich und die Strukturiertheit der Prozessmodelle gezeigt.

Um den Aufgabenmodell-Ansatz weiter zu formalisieren, präsentiert Abschnitt 5.3 ein präzises Metamodell für CTT-Aufgabenmodelle, das mit OCL-Invarianten Konsistenzeigenschaften für CTT-Modelle festlegt. Es wird gezeigt, dass dieser Ansatz mit dem UML-Tool USE praktikabel eingesetzt werden kann, um hierarchische, fehlerfreie Aufgabenmodelle zu erhalten. Ein wichtiger Teil der Workflowmodellierung ist die Entscheidungsmodellierung, den der Abschnitt 5.4 für Aufgabenmodelle diskutiert. Die derzeit übliche implizite Entscheidungsmodellierung bei CTT-Modellen und Möglichkeiten zur expliziten Entscheidungsmodellierung werden dort vorgestellt.

5.1 Einführung

Abschnitt 5.1.1 erörtert, inwieweit hierarchische Modelle zur Geschäftsprozessmodellierung bereits eingesetzt und genutzt werden. Aufgabenmodelle und Vor- und Nachteile für deren Einsatz zur Geschäftsprozessmodellierung werden daraufhin in Abschnitt 5.1.2 untersucht.

5.1.1 Verbreitete hierarchische Modellierungstechniken

Es gibt bereits Geschäftsprozessmodellierungssprachen, die eine hierarchische Baumstruktur besitzen. Der große Vorteil einer Baumstruktur ist, dass mehrere Abstraktionsebenen in einem Modell vereint werden. Dadurch können Zuordnungen einzelner Aktionen, Aktivitäten bzw. Funktionen zu höher geordneten

schneller erfasst werden, als wenn Dekompositionen bei flachen Modellierungssprachen wie BPMN über mehrere Modelle geschehen.

Funktionsbäume verfolgen beispielsweise diese hierarchische Modellierungsweise, die u.a. in der Softwaretechnik, im Requirements Engineering [Bal09] und in der Geschäftsprozessmodellierung (im ARIS-Ansatz) [Sch01] eingesetzt werden. Darin werden Funktionen beschrieben, die das System erfüllen soll und rekursiv durch Dekomposition in Subfunktionen untergliedert.

Im Gegensatz zur strukturierten Analyse [DeM79] oder UML-Aktivitätsdiagrammen [UML10] werden verschiedene Abstraktionsebenen in einem Modell visualisiert. Dekomposition von Funktionen ist zwar auch mit flachen Sprachen möglich, dort müssen aber mehrere Abstraktionsebenen durch mehrere Modelle ausgedrückt werden. UML-Aktivitätsdiagramme haben dafür die UML-Aktion *CallBehaviourAction*, die auf ein weiteres Modell verweist. Dieses Mittel wird in Abschnitt 5.2 dargestellt, in dem Transformationsregeln von Aufgabenmodellen zu Aktivitätsdiagrammen vorgestellt werden.

In Aufgabenmodellen ist die Angabe des Ablaufs ebenfalls Gegenstand der Modellierung innerhalb des hierarchischen Modells (s. Abschnitt 5.1.2). Dieses ist in Funktionsbäumen, bei denen temporale Abhängigkeiten nicht Gegenstand der Betrachtung sind, nicht der Fall. Bei der Entwicklung von Systemanforderungen bzw. Softwaremodulen spielt dieser Aspekt keine bzw. eine untergeordnete Rolle. Die Funktionsbäume stellen eine systemzentrierte Modellierung dar. In Abbildung 5.1(a) ist dafür ein Beispiel angegeben, das die Funktion und Unterfunktionen eines Enterprise Resource Planing (ERP-) Systems modelliert.

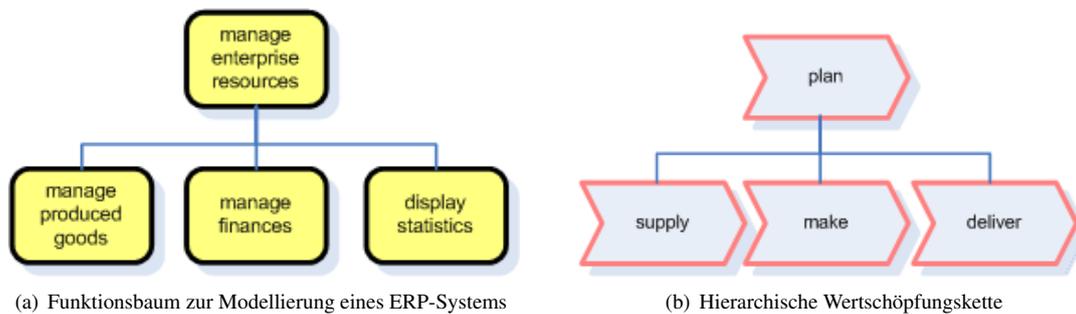


Abbildung 5.1: Hierarchische Modelle

In ARIS sind die Funktionsbäume ebenfalls als Modellierungsart in der Funktionssicht (s. Abbildung 2.2(a)) integriert. Ein Ausschnitt des Metamodells für die ARIS-Funktionssicht wurde des Weiteren in Abbildung 2.6 angegeben. Dort ist die hierarchische Darstellung durch die Assoziationsklasse *Funktionsstruktur* zu sehen. Zudem ist dort mit der Assoziationsklasse *Anordnung* die Möglichkeit bereitgestellt, eine sequenzielle Ablaufreihenfolge in der Hierarchie mitzumodellieren. Dieses Metamodell erweitert also die Betrachtungsweise um Ablaufketten in der Funktionsbaumdarstellung und kann damit auch hierarchische Wertschöpfungsketten ausdrücken. In Abbildung 5.1(b) wird ein Beispiel dafür aufgezeigt.

In Wertschöpfungsketten werden auf sehr hoher Abstraktionsebene die Tätigkeiten beschrieben, die sequenziell im Unternehmen abzuarbeiten sind, um die Werte zu schaffen, mit denen das Unternehmen Geld verdient [Por85]. In der Regel werden in einer Wertschöpfungskette Produkte produziert und Ressourcen dafür verbraucht. Die Beschaffung der Ressourcen wird mit Hilfe von Lieferketten beschrieben.

5.1.2 Aufgabenmodelle zur Spezifikation von Workflows

Die erste Modellierung von Aufgaben wurde mit Hierarchical Task Analysis (HTA) Ende der 1960er Jahre eingeführt [Ann03]. Wie es der Name der Modellierungsart schon vorgibt, ist die hierarchische Darstellung

der Aufgaben ein integraler Bestandteil der Modelle. Zusätzlich ist an jeder Dekomposition einer Aufgabe ein sogenannter Plan zu annotieren, der informell beschreibt, in welcher Reihenfolge die Unteraufgaben zu erledigen sind. Diese informelle Angabe mit umgangssprachlichen Text repräsentiert die Kontrollflusskanten in flussorientierten Modellierungssprachen. Informelle Texte beinhalten von Natur aus Mehrdeutigkeiten, womit diese für eine Soundness-Analyse bzw. Workflow-artige, rechnergestützte Ausführung der HTA-Modelle nicht ausreicht.

Weitere Aufgabenmodellierungsansätze sind mit *Goals Operations Methods Structure* (GOMS) [Kie03] und *Task Knowledge Structure* (TKS) [JJ91] gefolgt. Diese werden ebenfalls nur zur konzeptionellen Modellierung verwendet und sind nicht dafür vorgesehen, ausgeführt zu werden.

Die Sprache ConcurTaskTree (CTT) [Pat99] hat sich als populärste Aufgabenmodellierungssprache etabliert. Diese wird vornehmlich im Zusammenhang mit der User Interface-Modellierung eingesetzt. Die ConcurTaskTrees haben die Angabe des Kontrollflusses für Aufgabenmodelle eher formalisiert (s. Abschnitt 5.2.2), für eine Workflowausführung sind aber auch diese Modelle noch zu informell. Zum einen fehlt eine Veröffentlichung zur formale Fundierung, nach der eine eindeutige Abarbeitung bei allen Operatoren und deren Kombinationen definiert ist. Zum anderen liefert das Referenzwerkzeug ConcurTaskTree Environment (CTTE) [MPS02] zwar eine Umgebung zur Simulation der Modelle, dort entstehen aber schnell Probleme wie Dead- bzw. Lifelocks, die während der Modellierung nicht immer automatisch erkannt werden. Eine Formalisierung, die eine Soundness-Überprüfung zur Designtime erlaubt, ist hier notwendig, um Aufgabenmodelle zur Workflowausführung zu nutzen. Abschnitt 5.3 präsentiert einen präzisen metamodellbasierten Ansatz für den Zweck, Konsistenzeigenschaften von CTT-Modellen zu überprüfen.

Wendet man Aufgabenmodelle für Geschäftsprozesse an, muss man beachten, dass die Wurzelaufgabe kein Nutzerziel verfolgt, sondern eher das Ziel des Unternehmens oder der Abteilung. Dieser Sachverhalt ist im Kontrast zur nutzerzentrierten Aufgabenmodellierung zu sehen. Daraufhin wird das Ziel, wie in Aufgabenmodellen üblich, in einer baumartigen Form dekomponiert. Hierbei ist eine grafische Integration der Organisationsmodellierung durch Pools bzw. Swimlanes, so wie sie bei BPMN vorhanden ist, nicht vorgesehen. Stattdessen können Cooperative ConcurTaskTrees (CCTT) eingesetzt werden, um kooperierende Tätigkeiten zu modellieren [MPS02]. Dafür wird für jede beim Geschäftsprozess beteiligte Rolle bzw. Person ein Aufgabenbaum zugeordnet. Die Koordination der unterschiedlichen Aufgaben wird dann in einem separaten, koordinierenden Aufgabenbaum angegeben. In diesem wird im Wurzelknoten das gemeinschaftliche Ziel modelliert, das nicht einer Person sondern der Organisationseinheit zuzuordnen ist.

Hierarchische Modelle, die mehrere Abstraktionsebenen in einem Modell darstellen, haben neben der integrierten Zielmodellierung einen weiteren Vorteil gegenüber imperativen Sprachen. Es kann eine „depth-first“-Modellierung verfolgt werden im Gegensatz zu nicht hierarchischen Modellen, die eine „breadth-first“-Modellierung erzwingen. Omerod hat festgestellt, dass diese Herangehensweise Vorteile bei der Modellierung aus folgenden Gründen haben kann [OR99]. Kritische Probleme und wiederum spezielle Unterprobleme können beispielsweise durch Dekomposition als erstes genauer betrachtet werden. Dadurch kann man zur Erkenntnis gelangen, dass das Problem so nicht gelöst werden kann und ein anderes Handlungsmodell erforderlich ist. Es kann somit die Erstellung des Problemlösungsmodells abgebrochen und ein neues begonnen werden. Hätte man hier eine „breadth-first“-Modellierung verfolgt wäre deutlich mehr Arbeit zur Modellierung vonnöten gewesen, bis man zu dieser Erkenntnis gekommen wäre.

Strukturiertheit ist des Weiteren ein wichtiger Aspekt sowohl in der Programmierung als auch zunehmend in der Modellierung. Dijkstra hat Ende der 1960er Jahre erkannt, dass goto-Anweisungen in Computerprogrammen zu komplizierten, nicht nachvollziehbaren Programmen führen [Dij68]. So wie goto-Anweisungen bei komplexen Programmen nicht nachvollziehbar sind, können flussorientierte Prozessmodellierungssprachen ebenfalls zu solchen unverständlichen Prozessmodellen führen. In [FL11] wurde die kognitive Komplexität

und Verständlichkeit von Prozessmodellen untersucht. Viele Prozessmodelle sind zu komplex, um vom Anwender verstanden zu werden. Die kognitive Fähigkeit des Menschen ist sehr begrenzt und daher ist es äußerst wichtig auf die Verständlichkeit der Modelle zu achten.

Process Structure Trees (PSTs) wurden in [VVK09] als technische Repräsentationsmodelle für strukturierte Prozesse genutzt. Dieses ist analog zu Abstract Syntax Trees bei strukturierten Programmiersprachen zu sehen, die durch die Kompilierung solcher Programme entstehen. Die baumartige Darstellung wird dabei nicht als Modelldarstellung für den Modellierer verwendet, sondern zur Weiterverarbeitung des Programmcodes. Bei Aufgabenmodellen ist dies anders, indem die Bäume direkt zur Modellrepräsentation zur Designzeit genutzt werden. Diese Modelle sichern ebenfalls eine Strukturiertheit zu, die äußerst wichtig für die Verständlichkeit ist [LM10]. Analog dazu sind die baumartigen *Jackson Structure Diagrams* (JSD) zur semi-formalen, hierarchischen Entwurfstechnik von strukturierten Computerprogrammen zu sehen [Jac82]. Sie werden bei der *Jackson System Development*-Methode eingesetzt, um Ausgangsmodelle für das *Jackson Structured Programming* (JSP) zu schaffen [Cam89].

Eine große Ähnlichkeit von JSP-Diagrammen zu CTT-Aufgabenmodellen ist vorhanden. Beide Diagrammarten werden jedoch zu unterschiedlichen Modellierungszwecken eingesetzt. JSPs werden zum Entwurf von strukturiertem Programmcode genutzt und CTT-Modelle zur nutzerzentrierten Aktivitätsmodellierung im Zusammenhang mit der User Interface-Modellierung. Wo bei JSPs vorwiegend die sequenzielle Abarbeitung von Geschwisterknoten im Baum genutzt wird, gibt es bei CTT zudem deutlich umfangreichere Varianten der temporalen Operatoren. Im Zusammenhang mit den Transformationen von CTT-Modellen zu Aktivitätsdiagrammen wird in Abschnitt 5.2.2 näher auf die Eigenschaften der Operatoren eingegangen.

5.2 Transformation von CTT zu Aktivitätsdiagrammen

Aufgabenmodelle zählen zu den informellen bzw. semiformalen Modellierungssprachen, die Mehrdeutigkeiten in der Ablauflogik beinhalten. CTT-Aufgabenmodelle gehören hierbei zu den formalen Ansätzen im Vergleich zu z.B. HTA [Ann03] oder GOMS [CMN83]. In der Hierarchie werden für CTT zwischen Nachbarknoten im Baum temporale Operatoren genutzt, die von der Sprache LOTOS (Language Of Temporal Ordering Specification) abgeleitet wurden [BB89].

CTT wird vorwiegend, wie in Abschnitt 5.1.2 erwähnt, als Sprache für Analysemodelle zur Entwicklung von User Interfaces verwendet. Aufgabenmodelle stellen dabei die Ausgangsmodelle zur Erzeugung von abstrakten User Interface-Modellen dar, aus denen wiederum konkrete abgeleitet werden sollen [MPV11]. Die Nutzerziele im Zusammenhang mit den Aktionen, die der Nutzer mit den User Interface-Objekten erreichen will, werden mit den höher gelegenen Aufgaben in der hierarchischen Struktur der Aufgabenmodelle ausgedrückt. Der Einsatz von CTT-Aufgabenmodellen wird insbesondere bei der XML-basierten User Interfaces-Beschreibungssprache *UsiXML* für diese Zwecke genutzt [LVM⁺05]. Dort werden die Aufgabenmodelle nur als konzeptionelle Modelle eingesetzt und nicht zur Modellausführung verwendet. Es gibt jedoch auch Ansätze, die die Aufgabenmodelle mit einer Engine im Zusammenhang mit einer Dialogsteuerung ausführen sollen [Ric09]. Mit der Ausführung der Modelle stellen sich Fragen der Soundness-Problematik und der operationalen Semantik bestimmter Operatoren, die in den Abschnitten 5.3 und 6.2 für CTT-Modelle behandelt werden.

Anhand der Verwendung der sowohl ausführbaren als auch nicht-ausführbaren Ansätze soll Software mit nutzungsfreundlicheren User Interfaces entstehen, die die Ziele der Nutzer berücksichtigen. Bei beiden Ansätzen ist die Semantik der temporalen Operatoren wichtig. Insbesondere bei den ausführbaren Ansätzen sollten Mehrdeutigkeiten ausgeschlossen werden. Durch die in diesem Abschnitt betrachteten Transforma-

tionen der temporalen Operatoren von CTT zu UML-Aktivitätsdiagrammen kann diese Semantik genauer beleuchtet werden. Mit der Modellrepräsentationen in Aktivitätsdiagrammen lassen sich Eigenschaften bestimmter CTT-Operatoren besser visualisieren. Hier ist z.B. der *Disabling* zu nennen, dessen Semantik sich von der LOTOS-Modellierungssprache unterscheidet [BB89]. Außerdem ist der *Iteration*-Operator zu nennen, dessen Semantik von einer weiteren Aufgabenmodellierungssprache abweicht [DF09].

5.2.1 Transformation der Hierarchie

Die baumartige Repräsentation von Aufgabenmodellen und die entsprechende Darstellung in Aktivitätsdiagrammen ist in Abbildung 5.2 veranschaulicht und wird in diesem Abschnitt behandelt.

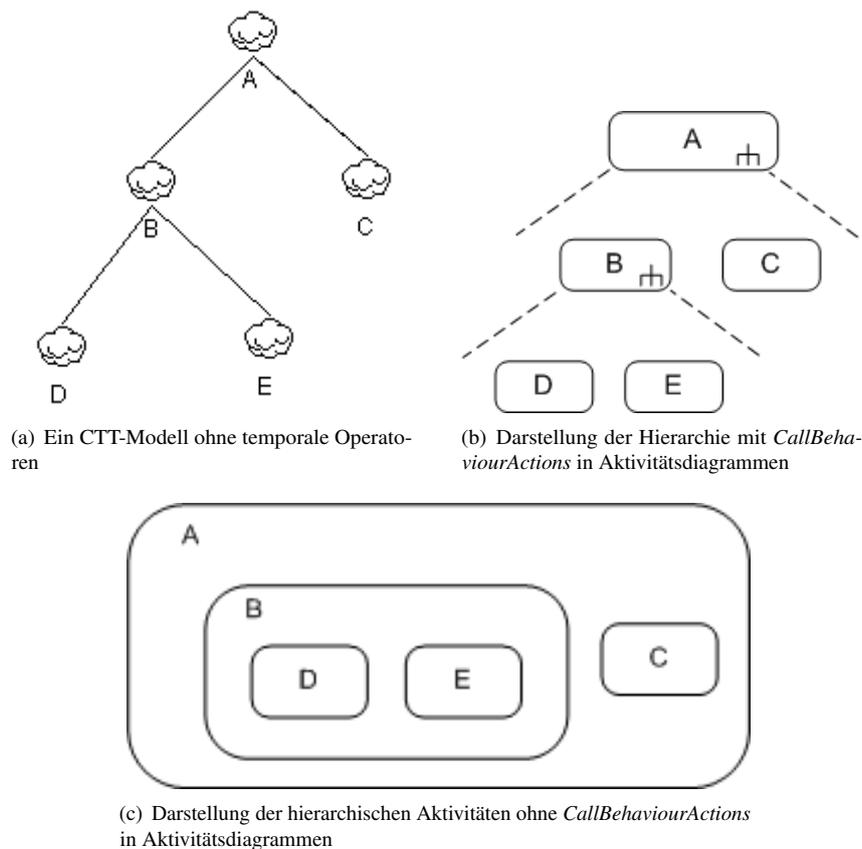


Abbildung 5.2: Darstellung der Hierarchie in Aktivitätsmodellen

Die Hierarchie, die die Dekomposition von Aufgaben in Aufgabenmodellen widerspiegelt, wird bei den Aktivitätsdiagrammen über *CallBehaviourActions* ausgedrückt. Eine spezielle Form dieser UML-Aktion wird mit einem UML-Hierarchiesymbol (in Abbildung 5.2(b) zu sehen) gekennzeichnet und repräsentiert eine Aufrufaktion einer untergeordneten Aktivität [UML10, Abs.12.3.14]. D.h. untergeordnete Aktivitäten können keine übergeordneten aufrufen und insbesondere sind damit keine rekursiven Aufrufe erlaubt. Für die Transformation von CTT-Aufgabenbäumen wird genau diese Aufrufaktionen benötigt.

Die Transformation der Hierarchie von CTT-Modellen zu Aktivitätsdiagrammen ist in Abbildung 5.2 zu sehen. Dafür ist zunächst ein CTT-Baummodell in Abbildung 5.2(a) angegeben, in dem die temporalen Operatoren weggelassen wurden. Es ist mit *A* die Wurzelaktivität angegeben, die mit den Aktivitäten *B* und *C* dekomponiert wurde. Zusätzlich wurde Aktivität *B* in die Unteraufgaben *D* und *E* aufgegliedert.

In Abbildung 5.2(b) sind dann die *CallBehaviourActions* mit dem UML-Hierarchiesymbol bei den Aktivitäten *A* und *B* angegeben. Hierbei sind mit den drei Hierarchieebenen drei Aktivitätsdiagramme zu sehen, die jedoch noch keinen Sequenzflusskanten beinhalten. Eine konzeptionelle Darstellung der Hierarchie, die bei Aktivitätsdiagrammen eher unüblich ist, ist in Abbildung 5.2(c) zu sehen.

An dem Modell von Abbildung 5.2(a) ist ersichtlich, dass durch die Baumdarstellung mehrere Abstraktionsebenen in einem Modell visualisiert werden. Anhand dessen lassen sich die oben erwähnten Nutzerziele ausdrücken. Dagegen ist wie in Abbildung 5.2(b) bei flachen Modellierungssprachen wie Aktivitätsdiagrammen diese Zielmodellierung nur über den Umweg weiterer Modelle ausdrückbar.

5.2.2 Einführung der temporalen CTT-Operatoren

Die Operatoren, die CTT zur Angabe der temporalen Beziehungen von Aufgaben bereitstellt, sind in Tabelle 5.1 aufgelistet.

CTT-Operator	Bezeichnung	Operationale Semantik
A [] B	Choice	Entweder A oder B sind auszuführen.
A B	Concurrency	A und B werden unabhängig voneinander abgearbeitet.
A [] B	Concurrency With Information Exchange	A und B sind unabhängig voneinander auszuführen, tauschen aber währenddessen Informationen aus.
A > B	Deactivation	B kann die Ausführung von A abbrechen. Zudem ist die Ausführung von B erforderlich, auch wenn A schon beendet wurde. Diese Eigenschaft unterscheidet den CTT-Deactivation vom LOTOS-Deactivation-Operator.
A > B	Suspend-resume	Solange A läuft kann B die Aktivität unterbrechen. Dieses kann mehrfach während der Abarbeitung von B geschehen, so dass B damit mehrmals ausgeführt wird.
A = B	Order Independence	Entweder wird A oder B (jeweils inklusive Unteraufgaben) komplett abgearbeitet bevor die andere Aufgabe starten darf.
A » B	Enabling	A muss vor dem Starten von B abgearbeitet worden sein.
A []» B	Enabling with information passing	A muss vor dem Starten von B abgearbeitet worden sein. A stellt des Weiteren Information für B bereicht.
A*	Iteration	A kann beliebig häufig ausgeführt werden, solange die darauffolgende Aufgabe mit einem <i>Deactivation</i> - oder <i>Choice</i> -Operator nicht zur Ausführung ausgewählt wurde.
[A]	Option	A kann optional ausgeführt werden

Tabelle 5.1: Temporale CTT-Operatoren

In diesem Abschnitt sollen die temporalen Operatoren informell eingeführt werden, um sie in den Abschnitten 5.2.3 und 5.2.4 zur semiformalen Repräsentation in Aktivitätsdiagrammen zu nutzen. Des Weiteren wird eine formale Spezifikation dieser Operatoren in Abschnitt 5.3 folgen, in dem ein präzises Metamodell für CTT vorgestellt wird. Dort werden zudem Konsistenzprobleme aufgezeigt, die temporale Operatoren in CTT-Modellen hervorrufen können.

In der linken Spalte der Tabelle 5.1 ist die Syntax der Operatoren angegeben. Daraufhin ist in der mittleren die Bezeichnung der Operatoren zu sehen und in der rechten umgangssprachlich und informell die operationale Semantik beschrieben.

Die temporalen Operatoren aus Tabelle 5.1 haben Prioritäten, die notwendig für die Interpretation der Abarbeitungsreihenfolge mehrerer Geschwisterknoten im Aufgabenbaum sind. CTT ist ein grafischer

Modellierungsformalismus, in der keine Klammerungen zur Angabe der Abarbeitungsreihenfolgen der temporalen Operatoren möglich sind. In textuellen Prozessalgebra-Termen können Klammerungen dagegen in den Gleichungen bzw. Formeln verwendet werden.

Die Prioritäten zur Angabe der Bindung für die binären CTT-Operatoren sind in der Reihenfolge wie in Tabelle 5.1 angegeben wie folgt: Choice ($[]$ Operator) > Parallel composition ($|||$ und $|||$ Operatoren) > Disabling ($[>$ Operator) > Suspend-resume ($[>$ Operator) > Order independence ($[=|$ Operator) > Enabling ($[>$ und $[]>$ Operatoren).

Mit der oben angegebenen Priorisierung der Geschwisterknoten würde der Term $A ||| B \gg C [] D$ damit wie folgt interpretiert werden: $(A ||| B) \gg (C [] D)$.

5.2.3 Transformation der binären temporalen Operatoren

Die CTT-Operatoren, die in diesem Abschnitt zu Aktivitätsdiagrammen transformiert werden, sind *Concurrency*, *Choice*, *Order independence*, *Deactivation* und *Enabling*. In Tabelle 5.2 sind die CTT-Operatoren und die entsprechenden Repräsentationen als Aktivitätsdiagramme angegeben.

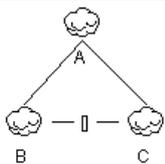
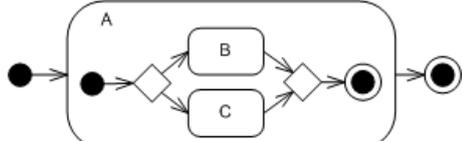
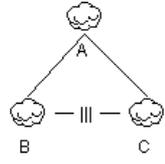
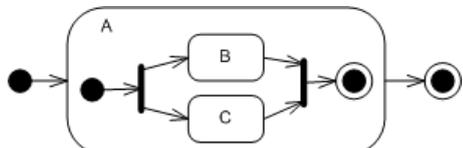
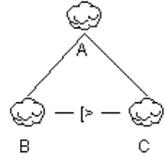
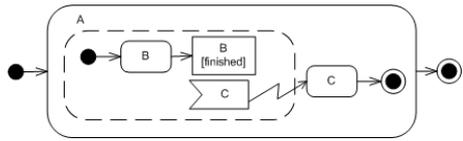
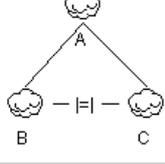
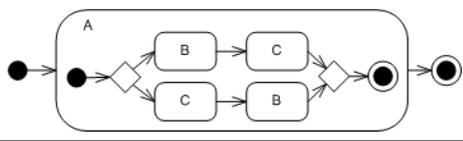
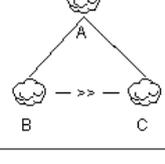
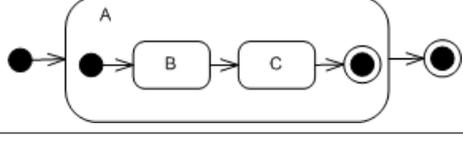
CTT-Modell	Repräsentation als Aktivitätsdiagramm
	
	
	
	
	

Tabelle 5.2: Transformation der CTT-Operatoren in Aktivitätsdiagramme

In Tabelle 5.2 ist als erstes der *Choice*-Operator transformiert worden. Für das zugehörige Aktivitätsdiagramm wird ein *Choice*- und *Merge*-Operator strukturiert eingesetzt. An dem *Choice*-Operator muss

entschieden werden, ob Aufgabe B oder C auszuführen ist. Diese Entscheidung muss bei CTT implizit vom Nutzer getroffen werden und basiert nicht auf Daten, wie z.B. bei BPMN (vgl. Abschnitt 2.5.2.3). Für eine explizite Entscheidungsmodellierung fehlt die Angabe der Entscheidungstätigkeit und die Kriterien zur Pfadauswahl, wie es z.B. bei EPKs gefordert wird (vgl. Abschnitt 2.5.2.1). Somit ist die Entscheidung, welche Aktivität auszuführen ist, implizit zu fällen. Bei den Workflow Patterns wird diese Art als *Deferred Choice* bezeichnet [AHKB03] und repräsentiert im Vergleich zur expliziten Entscheidungsmodellierung eher die Ausnahme bei Workflowmodellen. Eine Diskussion, wie eine explizite Entscheidungsmodellierung für Aufgabenmodelle erfolgen kann, wird in Abschnitt 5.4 gegeben.

Der zweite betrachtete Operator in der Tabelle 5.2 ist der *Concurrency*-Operator (|||). Für die Transformation wird für das Aktivitätsdiagramm ein *fork*- und *join*-Operator strukturiert eingesetzt. Die Aufgaben B und C stehen zwischen diesen beiden Operatoren und können somit unabhängig voneinander ausgeführt werden.

Der *Disabling*-Operator (>) wird daraufhin betrachtet. Bei dem zugehörigen Aktivitätsdiagramm ist eine Eigenheit von CTT zu sehen, die diesen Operator angeht und von der LOTOS-Semantik abweicht. Die Aufgabe C muss bei CTT in jedem Falle ausgeführt werden, auch wenn B schon beendet wurde. In diesem Falle endet der Kontrollfluss beim Aktivitätsdiagramm im Objektknoten *B [finished]* und die *Exception Region* bleibt weiterhin aktiv. Es wird somit in jedem Fall auf das Signal zur Abarbeitung der Aktivität C gewartet. Bei LOTOS ist C dagegen nicht weiter ausführbar, wenn B schon beendet wurde. Die operationale Semantik ist dort präziser an der eigentlichen Bezeichnung des Operators, die aussagt, dass die nachfolgende Aufgabe die Vorgängeraufgabe deaktiviert. Bereits beendete Aufgaben können eigentlich nicht deaktiviert werden.

OrderIndependency (|=) wird in Tabelle 5.2 im Aktivitätsdiagramm so umgesetzt, dass hinter dem *Choice*-Operator die möglichen Ablaufsequenzen der beteiligten Aktivitäten spezifiziert sind. Beim *Choice*-Operator wird dann entschieden, welche Aktivität als erstes auszuführen ist. Die übrig gebliebene bleibt danach auszuführen.

Der *Enabling*-Operator (>>) wird als letztes in der Tabelle 5.2 transformiert. Die sequenzielle Abfolge von Aufgabe A und B wird beim Aktivitätsdiagramm durch eine simple Kontrollflusskante ausgedrückt, die die Abarbeitungsreihenfolge spezifiziert.

Aus Tabelle 5.1 sind des Weiteren noch die Operatoren *Concurrency with Information Exchange* und *Enabling with Information Passing* vorhanden. Wie dort in der *operationalen Semantik*-Spalte angegeben, unterscheidet sich die Ablaufsemantik dieser Operatoren nicht von *Concurrency* bzw. *Enabling*. CTT gibt mit diesen Operatoren lediglich an, dass Informationen zwischen den Geschwisterknoten ausgetauscht werden. Eine genauere Angabe der Daten und deren Inhalt ist in CTT nicht möglich. Aktivitätsdiagramme verlangen bei der Datenintegration mit Objektflüssen eine Typangabe der Objekte, die zwischen Aktivitäten ausgetauscht werden (s. Abschnitt 3.3.1). Da die Konzepte hier nicht zueinander passen, werden für diese spezifischen Operatoren keine Transformation vorgestellt.

Der einzige temporale CTT-Operator, dessen Kontrollflusssemantik nicht in einem Aktivitätsdiagramm dargestellt werden kann ist *Suspend-resume*. Für dessen Umsetzung im Aktivitätsdiagramm wäre ein *History-Zustand*, analog zu dem, der bei UML-Zustandsdiagrammen vorhanden ist [UML10, Kap.15], notwendig. Das Merken des Ausführungszustandes der unterbrochenen Aufgabe ist notwendig, um zu diesem zurückzukehren, wenn die unterbrechende Tätigkeit beendet wurde. Da der *History-Zustand* aber für Aktivitätsdiagramme nicht verfügbar ist, kann die Transformation ohne Erweiterung der Sprachmittel der Aktivitätsdiagramme nicht ausgedrückt werden und bleibt offen.

5.2.4 Transformation der unären temporalen Operatoren

Zu den unären CTT-Operatoren gehört die Iteration und die Option (s. Tabelle 5.1). Die Iteration drückt aus, dass die damit versehene Aufgabe beliebig häufig (0 bis n mal) ausgeführt werden kann. Bei der Option kann die damit angegebene Aktivität 0 oder höchstens einmal ausgeführt werden. CTT-Modelle mit diesen Operatoren werden in Tabelle 5.3 in Aktivitätsdiagramme transformiert.

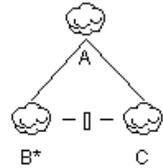
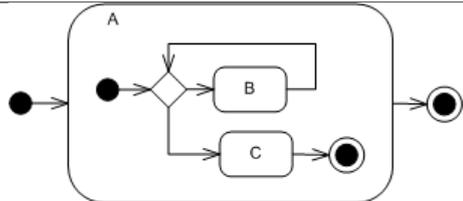
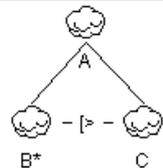
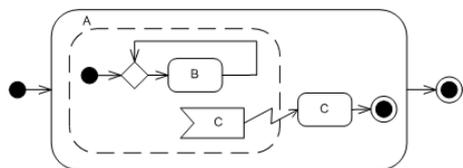
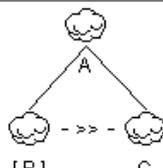
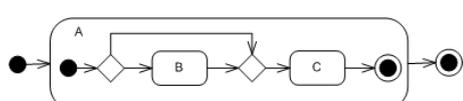
CTT-Modell	Repräsentation als Aktivitätsdiagramm
	
	
	

Tabelle 5.3: Transformation der unären CTT-Operatoren in Aktivitätsdiagramme

Für die Iteration ist zu beachten, dass auf eine Iteration in CTT kein *Sequence*-Operator folgen darf (s. Abbildung 5.3(d) in Abschnitt 5.3.1). Ein Abbruch der Iteration kann bei der Semantik, die die CTT-Referenzimplementierung in CTTE vorgibt, nur durch einen *Choice* oder *Disabling*-Operator hervorgerufen werden. Eine Spezifikation zur operationalen Semantik wurde bei CTT hierfür nicht veröffentlicht. Eine Korrespondenz zum Entwickler von CTT Fabio Paterno hat hier aber für Aufschluss gesorgt. In Tabelle 5.3 sind die beiden Möglichkeiten, eine Iteration zu beenden bzw. abubrechen, als erstes angegeben. Bei der ersten Transformation zum Aktivitätsdiagramm ist zu sehen, dass der Choice-Knoten bei jedem Iterationsdurchlauf von Aufgabe B immer wieder durchlaufen wird. Dieser Knoten ist somit gleichzeitig auch ein Merge, so dass mit Angabe der Iteration aus dem Choice ein hybrider Knoten entsteht. Dort wird entschieden, ob ein (weiterer) Iterationszyklus von Aufgabe B durchgeführt oder mit Ausführung der Aufgabe C die Iteration beendet werden soll.

Die Verwendung des Disabling-Operators mit der Iteration ist insofern anders, als dass C dort jederzeit die Iteration beenden kann, auch wenn sich Aufgabe B noch in der Ausführung befindet. Dagegen kann bei der ersten Iterationsvariante nur bei vollständig durchgeführten Iterationszyklen die Iteration beendet werden. Die erste Variante ist bei Workflowmodellen eher üblich, wohingegen bei CTT-Aufgabenmodellen zur Modellierung für User Interfaces häufiger die zweite Variante genutzt wird (vgl. [Pat99]). In jedem Falle ist einer dieser Operatoren in Verbindung mit der Iteration vonnöten, um ein von Livelocks befreites und damit fehlerfreies Prozessmodell zu erhalten. In Abschnitt 5.3.1 werden u.a. diese Probleme bei CTT-Modellen angesprochen.

Die dritte in Tabelle 5.3 zeigt eine Transformation einer optionalen Aufgabe in Verbindung mit dem Sequenz-Operator. Das Aktivitätsdiagramm zeigt die optionale Aufgabe durch einen alternativen Kontrollfluss, der über einen Choice- und Merge-Operator die Aufgabe B umgeht. Der Nutzer kann also B ausführen oder direkt C und damit den alternativen Kontrollflusspfad nutzen.

Bei beiden in diesem Abschnitt betrachteten unären Operatoren ist zu beachten, dass sie in Verbindung mit weiteren binären Operatoren Konsistenzprobleme im CTT-Modell hervorrufen können. Einige dieser Probleme werden in Abschnitt 5.3.1 angesprochen.

5.3 Definition konsistenter CTT-Aufgabenmodelle

In diesem Abschnitt wird das Metamodell zur Definition der *Metamodel-based ConcurTaskTrees* (MCTT) vorgestellt. Für CTT gibt es bereits Metamodelle [BB06, Bas09], die aber noch nicht zur strikten Konsistenzprüfung der Modelle zur Designzeit eingesetzt werden. Zunächst werden Probleme bei CTT-Modellen in Abschnitt 5.3.1 vorgestellt. Entsprechende Modellierungsfehler sollen mit dem hier vorgestellten Ansatz bereits während der Designzeit erkannt werden.

Das UML-Klassendiagramm, welches das CTT-Metamodell darstellt, wird in Abschnitt 5.3.2 eingeführt. In Abschnitt 5.3.3 werden OCL-Invarianten ebenfalls wie beim DMWM-Ansatz eingesetzt (s. Abschnitt 3.2.6), um strukturelle Eigenschaften der Modelle festzulegen. Damit soll der Modellierer bereits während der Designzeit auf potenzielle Laufzeitfehler hingewiesen werden. Die Anwendung des Metamodells zur MCTT-Modellierung wird Thema in Abschnitt 5.3.4 sein.

5.3.1 Soundness-Probleme bei CTT-Aufgabenmodellen

Es können diverse Soundness-Probleme bei der CTT-Modellierung auftreten, wie es in Abbildung 5.3 angegeben ist und im Folgenden diskutiert wird. Die Modelle von Abbildung 5.3(a) bis 5.3(d) werden dabei von der CTTE-Modellierungsumgebung als problematisch erkannt. Die Probleme der Modelle in Abbildung 5.3(e) und 5.3(f) werden von CTTE nicht identifiziert.

In dem angegebenen Beispiel von Abbildung 5.3(a) ist eine optionale Aufgabe hinter einem Disabling-Operator spezifiziert. Die Problematik dieses Modells wird anhand der Aktivitätsdiagramm-Transformationen des Disabling-Operators in Tabelle 5.2 bzw. 5.3 deutlich. Der Disabling-Operator wartet auf das Signal, das die folgende Aufgabe startet. Erst dann kann der Kontrollfluss mit dieser fortfahren. Wenn diese Aufgabe als optional angegeben ist, bedeutet das, dass sie nicht ausgeführt werden muss. Soll sie nun nicht ausgeführt werden, wird dieses Signal nicht kommen und die Abarbeitung des Modells bleibt stecken. Man befindet sich in einem Deadlock.

Ein ähnliches Problem existiert bei CTT-Modellen mit optionalen Aufgaben, die als letztes im Aufgabenbaum spezifiziert sind. Wenn bei der Ausführung die letzte Aufgabe nicht ausgeführt werden soll, ist nicht klar, dass die Modellarbeitung bereits beendet ist. CTTE wartet bei der Ausführung eines solchen Modells weiterhin auf die Abarbeitung der letzten Aufgabe, so dass deren Abarbeitung doch notwendig und nicht optional ist.

Ein weiteres Konsistenzproblem ist in Abbildung 5.3(b) zu sehen. Dort ist die Aufgabe *B* optional spezifiziert und steht in der *Choice*-Beziehung zu Aufgabe *C*. Betrachten wir den Fall, dass *C* nicht abgearbeitet werden soll und sich deswegen für den linken Zweig des Aufgabenbaumes entschieden wird. Hierfür muss *C* ausgewählt werden, womit diese Aufgabe auch ausgeführt wird. Der Fall, dass die optionale Aufgabe *C* aber nicht ausgeführt werden soll, wird in diesem Modell nicht abgedeckt.

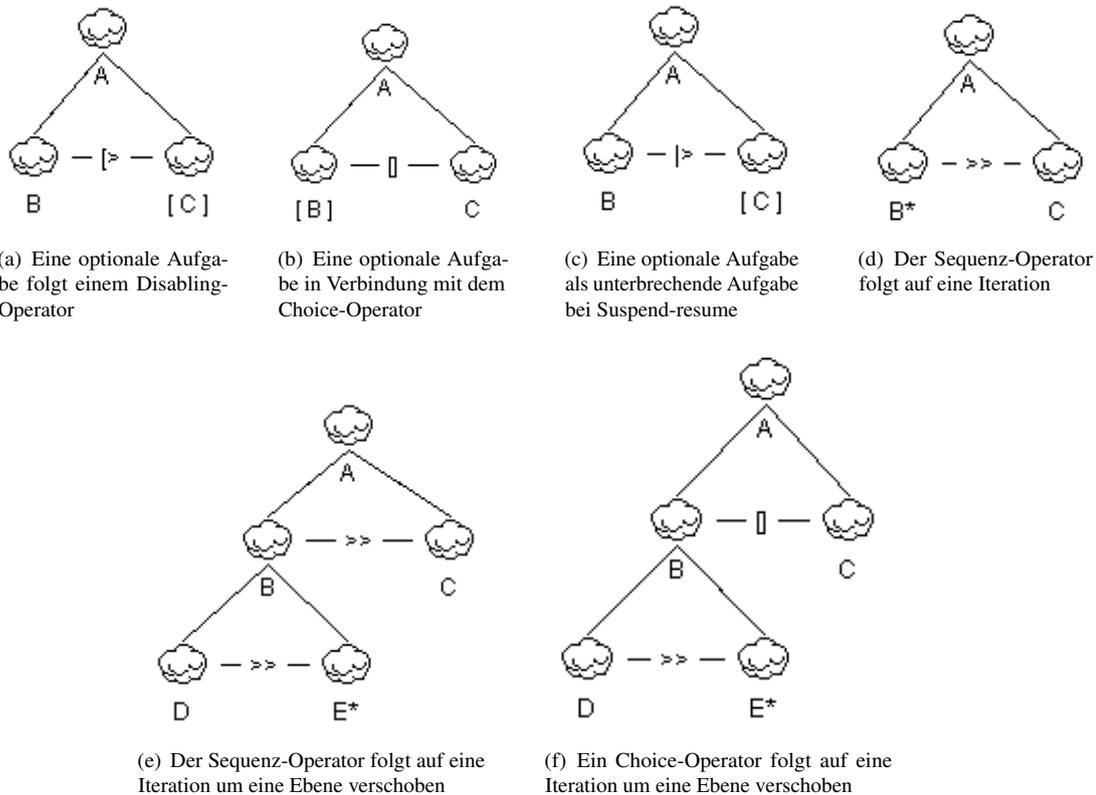


Abbildung 5.3: Problematische CTT-Modelle

Kein Kontrollflussproblem, dafür aber ein unschönes Modell ist in Abbildung 5.3(c) in Verbindung mit einer optionalen Aufgabe und dem Suspend-resume Operator zu sehen. Die unterbrechende Aufgabe C ist dort als optional spezifiziert. Jedoch ist durch den Suspend-resume Operator bereits implizit angegeben, dass C nicht zwangsläufig ausgeführt werden muss. Sobald B beendet wurde steht die Möglichkeit der Ausführung von C nicht mehr bereit. Zudem ist sobald C die laufende Aufgabe B unterbricht diese nicht mehr optional. Die Semantik von Suspend-resume steht im Kontrast zum Disabling Operator, bei der auch nach Abarbeitung der Vorgängeraufgabe, die Nachfolgeraufgabe weiterhin abzuarbeiten bleibt. Hierbei weicht CTT von der ursprünglichen LOTOS-Semantik ab [BB89].

Ein weiteres Problem ist in Abbildung 5.3(d) angegeben. Dort wird eine Iteration im Zusammenhang mit einem Sequenz-Operator verwendet. Wie in Abschnitt 5.2.4 bereits erwähnt, ist dieses bei CTT nicht zulässig. Für jeden Iterationszyklus wird eine Entscheidung erwartet, ob ein weiterer Iterationszyklus erfolgen oder mit der folgenden Aufgabe fortgefahren werden soll. Die Iteration in Verbindung mit dem Choice-Operator ist in Tabelle 5.3 angegeben.

Weitere Probleme treten auf, wenn über Hierarchieebenen hinweg temporale Beziehungen zu Aufgaben aufgebaut werden. Die Abbildungen 5.3(f) und 5.3(e) verdeutlichen dieses in Verbindung mit Iterationsoperatoren. Das Modell von Abbildung 5.3(e) ist aus den Gründen, die zu dem Modell von Abbildung 5.3(d) angeführt wurden ebenfalls nicht korrekt.

Ein Modell, das zwar laut CTTE zulässig, aber trotzdem problematisch ist, ist in Abbildung 5.3(e) zu sehen. Im Vergleich zur Iteration in Verbindung mit dem Choice-Operator von Tabelle 5.3, ist die Iteration hier eine Hierarchieebene tiefer angegeben. Die Entscheidung, ob ein neuer Iterationszyklus von E begonnen oder mit Aufgabe C fortgefahren werden soll, ist also über eine Hierarchieebene verschränkt. Betrachtet man

den Choice-Operator zwischen B und C würde dieser mehrfach ausgewertet werden. Da in dieser Ebene aber keine Iteration spezifiziert ist, würde dieses Modellverhalten zu einem semantischen Problem führen. In der Ebene ist der Choice einmal zu vollziehen. Außerdem wäre eine Transformation dieses Modells zu einem Aktivitätsdiagramm nicht möglich. Die Kontrollflusskante ausgehend von E zum Choice-Operator, der entscheidet, ob E oder C auszuführen ist, müsste über die Aktivitätsgrenze der Aktivität B gehen. Ein solches Modell ist jedoch bei Aktivitätsdiagrammen nicht erlaubt [UML10].

Die in diesem Abschnitt aufgezeigten Konsistenzprobleme verdeutlichen, dass eine genauere Spezifikation der CTT-Aufgabenmodelle und eine bessere Toolunterstützung während der Designzeit wünschenswert ist. CTTE bietet zwar Möglichkeiten zur Konsistenzprüfung der Modelle, diese sind aber nicht komplett, wie Beispiele aus diesem Abschnitt gezeigt haben. Außerdem gibt es keine verlässlichen Spezifikationen, die die Soundness-Eigenschaften für CTT-Modelle festlegen. Einen transparenten Ansatz zur Definition der Soundness-Eigenschaften bietet ein metamodellierter Ansatz, der in Abschnitt 5.3.2 vorgestellt wird.

5.3.2 Metamodell für MCTT

In diesem Abschnitt wird in Abbildung 5.4 das Klassendiagramm vorgestellt, das die Basis zur Definition der Sprache *Metamodel-Based ConcurTaskTree* (MCTT) liefert. Diese Sprache ist wie der Name schon verrät stark angelehnt an CTT und soll deren Eigenschaften und Semantiken beinhalten.

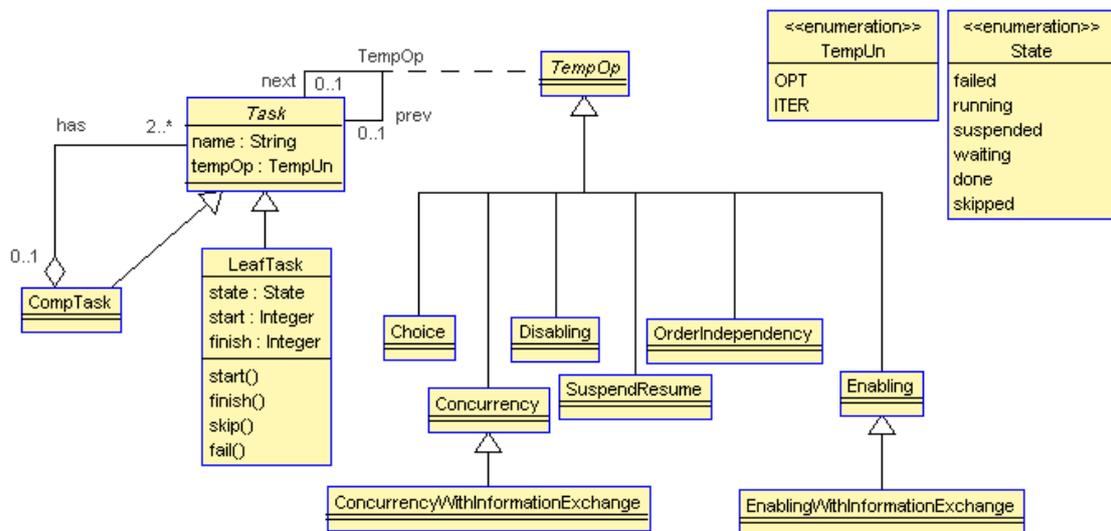


Abbildung 5.4: Das Metamodel für den MCTT-Ansatz

Im Klassendiagramm ist die abstrakte Klasse *Task* zu sehen. Sie beinhaltet das Attribut *name*, mit der das Aufgabenobjekt bezeichnet werden kann. Alternativ kann wie bei dem DMWM-Modell von Abbildung 3.9 auch bei MCTT die Objekt-ID zur Bezeichnung verwendet werden. Des Weiteren ist mit dem Attribut *tempOp* die Möglichkeit gegeben, der Aufgabe einen unären temporalen Operator zuzuweisen. In der Enumeration *TempUn* sind dafür die zwei möglichen Operatoren angegeben. Der Eintrag *OPT* steht für optional auszuführende und *ITER* für mehrfach auszuführende, iterative Aufgaben.

Die abstrakten Aufgaben gliedern sich auf in innere Knoten und Blattknoten. Die Klasse *CompTask* repräsentiert die inneren Knoten, wohingegen die Blattknoten durch die Klasse *LeafTask* spezifiziert werden.

Die inneren Knoten sind mit mindestens 2 Kinderaufgaben assoziiert, was durch die Assoziation *has* und die Multiplizität 2..* ausgedrückt wird.

Zusätzlich zur Dekomposition ist die Angabe der temporalen Operatoren Kernmerkmal der CTT-Aufgabenmodelle. Diese Beziehung zwischen Aufgaben wird im Metamodell über die abstrakte Assoziationsklasse *TempOp* ausgedrückt. CTT sagt aus, dass Nachbarknoten in eine temporale Beziehung gesetzt werden müssen. Über die dafür verwendete Assoziationsklasse *TempOp* werden die Nachbarknoten im Baum definiert. Aufgaben, die mit keinem *prev*-Objekt in Beziehung stehen, sind die ersten Aufgaben in der nächst unteren Ebene des Teilbaums. Am CTT-Beispiel von Abbildung 5.3(e) sind das die Knoten *A*, *B* und *D*. Analog wird die letzte Aufgabe in der unterliegenden Hierarchie dadurch identifiziert, dass kein *next*-Objekt existiert. Die Aufgaben dazwischen müssen jeweils mit *pred*- und *next*-Objekten über *TempOp* verbunden sein. In Abschnitt 5.3.3 wird zum Definieren dieser Eigenschaft eine OCL-Invariante für das MCTT-Metamodell definiert.

Für die Spezifikation der temporalen Beziehungen müssen die Unterklassen der abstrakten Assoziationsklasse *TempOp* verwendet werden. Hierfür gibt es bei CTT die acht in Tabelle 5.1 vorgestellten binären Operatoren. Von *Choice* bis *Enabling* sind in Abbildung 5.4 die Klassen von links nach rechts in der Bindungsreihenfolge angegeben, wie in Abschnitt 5.2.2 beschrieben. Für MCTT gilt diese Reihenfolge ebenso.

Im Gegensatz zu den durch die Klasse *CompTask* repräsentierten inneren Aufgaben sind die mit *LeafTask* modellierten Blattaufgaben ausführbar. Dieses wird durch die in der Klasse *LeafTask* angegebenen Operationen ausgedrückt, die analog zu DMWM (s. Abschnitt 3.2.1) die Schnittstelle eines Aktivitätsobjekts zum Nutzer darstellen.

Bezüglich der operationalen Semantik unterscheidet sich MCTT von CTT, indem Zustandsautomaten zur Definition der operationalen Ablaufsemantik verwendet werden. Der Ansatz ist analog zu dem in Abschnitt 3.2 vorgestellten von DMWM zu sehen. Diese Verwendung ist nötig, um eine ähnliche Betrachtung und Benutzung wie bei Workflow Management Systemen herzustellen [Hol98]. Dabei werden dem Anwender in einer Art Arbeitsliste die Tätigkeiten angezeigt, die er starten und beenden kann. Während der Abarbeitung muss der Nutzer dann die entsprechenden Aktionen zur Erledigung der Aufgabe ausführen. Nach Fertigstellung hat der Nutzer die Aufgabe dann zu beenden. CTT fußt im Gegensatz zu Automaten auf der ereignisbasierten Prozessalgebra LOTOS [BB89].

Für die ausführbaren Tätigkeiten bzw. Aufgaben ist im Metamodell von Abbildung 5.4 die Klasse *LeafTask* abgebildet. Wie bei DMWM werden dem Nutzer die Operationen *start()*, *finish()*, *skip()* und *fail()* als Schnittstelle zur Benutzung der Aktivitäten (bzw. Blattaufgaben) bereitgestellt. Mit diesen Operationen werden die Ausführungszustände der Blattaufgaben verändert, die mit dem Attribut *state* in der Klasse *LeafTask* gespeichert werden. Die möglichen Ausführungszustände dafür sind in der Enumeration *State* zu sehen. Sie repräsentieren die Zustände, die in den Zustandsautomaten zur Spezifikation der Lebenszyklen der Blattaufgaben verwendet werden. Die Zustandsautomaten sind in der Abbildung 5.5 angegeben.

Wie bei DMWM wird auch bei MCTT für sowohl die Designzeit als auch die Runtime das gleiche Metamodell verwendet. Über die Einbeziehung der Ausführungszustände lässt sich die operationale Semantik von MCTT mit Hilfe von OCL plattformunabhängig im Metamodell ausdrücken. Die imperative Umsetzung ist dann wiederum plattformabhängig und wird mit der *Simple OCL-based Imperative Language* (SOIL) in USE umgesetzt. Diese Thematik wird in Abschnitt 6.2.1 näher vorgestellt.

Im MCTT-Metamodell wird die Dekomposition der Aufgaben über die *has* Assoziation ausgedrückt. Die inneren Knoten sind vom Typ *CompTask* und die ausführbaren Blattknoten vom Typ *LeafTask*. Die integrierte Zielmodellierung bei CTT wird über die inneren Aufgaben spezifiziert, die die Kontextziele ausdrücken.

Wie bereits in Abschnitt 2.5.3 angesprochen, hat das ARIS-Metamodell für die Funktionssicht von Abbildung

2.6 starke Ähnlichkeiten zum MCTT-Metamodell. Die Dekomposition der Funktionen bzw. Aufgaben, die bei den EPKs bzw. in ARIS synonym bezeichnet werden [KNS92], wird im ARIS-Metamodell mit der Assoziationsklasse *FUNKTIONSSTRUKTUR* bezeichnet. Die sequenzielle Ablaufspezifikation wird mit der Assoziationsklasse *ANORDNUNG* angegeben, wofür bei MCTT *Enabling* verwendet wird. Beim MCTT-Metamodell gibt es zusätzlich noch fünf weitere von der Ablaufsemantik unterschiedliche temporale Beziehung, die bereits erklärt wurden.

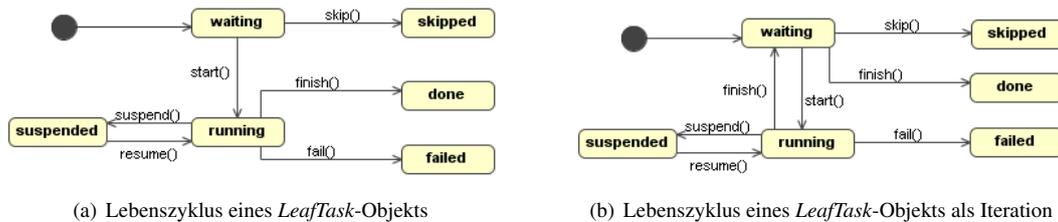


Abbildung 5.5: Lebenszyklen von Aufgabenobjekten modelliert in UML-Zustandsautomaten

Die in Abbildung 5.5 vorgestellten Zustandsdiagramme unterscheiden sich von denen des DMWM-Ansatzes (s. Abbildung 3.3), indem ein zusätzlicher Zustand *suspended* eingeführt und die Transition mit *nextIteration()* zum Rücksetzen der Aktivitätszustände weggelassen wurde. Der neue Zustand *suspended* ist notwendig, um den temporalen CTT-Operator *SuspendResume* umzusetzen. In Abbildung 5.5(a) ist der Lebenszyklus eines normalen *LeafTask*-Objekts angegeben. Dieser Lebenszyklus gilt, wenn das Aufgabenobjekt selber und keines seiner Oberaufgaben eine Iterationsmarkierung hat. Sollte hingegen bei der Blattaufgabe oder einer ihrer Oberaufgaben das Attribut *tempOp* mit *ITER* belegt sein, gilt der Lebenszyklus von Abbildung 5.5(b).

Analog zu dem DMWM-Ansatz (s. Abschnitt 3.2.1) werden OCL-Vor- bzw. Nachbedingungen an den Operationen in der Klasse *LeafTask* verwendet, um die Zustandsautomaten umzusetzen. Im Zusammenhang mit diesen Bedingungen muss geprüft werden, ob die aktuelle Aufgabe oder eine ihrer Oberaufgaben eine Iteration oder eine normale Aufgabe ist. Bei einer normalen Aufgabe ist entsprechend der Zustandsautomat von Abbildung 5.5(a) umzusetzen und im anderen Fall der von Abbildung 5.5(b).

5.3.3 OCL-Invarianten zum Prüfen von strukturellen Eigenschaften

Für eine Anwendung der CTT-Aufgabenmodelle in der Workflowmodellierung ist es notwendig, eine verlässliche Prüfung der Konsistenzigenschaften der Modelle zur Designzeit zu haben. Hier bieten die etablierten Sprachen wie Petri-Netze und Tools dafür Soundness-Prüfungen an, wohingegen Aufgabenmodelle noch Schwächen aufweisen, wie in Abschnitt 5.3.1 vorgestellt. Um entsprechende Prüfungen auch für Aufgabenmodelle bereitzustellen, werden in diesem Abschnitt OCL-Konsistenzbedingungen definiert, die im Metamodell hinterlegt und bereits während des Modellierens zur Designzeit vom UML-Tool geprüft werden. Auf Verletzungen wird der Modellierer unmittelbar hingewiesen.

Zunächst kann die baumartige Modellierung anhand von OCL-Invarianten sichergestellt werden. Alle Nachbarn im Aufgabenbaum müssen zudem über binäre temporale Beziehungen verbunden sein. Diese grundlegende Eigenschaft für CTT-Aufgabenbäume wird in den ersten zwei OCL-Invarianten von Listing 5.1 festgelegt.

Invariante *taskStructure* ist der Assoziationsklasse *TempOp* zugeordnet und prüft die Nachbarschaft der mit temporalen Operatoren verbundenen Aufgaben. Dieses wird sichergestellt, indem der Vaterknoten der linken mit dem der rechten Aufgabe auf Gleichheit überprüft wird. Die Invariante *startTaskAndEndTaskExists*

stellt des Weiteren sicher, dass genau eine Aufgabe mit keiner Vorgängeraufgabe verbunden und damit Startknoten ist. Analog dazu muss ein Endknoten existieren. Alle weiteren Aufgaben müssen in einer Kette über die Assoziationsklasse *TempOp* verbunden sein, da durch die Invariante die Multiplizitätsbedingung *l* sowohl für die Rolle *prev* als auch für die Rolle *next* gilt.

```

1 context TempOp inv taskStructure:
2   prev.compTask = next.compTask
3
4 context CompTask inv startTaskAndEndTaskExist:
5   task->select(prev.isUndefined())->size()=1 and
6   task->select(next.isUndefined())->size()=1
7
8 context Disabling inv noDisablingWithOptionalTasks:
9   next.hasNonOptionalLeafTaskDisabling()
10
11 context Choice inv noChoiceWithOptionalTasks:
12   prev.hasNonOptionalLeafTaskChoicePredecessor() and next.hasNonOptionalLeafTaskChoiceSuccessor()

```

Listing 5.1: OCL-Invarianten zur Festlegung der strukturellen Eigenschaften von Aufgabenbäumen

Zwei Invarianten, die die Konsistenzprobleme von Abbildung 5.3(a) und 5.3(b) identifizieren sollen, sind in den zwei unteren Invarianten von Listing 5.1 angegeben. Beide verwenden Operationsaufrufe, die seiten-effektfreie OCL-Abfragen repräsentieren. Der Operationsaufruf *hasNonOptionalLeafTaskDisabling* geht vom Disabling-Operator zum nächsten Aufgabenobjekt und traversiert die Unteraufgaben abhängig von den temporalen Operatoren und deren Bindungsstärke. Die relevanten Aufgaben dürfen keine Optionalitätseigenschaft besitzen.

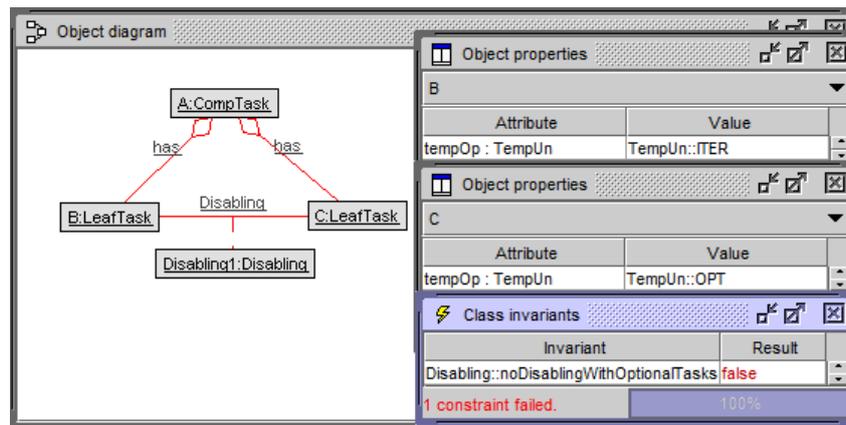
Für die zweite Invariante werden die OCL-Operationen *hasNonOptionalLeafTaskChoicePredecessor* und *hasNonOptionalLeafTaskChoiceSuccessor* verwendet. Entsprechend der temporalen Beziehungen und deren Bindungsstärke werden auch hier die Unteraufgaben in der Baumhierarchie traversiert. Die relevanten Aufgaben werden auf die Optionalitätseigenschaft geprüft, die nicht gesetzt sein darf. Auf die genaue Umsetzung der OCL-Abfrageoperationen wird hier nicht eingegangen.

Im folgenden wird das USE-Tool analog zu DMWM in Abschnitt 3.5.3 eingesetzt, um MCTT-Aufgabenmodelle zu modellieren. Zwei inkonsistente MCTT-Aufgabenmodelle sind in Abbildung 5.3 angegeben. Dort ist zu sehen, dass die Inkonsistenzen durch die OCL-Invariantenüberprüfung unmittelbar während der Design-time angezeigt werden und der Modellierer darauf hingewiesen wird. Werkzeuge zum Identifizieren der Ursachen der Inkonsistenzen bei DMWM-Modelle wurden in Abschnitt 3.5.4 präsentiert. Diese können ebenso bei MCTT eingesetzt werden.

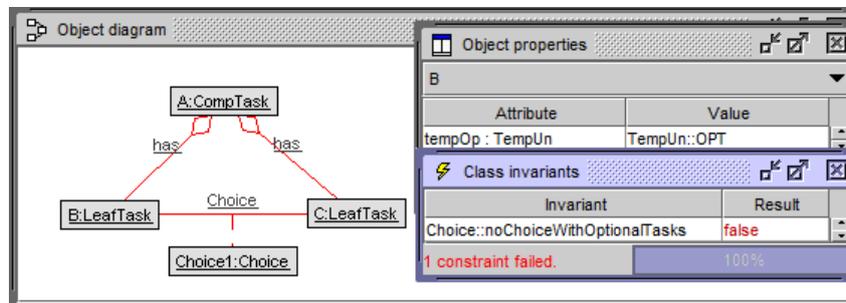
In Abbildung 5.6(a) ist ein MCTT-Modell von Abbildung 5.3(a) zu sehen, bei dem die abbrechende Aufgabe *C* als optional spezifiziert ist. Aus Gründen, die bereits in Abschnitt 5.3.1 erklärt wurden, ist dieses Modell nicht sound.

Das USE-Tool evaluiert permanent die OCL-Invarianten und erkennt in Abbildung 5.6(a), dass die Invariante *noDisablingWithOptionalTasks* verletzt wurde. Wie in Abschnitt 3.5.4 beschrieben, können hier Hilfsmittel und Tools eingesetzt werden, um den Grund der Invariantenverletzung und involvierten Modellobjekte herauszufinden.

Ein weiteres Beispiel ist in Abbildung 5.6(b) angegeben. Dafür wurde das CTT-Modell von Abbildung 5.3(b) herangezogen und nachmodelliert. Aus Gründen, die ebenfalls bereits in Abschnitt 5.3.1 vorgestellt wurden, ist das Modell inkonsistent. In Abbildung 5.6(b) ist des Weiteren zu sehen, dass die Invariantenverletzung von *noChoiceWithOptionalTasks* angezeigt wird. Der Fehler ist damit erkannt worden und kann vom Modellierer



(a) Eine optionale Aufgabe folgt auf einen Disabling-Operator in einem MCTT-Modell



(b) Eine optionale Aufgabe in Verbindung mit einem Choice-Operator

Abbildung 5.6: Inkonsistente MCTT-Aufgabenmodelle

daraufhin behoben werden.

5.3.4 Metamodellbasierte Spezifikation von MCTT-Modellen

In diesem Abschnitt wird das in Abschnitt 3.5.3 vorgestellte Notfallprozessbeispiel für die hierarchische MCTT-Modellierung angewendet. Der dazugehörige MCTT-Aufgabenbaum ist in Abbildung 5.7 zu sehen. Bei der Notation ist anzumerken, dass die Baumdarstellung um 90 Grad im Vergleich zur CTTE-Darstellung gedreht und zudem gespiegelt wurde. Die CTTE-Aufgabenmodelle hatten die Blätter unten angeordnet und sie waren von links nach rechts zu lesen. Dagegen hat der Aufgabenbaum in der MCTT-Darstellung die Blätter rechts und sie sind von oben nach unten zu lesen. Bei großen Prozessmodellen hat diese Darstellung deutliche Vorteile, da die Aktivitätsbezeichnungen untereinander notiert deutliche Platzersparnisse hervorrufen. Eine ähnliche Darstellung wurde bei dem *Groupware Task Analysis*-Ansatz (GTA) gewählt [VLB96, VWC02].

Der Wurzelknoten im Modell von Abbildung 5.7 bezeichnet den Gesamtprozess mit *EmergencyProcess*. Dieser gliedert sich auf in *TransportSurgery* und *AdjustMedication*. Mit dem temporalen Operator *SuspendResume* wird verdeutlicht, dass die Aufgabe *TransportSurgery* jederzeit und mehrfach unterbrochen werden kann, solange diese läuft. Mit der unterbrechenden Aufgabe *AdjustMedication* wird die Medikation des Patienten durchgeführt und protokolliert.

Die Aufgabe *TransportSurgery* wird dekomponiert in die Aufgaben *Transport* und *SurgeryCheck*. Für den Transport des Patienten stehen die beiden Aufgaben *AmbulanceDelivery* und *HelicopterDelivery* zur Auswahl bereit. Wie bereits in Abschnitt 5.2.2 diskutiert, wird hier eine implizite Entscheidungsmodellierung,

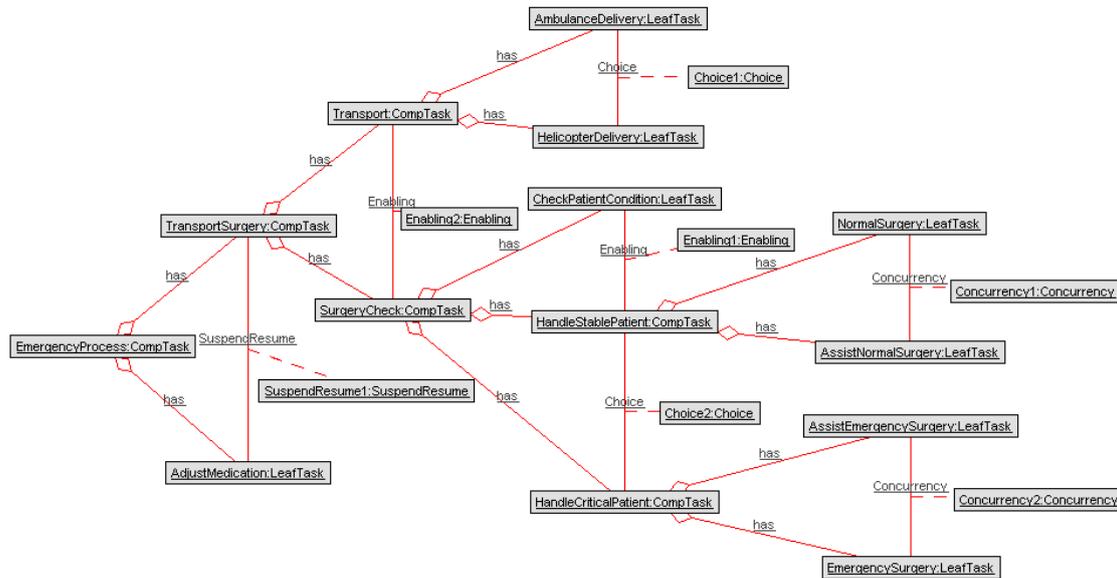


Abbildung 5.7: MCTT-Prozessmodell für die Notfallaufnahme

die auch als Deferred Choice bezeichnet wird (s. Abschnitt 3.2.4.4), verfolgt. D.h. mit dem Starten einer Aufgabe wird die andere implizit deaktiviert.

Die Aufgabe *SurgeryCheck* wird in die Unteraufgaben *CheckPatientCondition*, *HandleStablePatient* und *HandleCriticalPatient* aufgegliedert. Durch die schwächere Bindung des *Enabling*-Operators ist die Aufgabe *CheckPatientCondition* in diesem Teilbaum zuerst auszuführen. Zwischen den zwei folgenden Aufgaben ist eine implizite Entscheidung durchzuführen. Eine explizite Entscheidungsmodellierung, in der eine Entscheidungsaktivität und die Kriterien zur Auswahl spezifiziert werden können, ist in CTT nicht vorgesehen. Dieses wäre analog zum DMWM-Ansatz für das angegebene Beispiel wünschenswert (s. Abschnitt 3.5.3). Im MCTT-Modell fehlt die Angabe, dass in *CheckPatientCondition* die Entscheidung zu treffen ist. In Abschnitt 5.4.2 wird eine explizite Möglichkeit mit der datenbasierten Entscheidungsmodellierung für MCTT eingeführt.

Die beiden Aufgaben zum Behandeln des Patienten zum einen für den Patienten im kritischen und zum anderen im stabilen Zustand beinhalten jeweils zwei Blattaufgaben. Es ist jeweils eine Operationsaufgabe und eine Assistenzaufgabe angegeben. Diese sind mit dem *Concurrency*-Operator verbunden und nebenläufig auszuführen. Eine Angabe, dass die Operations- und Assistenzaufgabe garantiert parallel geschehen müssen, wie es beim DMWM-Ansatz möglich ist (s. Abschnitt 3.5.3), ist bei CTT nicht spezifizierbar.

Vergleicht man die Klassenbezeichnungen vom MCTT-Metamodell zum DMWM-Metamodell ist zu erkennen, dass die ausführbaren Aktivitäten bzw. Aufgaben unterschiedlich benannt sind. Ausführbare Aktivitäten werden bei MCTT mit *LeafTask*, bei DMWM hingegen mit *Activity* bzw. deren Unterklassen bezeichnet. Die Verschachtelung von Aktivitäten über die Klasse *CompTask* ist bei MCTT im Gegensatz zu DMWM unbeschränkt möglich. DMWM steht nur eine Verschachtelung über höchstens eine Ebene anhand einer Gruppenaggregation von Aktivitäten mit der Klasse *Group* bereit. Es ist dafür bei DMWM möglich, wie in Abbildung 3.10 angegeben, eine Aktivität mehreren Gruppen zuzuordnen. Bei MCTT sind Aufgaben über die Assoziation *has* exklusiv einer übergeordneten Aufgabe zugeordnet.

5.4 Entscheidungsmodellierung und Datenaspekte in Aufgabenmodellen

Bei Geschäftsprozessmodellen spielt u.a. die Entscheidungsmodellierung und die Datenintegration eine große Rolle, wie in Abschnitt 2.5.2 erwähnt wurde. Diese Aspekte werden im Folgenden für Aufgabenmodelle weiter eingeführt.

Bei der Entscheidungsmodellierung in Geschäftsprozessmodellen soll der Anwender bei entsprechenden Situationen unterstützt werden, den korrekten Pfad im Prozessmodell zu wählen. Bei EPKs wird eine explizite Entscheidungsmodellierung gefordert, indem vorgeschrieben wird, dass vor einem XOR- oder OR-Splitkonnektor eine Funktion stehen muss. Der Konnektor ist also nicht mehr isoliert zu betrachten, sondern es sind die davorstehende Aktivität und die dahinterstehenden Ereignisse inkl. ihrer Bezeichnungen für die Entscheidungsmodellierung von Bedeutung (vgl. Abschnitt 2.5.2.1). Eine implizite Variante, die bei den Workflow Patterns als *Deferred Choice* bezeichnet wird, ist bei Geschäftsprozessmodellen, insbesondere bei EPKs, eher die Ausnahme. Diese ist dagegen, wie in Abschnitt 5.3.2 schon erwähnt, bei CTT-Aufgabenmodellen die bisher einzige Variante. Wenn man Aufgabenmodelle zur Geschäftsprozessmodellierung einsetzen möchte, ist eine explizite Variante zur Entscheidungsmodellierung dagegen notwendig. In Abschnitt 5.4.1 wird dafür ein bestehender Entscheidungsformalismus, der zur Konfiguration von Aufgabenmodellen eingeführt wurde, für eine explizite Entscheidungsmodellierung adaptiert. Ein Ansatz, der eher an der Entscheidungsmodellierung von BPMN orientiert ist, wird in Abschnitt 5.4.2 mit einem datenbasierten *Choice* eingeführt. Hierbei ist die Integration eines Datenmodells notwendig, das im gleichen Abschnitt eingeführt wird.

5.4.1 Explizite Entscheidungsmodellierung mit Decisionnodes

Wie bereits in Abschnitt 5.1.2 erwähnt, werden Aufgabenmodelle vorwiegend in dem Bereich der User Interface-Entwicklung eingesetzt. In diesem Zusammenhang wurden *Decisionnodes* (Entscheidungsknoten) für CTT-Aufgabenmodelle eingeführt, um die Modelle an bestimmte Ausführungskontexte anpassen zu können [Luy04]. Mit dem Auswerten dieser Knoten werden die Modelle für den Einsatzzweck und den Kontext konfiguriert. Für die explizite Entscheidungsmodellierung wird dieser Formalismus in Abbildung 5.8 adaptiert.

Bei konfigurierbaren Geschäftsprozessmodellen spricht man von Referenzprozessmodellen [RA07]. Sie enthalten analog zu den *Decisionnodes* bei CTTs, Entscheidungsoperatoren, die vor der Ausführung auszuwerten sind und damit das Prozessmodell konfigurieren. Referenzmodelle werden üblicherweise für mehrere Unternehmen in bestimmten Branchen entwickelt. Für die konkreten Unternehmen und Einsatzzwecke werden diese Modelle dann angepasst. Die konfigurierbaren EPKs werden u.a. dafür eingesetzt, die SAP-Software an unternehmensspezifische Abläufe anzupassen [Sch01].

Sowohl bei Referenzprozessmodellen als auch bei Aufgabenmodellen mit *Decisionnodes* hat die Konfiguration vor der Ausführung der Modelle zu geschehen. Es muss also vor der Runtime die Entscheidung getroffen werden, welcher Zweig unter dem *Decisionnode* bei CTTs bzw. hinter dem Entscheidungsknoten bei C-EPCs auszuwählen ist. Für eine explizite Entscheidungsmodellierung in CTT wird der Formalismus adaptiert, indem die *Decisionnodes* hier ausführbare Entscheidungsaktivitäten repräsentieren, die zur Runtime abzuarbeiten sind.

Die Entscheidungsmodellierung wird in Abbildung 5.8 verdeutlicht. *Decisionnodes* sind innere Knoten im Aufgabenbaum und müssen entgegen der eigentlichen CTT-Philosophie vom Anwender zur Runtime

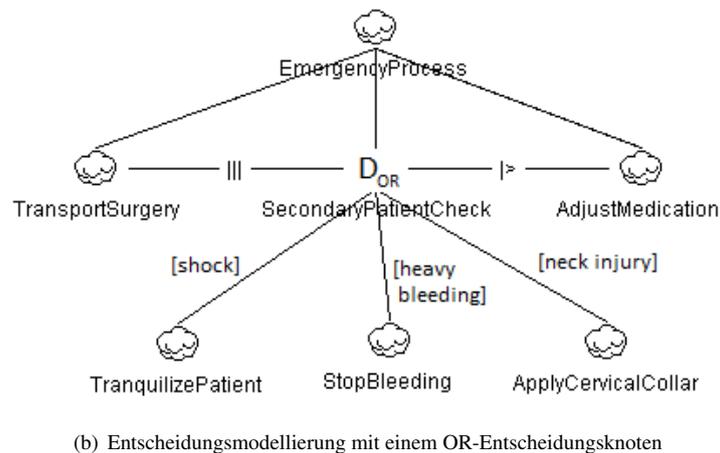
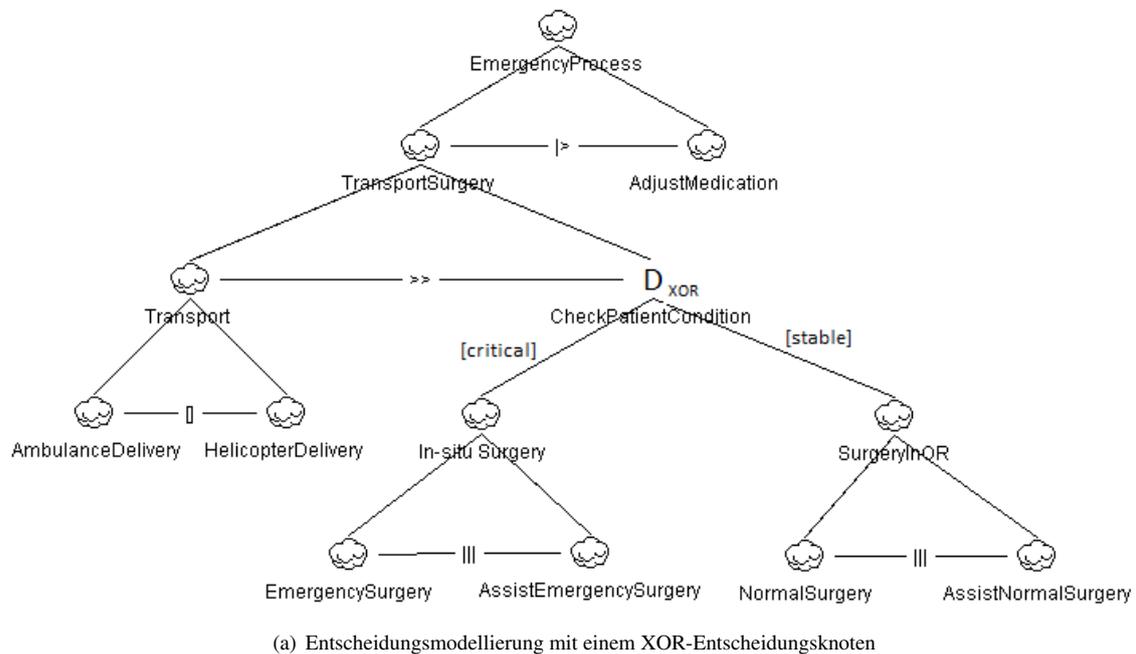


Abbildung 5.8: Entscheidungsknoten zur expliziten Entscheidungsmodellierung in CTT-Aufgabenmodellen

ausgeführt werden. Im Decisionnode ist die Aktivität zu bezeichnen, mit der der Nutzer die Entscheidung zu treffen hat. An den Kanten ausgehend vom Entscheidungsknoten werden die Kriterien zur Auswahl der Alternativen modelliert.

In Abbildung 5.8 wird die Entscheidungsmodellierung mit Entscheidungsknoten an zwei Modellen verdeutlicht. Für das Beispiel von Abbildung 5.8(a) wurde das Notfallprozessmodell von Abschnitt 5.3.4 angepasst. Die Entscheidungsaktivität *CheckPatientCondition* wurde von einem Blattknoten in einen Decisionnode im inneren des Baumes umgewandelt. Die Kriterien zur Auswahl und die damit verbundenen Folgeaktivitäten wurden an den von dem Decisionnode ausgehenden Kanten modelliert. Der Patient ist in der Aktivität *CheckPatientCondition* zu untersuchen und abhängig, ob er sich in einem kritischen (*critical*) oder einem stabilen (*stable*) Zustand befindet ist eine Notoperation oder eine normale Operation durchzuführen. Hier ist eine exklusive Entscheidung zu treffen, d.h. entweder ist eine Notoperation (*In-situ Surgery*) oder eine normale Operation in einem Operationsraum mit steriler Umgebung (*SurgeryInOR*) durchzuführen.

In Abbildung 5.8(b) wurde die erste Ebene vom Aufgabenbaum aus Abbildung 5.8(a) um einen Decisionnode erweitert. Es wurde eine inklusive *OR*-Entscheidung mit dem Decisionnode *SecondaryPatientCheck* spezifiziert. Damit ist mindestens eine der darunterstehenden Alternativen auszuwählen. Es dürfen mehrere bzw. alle drei Alternativen *shock*, *heavy bleeding* und *neck injury* zutreffen, womit die zugeordneten Aktivitäten dann auszuführen sind. Die inklusive *OR*-Entscheidung ist seit den EPKs sehr verbreitet in der Geschäftsprozessmodellierung und beispielsweise auch bei BPMN, YAWL und ADEPT integriert worden.

Ein Nachteil der hier vorgestellten Decisionnodes zur expliziten Entscheidungsmodellierung besteht in der Verletzung einer Modellierungsvorschrift von CTT. Sie besagt, dass innere Knoten nicht ausführbar sind und die Hierarchie nur zur Aufgabendeckomposition eingesetzt werden darf. Diese beiden Eigenschaften werden von den Decisionnodes in den Aufgabenmodellen verletzt. Es gibt aber auch Vorteile für die explizite Entscheidungsmodellierung mit Decisionnodes. Die Modelle bleiben durch die beibehaltene Baumstruktur strukturiert und daher übersichtlich. Außerdem ist diese Art der Entscheidungsmodellierung exklusiver Bestandteil der Prozessmodellierungssprache und es wird keine Zuhilfenahme eines Datenmodells benötigt. Dieses hat eine unkompliziertere Benutzbarkeit zur Folge. Eine kompliziertere Art der Entscheidungsmodellierung führt Abschnitt 5.4.2 mit dem datenbasierten Choice ein.

Die in diesem Abschnitt vorgestellten Erweiterungen wurden nicht metamodellbasiert umgesetzt, sondern nur an dem Aufgabenmodell-basierten Workflow Management System *TTMS*, das in Abschnitt 6.3 näher eingeführt wird. Daher wird hier auf die Vorstellung der Metamodellerweiterung verzichtet.

5.4.2 Datenbasierte Entscheidung und Datenintegration bei MCTT

Ein Ansatz, der eine explizite Entscheidungsmodellierung ermöglicht und die CTT-Modellierungsvorschrift berücksichtigt, dass nur Blattknoten ausgeführt werden dürfen, wird in diesem Abschnitt eingeführt. Dafür wurde ein datenbasierter binären Entscheidungsoperator analog zum bereits bekannten und in Abschnitt 5.2.2 beschriebenen *Choice*-Operator entwickelt.

Der Ansatz basiert auf dem MCTT-Metamodell, auf dessen Erweiterung in Abschnitt 5.4.2.1 näher eingegangen wird. Eine weitergehende Datenmodellintegration durch eine weitere MCTT-Metamodellerweiterung wird in Abschnitt 5.4.2.2 folgen. Die Anwendung des Metamodells zur Workflowmodellierung mit datenbasiertem Entscheidungsoperator und Datenintegration wird in Abschnitt 5.4.2.3 beschrieben.

5.4.2.1 Metamodellerweiterung für die Entscheidungsmodellierung

Die Metamodellerweiterung für MCTT ist in Abbildung 5.9 zu sehen. Verglichen zum Metamodell von Abbildung 5.4 sind die Klassen *DataChoice* und die abstrakte Klasse *DecisionObject* hinzugekommen. Die Klasse *DataChoice* beinhaltet die Attribute *leftCondition* und *rightCondition*, in denen die Kriterien zur Pfadauswahl spezifiziert werden sollen. Sie greifen auf Attribute des über *basedOn* verbundenen Entscheidungsobjekts (*DecisionObject*) zu. Dort repräsentieren die String-Attribute OCL-Terme, die einen Booleanwert zurückliefern sollen. Die OCL-Terme werden vom MCTT-Plugin für USE (in Abschnitt 6.2.2 näher vorgestellt) zur Runtime ausgewertet.

Die Abbildung 5.9 ist in zwei Teile geteilt. Erstens existiert in dem UML-Klassendiagramm das CTT-Metamodell und zweitens das Datenmodell. Die abstrakte Klasse *DecisionObject* ist noch dem CTT-Metamodell zugeordnet. Die konkrete Klasse *PatientData* ist dagegen Bestandteil des Datenmodells. Über eine Generalisierungsbeziehung zur Klasse *DecisionObject* wird die Verbindung vom CTT-Metamodell zum Datenmodell hergestellt. Dadurch können Objekte vom Typ *PatientData* auch im Prozessmodell verwendet werden. In analoger Weise wurden bereits in Abschnitt 3.3.2 die Datenflussobjekte (*DataflowObject*) in das

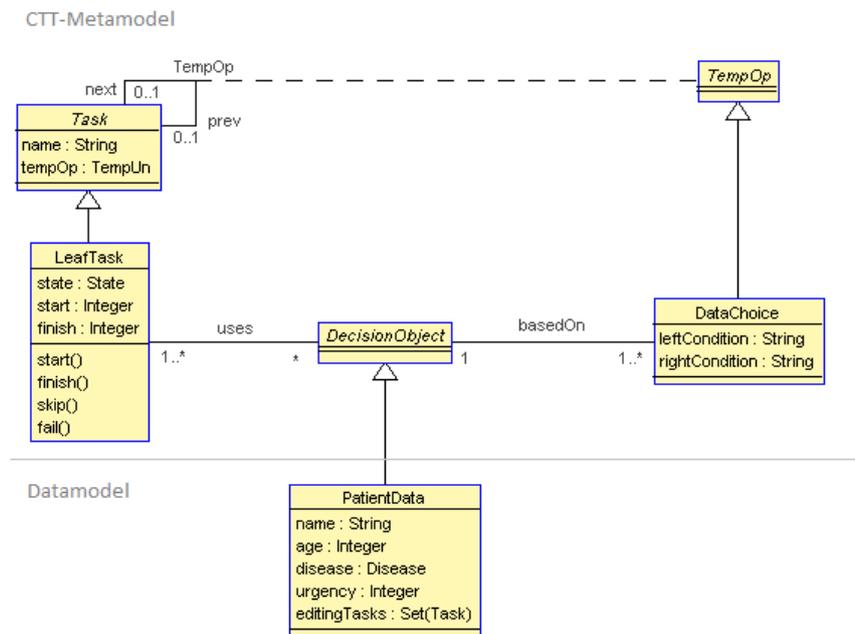


Abbildung 5.9: MCTT-Metamodellerweiterung für DataChoice

DMWM-Metamodell integriert.

Die Aktivität, die die Entscheidung zur Pfadauswahl trifft ist mit der Assoziation *uses* mit dem Entscheidungsobjekt verbunden. Sobald diese beendet wird, wird die Entscheidung auf Basis der Daten, die für das Entscheidungsobjekt eingetragen wurden, getroffen. Dafür werden die Bedingungen *leftCondition* und *rightCondition* der Klasse *DataChoice* ausgewertet. Wird die Bedingung *leftCondition* zu *false* ausgewertet, so wird der mit *prev* verbundene Teilbaum übersprungen. In analoger Weise wird der Teilbaum, der mit *next* verbunden ist übersprungen, wenn *rightCondition* zu *false* ausgewertet wird. Falls beide Bedingungen fehlschlagen, werden beide Teilbäume übersprungen. Wenn keiner der Bedingungen mit *false* ausgewertet wird, verhält sich der *Datachoice* wie ein normaler *Choice* aus dem MCTT-Metamodell von Abbildung 5.4. Ein *Or*-Entscheidungsoperator ähnlich zu dem in Abschnitt 5.4.1 vorgestellten *Or*-Decisionnode ist hier nicht vorgesehen.

5.4.2.2 Metamodellerweiterung für die Datenintegration

Ein Auszug des MCTT-Metamodells ist in Abbildung 5.10 zu sehen. Dort wird verdeutlicht, wie das Datenmodell bei MCTT integriert wurde. Analog zu DMWM liegt das Datenmodell und das MCTT-Metamodell als integriertes UML-Klassendiagramm vor.

Die abstrakte Klasse *DataflowObjekt* repräsentiert Datenobjekte, die von mehreren Aufgaben zur Ausführung benötigt werden. Über die Assoziation *flow* wird das Datenobjekt mit den entsprechenden Aufgaben im MCTT-Modell verbunden. Die Klasse ist dem MCTT-Metamodell zugeordnet und über Generalisierungsbeziehungen können Verbindungen zu Klassen des Datenmodells hergestellt werden.

Verwendet man Mehrfachvererbung können Klassen aus dem Datenmodell sowohl Entscheidungsobjekt für datenbasierte Entscheidungen als auch Datenflussobjekt sein. Wie man an den Modellen von Abbildung 5.9 und 5.10 sehen kann, gilt das für die Klasse *PatientData*. Für sie besteht eine Vererbungsbeziehung sowohl zur Klasse *DataflowObject* als auch zu *DecisionObject*.

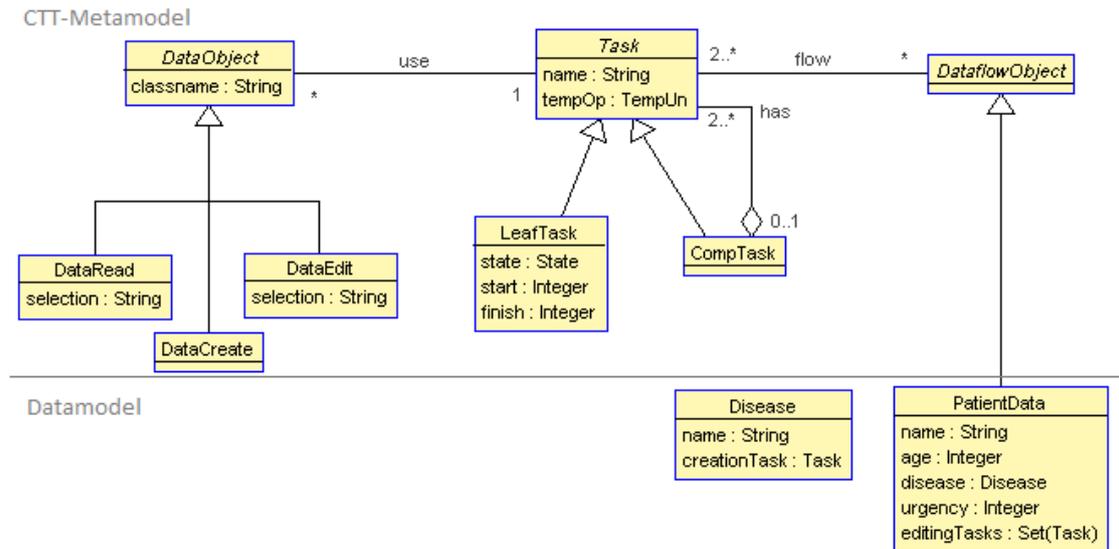


Abbildung 5.10: Das MCTT-Metamodell mit Datenmodellintegration

Wie in Abschnitt 3.3.2 zum DMWM-Metamodell diskutiert, besteht auch bei MCTT die Möglichkeit, über die *DataRead*-Klasse eine Datenanfrage zu stellen. Über *DataEdit* können ebenfalls Daten angefragt und zusätzlich editiert werden. Mit *DataCreate* werden Datenobjekte erzeugt. Die Modellierung und Semantik ist genauso wie bereits in Abschnitt 3.3.2 für DMWM diskutiert.

Der wesentliche Unterschied der Datenintegration vom DMWM- zum MCTT-Metamodell besteht darin, dass bei MCTT auch innere, nicht ausführbare Aufgaben mit den Elementen der Datenintegration verbunden werden können. Dieser Fakt kann Vorteile haben, indem die Datenbeziehungen der inneren Knoten des Aufgabenbaums auf die Blattknoten übertragen werden. Im Modell können damit Verbindungen zu Datenelementen eingespart werden, wenn statt zu allen relevanten Blattaufgaben lediglich eine Verbindung zur inneren Aufgabe hergestellt wird. Weniger Linien und Verbindungen fördern hiermit die Übersichtlichkeit der Modelle.

5.4.2.3 Modellierung eines MCTT-Modells mit DataChoice und Datenintegration

Für das Anwendungsbeispiel wird hier das bereits in Abschnitt 5.3.4 eingeführte MCTT-Beispiel herangezogen. Die implizite Entscheidungsmodellierung zwischen *HandleCriticalPatient* und *HandleStablePatient* wird in Abbildung 3.13 durch eine explizite, datenbasierte ersetzt.

Im Assoziationsobjekt *datachoice1* werden mit den Attributen *leftCondition* und *rightCondition* die Bedingungen spezifiziert, die die Kriterien zur Auswahl des entsprechenden Teilbaums festlegen. Da die Baumdarstellung, wie bereits in Abschnitt 5.3.4 erwähnt, für MCTT gespiegelt und gedreht wurde, bezieht sich *leftCondition* auf die obere Aufgabe *HandleStablePatient*. Die Bedingung ist hierfür mit *urgency*<5 festgelegt. Diese bezieht sich auf das Attribut *urgency* der Klasse *PatientData* aus dem Datenmodell (s. Abbildung 5.10). Die Auswertung wird zur Runtime vom MCTT-Plugin vorgenommen.

Für die von dem Datachoice darunter angeordnete Aufgabe *HandleCriticalPatient* gilt die Bedingung *urgency*>=5. Sobald die über *uses* verbundene Entscheidungsaufgabe *CheckPatientCondition* beendet wurde, wird der entsprechende Teilbaum übersprungen für den die Bedingung nicht zutrifft. Bei mehreren

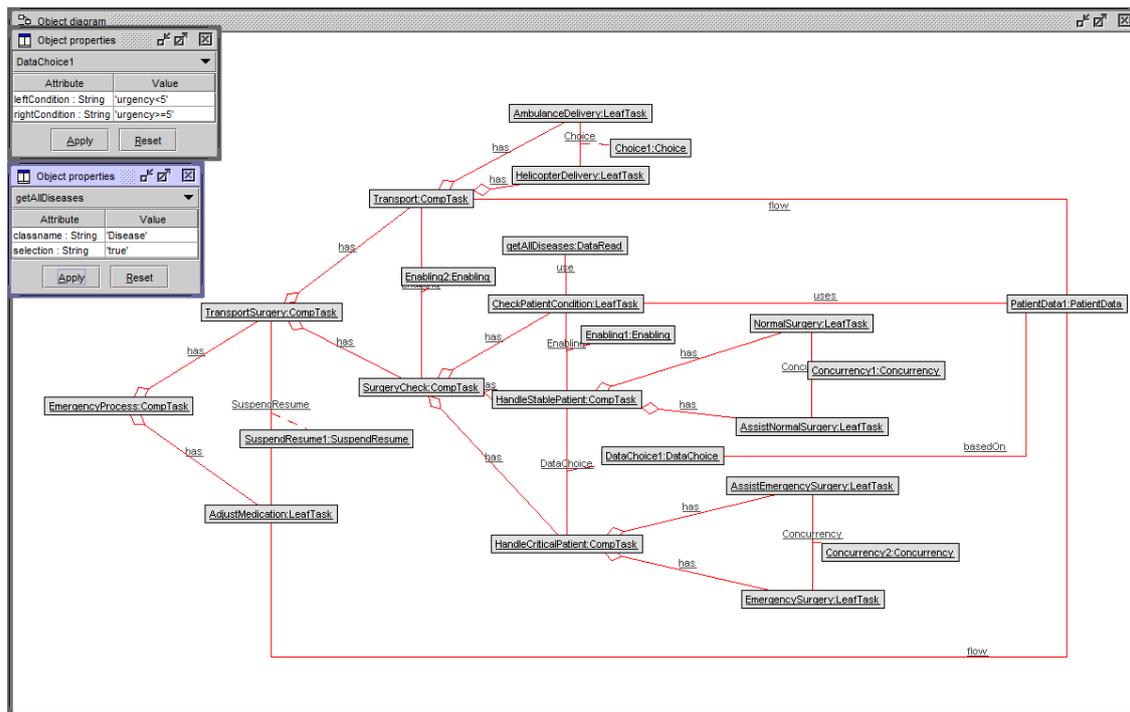


Abbildung 5.11: Das MCTT-Modell mit Datenintegration

in einer Kette angeordneten *Datachoices* können auch mehrere Teilbäume übersprungen werden. Falls das Attribut *urgency* zur Runtime nicht gesetzt wurde, werden beide Teilbäume zur Ausführung freigegeben und es entscheidet sich wie im Modell von Abbildung 5.7 implizit, welcher Teilbaum ausgewählt wird.

Das Objekt *PatientData* repräsentiert zusätzlich zum Entscheidungsobjekt auch ein Datenflussobjekt. Über die Assoziation *uses* ist das Datenobjekt mit den Aufgaben *Transport* und *AdjustMedication* verbunden. Somit werden dem Nutzer während der Ausführung dieser Aufgaben bzw. deren Unteraufgaben die Patiententendaten präsentiert.

Zusätzlich zu der Datenflussspezifikation ist ein Datenanfrageobjekt mit *getAlIDiseases* vorhanden. Im unteren der beiden Objekteigenschaftsfenster von Abbildung 5.11 ist zu sehen, dass alle Objekte vom Typ *Disease* angefragt werden. In der *selection*-Klausel ist *true* angegeben, so dass keine Filterung dieser Objekte stattfindet. Das Anfrageobjekt ist mit der Blattaufgabe *CheckPatientCondition* verbunden, so dass dem Nutzer während deren Ausführung die Informationsobjekte präsentiert werden.

Kapitel 6

Workflow-Ausführung und -Management mit Aufgabenmodellen

Dieses Kapitel behandelt die Ausführung von Aufgabenmodellen für zwei unterschiedliche Einsatzgebiete. Hierfür gibt Abschnitt 6.1 eine Einführung einerseits in die Thematik der lokalen Modellsimulation mit Aufgabenmodellen und andererseits in die verteilte Workflowausführung mit Workflow Management Systemen (WfMS).

In Abschnitt 6.2 wird die metamodellbasierte Umsetzung der Ausführung von MCTT-Modellen behandelt. In dem Zusammenhang wird ein Plugin für das UML-Spezifikationswerkzeug USE vorgestellt, das eine adäquate Nutzerschnittstelle zur Validierung der dynamischen Modelleigenschaften bereitstellt. Kapitel 5 hat bereits die Konzepte zur Modellierung von Geschäftsprozessmodellen mit Aufgabenmodellen behandelt. Im Metamodell wurde teilweise dafür die Ablaufsemantik analog zum DMWM-Ansatz von Abschnitt 3.2.4 deklarativ, plattformunabhängig auf Basis von OCL spezifiziert. Zur Ausführung der MCTT-Modelle ist zudem eine weitere imperative Erweiterung des Metamodells notwendig, die hier vorgestellt wird.

Eine andere Variante der Workflow-Modellausführung behandelt Abschnitt 6.3. Es wird mit dem *Task Tree Based Workflow Management System (TTMS)* ein Aufgabenmodell-basiertes WfMS eingeführt. Die verwendete TTMS-Workflowsprache, der Editor zur Definition der Modelle und das Workflow Management System ist dort Thema.

Schließlich werden weitere, verwandte Arbeiten und Veröffentlichungen im Bereich der Modellierung und Ausführung von Aufgabenmodellen und von weiteren Workflow Management Systemen in Abschnitt 6.4 vorgestellt.

6.1 Einführung

Zunächst werden die Eigenschaften und Ziele der lokalen Workflow-Modellausführung betrachtet. Hiermit soll mit MCTT eine einfach anzuwendende Validationsmöglichkeit der Modelle geschaffen werden. Die Darstellung und Runtime-Perspektive ist eine andere als die der Modelldarstellung zur Design-time. Somit können mit der Ausführung Fehler in den Modellen erkannt und behoben werden, die ohne Validierung durch Modellausführung nicht aufgefallen wären. Die hier vorgenommene Validierung kann als Diskussionsgrundlage zwischen Entwickler und Stakeholder genutzt werden. So leistet sie Beiträge beim Requirement Engineering für Modelle, die z.B. zur Systemimplementierung weiterverwendet werden können.

Mit der Ausführung der Modelle wird des Weiteren eine Validierung des MCTT-Metamodells vorgenommen. Auch hier könnten Fehler enthalten sein. Das validierte MCTT-Metamodell kann daraufhin zur Implementierung einer Workflow-Engine angewendet werden, die im Zusammenhang mit einem Workflow Management System MCTT-Workflowmodelle abarbeitet. Soundness-Probleme und Mehrdeutigkeiten in der operationalen Semantik werden mit dem in Abschnitt 5.3 vorgestellten Ansatz ausgeräumt. Außerdem kann der in diesem Abschnitt behandelte Ausführungsansatz mit dem UML-Tool USE für die Entwicklung einer Workflow-Engine weiterverwendet werden.

Betrachtet man die im folgenden Abschnitt 6.2 behandelte Ausführung der MCTT-Modelle, gilt dafür größtenteils das Use Case-Diagramm für die Ausführung der DMWM-Modelle in Abbildung 4.1 analog. Wie bereits erwähnt, wird der Organisationsaspekt für MCTT-Modelle jedoch nicht berücksichtigt und somit gilt der Use Case zur Validierung des Organisationsmodells nicht. Dagegen werden die anderen Anwendungsfälle zur Ausführung bzw. Validierung der Modelle auch bei MCTT für den Nutzer bereitgestellt. Während der Ausführung werden auch die in Abschnitt 5.4.2.2 betrachteten Datenaspekte berücksichtigt. Das Plugin fragt entsprechende Daten während der Workflowausführung vom Nutzer ab und präsentiert Daten, die zur Durchführung der Aufgabe benötigt werden.

Zusätzlich zur lokalen Ausführung gibt es die verteilte Ausführung im Zusammenhang mit Workflow Management Systemen (WfMS). Diese sind dafür entwickelt worden, Arbeitsabläufe in Unternehmen operativ zu koordinieren. Sie sind generische Softwaresysteme, die Geschäftsprozessmodelle ausführen und anhand derer die Geschäftsabläufe und -logik implementiert werden. Durch WfMSe werden Aufgaben an Angestellte zugewiesen, die diese auf Basis und mit deren Hilfe ausführen sollen. In der Regel stellt das WfMS eine Webschnittstelle bereit, mit der der Nutzer von beliebigen Orten und mit beliebigen Endgeräten auf das System zugreifen und die anstehenden Aufgaben ausführen kann. YAWL [HAAR09] oder ADEPT [DRRM11] sind Beispiele für solche Systeme.

Im Gegensatz zu WfMSen sind viele Geschäftsabläufe in Unternehmen in konventioneller Software, wie z.B. Enterprise Resource Planning (ERP-) oder Customer Relationship Management (CRM-) Systemen fest codiert. WfMS sind im Vergleich zu konventioneller Software flexibler, da die Modelle und damit die Abläufe im Unternehmen vergleichsweise einfach verändert werden können. Nachdem die Modelle anhand von visuellen Modellen in Workfloweditoren angepasst wurden, können diese für das WfMS bereitgestellt und instanziiert werden. Daraufhin werden die veränderten Prozesse ausgeführt, ohne dass das WfMS angepasst werden muss. Will man die Geschäftsabläufe dagegen bei Einsatz konventioneller Software verändern, so muss deren Quellcode angepasst werden, was einen deutlichen Mehraufwand bedeuten würde. Geschäftsprozesse können mit WfMS schneller verändert bzw. optimiert werden. Diese sind dadurch flexibler einsetzbar und können auf veränderte Bedingungen, die Anpassungen der Geschäftsprozesse erfordern, besser reagieren.

Die Vorteile von Aufgabenmodellen (s. Abschnitt 5.1.2) lassen sich auch mit Workflow Management Systemen verbinden. Der Ansatz wird in Abschnitt 6.3 mit dem TTMS-System verfolgt. Dem Nutzer wird während der Modellausführung die Hierarchie der Modelle präsentiert und damit eine integrierte Zielmodellierung an die Hand gegeben. Außerdem sind die Modelle strukturiert, was ein wichtiger Aspekt für die Verständlichkeit von Modellen ist [LM10]. Mit diesen Eigenschaften scheinen Aufgabenmodelle eine vielversprechende Möglichkeit zu sein, um Arbeit in Unternehmen zu koordinieren und mit deren Hilfe vom Nutzern ausführen zu lassen.

Die Entwicklung von TTMS hatte bereits vor der Entwicklung von MCTT begonnen und ging dann parallel weiter. Damit war eine vollständige Vereinheitlichung der Konzepte nicht möglich, was jedoch wünschenswert gewesen wäre. Jedoch sind die Ansätze ähnlich und die entwickelten Tools können trotzdem in Kombination eingesetzt werden, worauf Abschnitt 6.3.2.3 näher eingeht.

6.2 Modellausführung von MCTT-Aufgabenmodellen

Die lokale Ausführung der MCTT-Workflowmodelle besteht aus zwei wesentlichen Bestandteilen, die in diesem Abschnitt näher vorgestellt werden. Die Grundlage zur Ausführung der Workflowmodelle wird in Abschnitt 6.2.1 mit den auf Basis der *Simple OCL-based Imperative Language* (SOIL) [BG11] umgesetzten Operationen des MCTT-Metamodells gelegt. Darauf basierend wurde das für DMWM bereits entstandene und in Abschnitt 4.3 vorgestellte Workflow-Plugin für MCTT adaptiert und erweitert. Dieses ist damit fähig, auch MCTT-Modelle zu interpretieren und dem Nutzer eine adäquate Interaktionsschnittstelle bereitzustellen, die in Abschnitt 6.2.2 näher beschrieben wird.

6.2.1 Umsetzung der operationalen Semantik mit SOIL

Im MCTT-Metamodell sind nicht nur strukturelle Einschränkungen angegeben (s. Abschnitt 5.3.3), auch Teile der operationalen Ausführungssemantik sind deklarativ und plattformunabhängig über OCL spezifiziert. Sowohl die deklarativen Einschränkungen als auch weitere benötigte Hilfsfunktionen, die zur Ausführung der MCTT-Modelle gebraucht werden, wurden mit SOIL umgesetzt. Zudem werden, wie auch schon beim DMWM-Ansatz, OCL-Hilfsfunktionen (s. Abschnitt 3.2.3.2) genutzt, um die operationale Semantik umzusetzen.

Die OCL- und SOIL-Hilfsoperationen wurden zur Übersichtlichkeit in Abbildung 5.4 weggelassen. Diese stellen keine Nutzerschnittstelle dar, über die der Anwender mit den Aufgabenobjekten direkt interagieren kann. Sie werden lediglich indirekt über Interaktionen mit anderen Aufgabenobjekten aufgerufen. Ein Teil der Operationen, die für die metamodellebasierte Umsetzung der operationalen Semantik notwendig sind, werden in Abbildung 6.1 für die Klassen *LeafTask* und *CompTask* sowie für die Assoziationsklassen *TempOp*, *DataChoice*, *Choice* und *Enabling* aufgeführt. Die abstrakte Klasse *Task* wurde hier weggelassen, da sie mit ca. 70 Hilfsfunktionen zu umfangreich für die hiesige Dokumentation wurde.

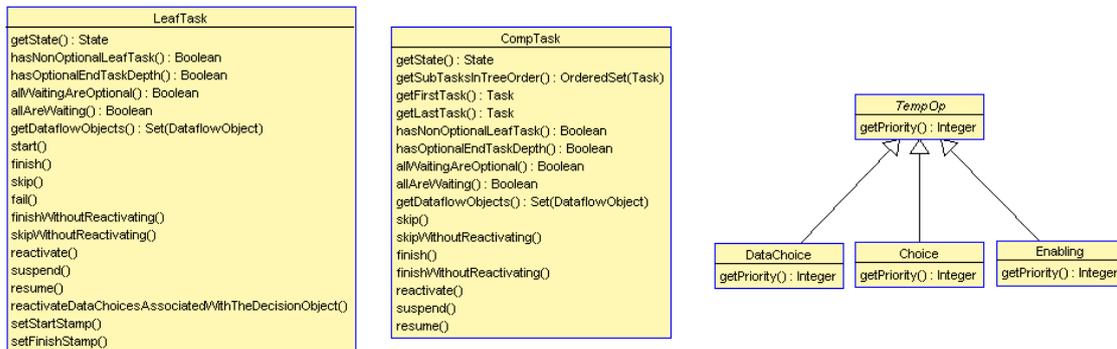


Abbildung 6.1: Klassen aus dem MCTT-Metamodell inkl. der Hilfsoperationen

Eine wichtige Funktion zur Angabe der Bindungsstärke bzw. Prioritäten der temporalen CTT-Operatoren (s. Abschnitt 5.2.2) ist bei der Assoziationsklasse *TempOp* und deren Unterklassen mit der Operation *getPriority()* hinterlegt. Die Unterklassen überschreiben die entsprechende Operation und geben damit ihre spezifische Bindungszahl wieder. So liefert *DataChoice* bzw. *Choice* die Zahl mit der größten und *Enabling* den mit der geringsten Bindung zurück. Auf Basis dieser Information wird der Einflussbereich der entsprechenden Operationsaufrufe ausgehend von der aufgerufenen Blattaufgabe bestimmt. Die Bindungsstärke wird beispielsweise bei den in Listing 6.1 aufgerufenen Operationen der *start()*-Operation geprüft

und berücksichtigt.

Bei *LeafTask* in Abbildung 6.1 sind zunächst diverse Abfrageoperationen angegeben, die keine Effekte haben und lediglich Werte zurückliefern. Daraufhin folgen die für den Nutzer relevanten Schnittstellenfunktionen *start()*, *finish()*, *fail()* und *skip()*, die auch bereits in Abbildung 5.4 angegeben sind. Um einen Eindruck von der SOIL-Umsetzung dieser Operationen zu kriegen, ist exemplarisch in Listing 6.1 die *start()* Operation angegeben.

Die unter *fail()* bei der Klasse *LeafTask* aufgeführten Operationen verändern ebenfalls den Zustand der Aufgabe und werden je nach temporalen Abhängigkeiten zu anderen Aufgaben indirekt aufgerufen. Die Operationen *setStartStamp()* und *setFinishStamp()* setzen schließlich die entsprechenden Attribute von *LeafTask*, die für die Zeitstempel vorgesehen sind.

Die Klasse *CompTask* umfasst ebenfalls sowohl OCL-Abfrage- als auch imperative SOIL-Operationen. Die Anfrageoperationen reichen von *getState()*, die den Zustand der zusammengesetzten Aufgabe auf Basis deren Kindaufgaben berechnet, bis *getDataflowObjects()*, die die verbundenen Datenobjekte zurückliefert. Darunter folgen mit *skip()* bis *resume()* Operationen, die den Zustand der in der Aufgabenhierarchie darunterliegenden Blattaufgaben entsprechend verändern sollen. Im Unterschied zu *LeafTask* sind alle imperativen Operationen indirekt von anderen Aufgabenobjekten und nicht vom Nutzer aufrufbar. Beispielsweise kann beim Starten einer mit *Enabling* folgenden Aufgabe, die optionale vorgelagerte zusammengesetzte Aufgabe mit *skip()* übersprungen werden. In Listing 6.1 ist mit dem Aufruf *skipOptionalTasks()* diese Semantik umgesetzt.

```

1  start() begin
2    declare
3      suspended : Boolean,
4      startable : Boolean;
5    suspended := self.isSuspendedByOrderIndependency();
6    startable := not suspended;
7    if (startable) then
8      suspended := self.isSuspendedBySuspendResume();
9      startable := not suspended;
10   end;
11   if (startable) then
12     suspended := self.isSuspendedByEnabling();
13     startable := not suspended;
14   end;
15   if (startable) then
16     self.state := #running;
17     self.skipOptionalTasks();
18     self.skipDataChoice();
19     self.skipChoice();
20     self.disableTasks();
21     self.suspendTasks();
22     self.setStartStamp();
23   end
24 end
25 pre leafStart_waitingTask: self.state = #waiting
26 post leafStart_runningTask: self.state = #running

```

Listing 6.1: SOIL-Umsetzung der *start()*-Operation der Klasse *LeafTask*

In Listing 6.1 ist mit *start()* eine SOIL-Operation aus dem MCTT-Metamodell angegeben. Analog dazu sind alle weiteren SOIL-Operationen umgesetzt. In der *start()*-Operation werden zunächst die zwei

Boolean-Variablen *suspended* und *startable* deklariert. Mit Hilfe dieser und den Operationen *isSuspendedByOrderIndependency()*, *isSuspendedBySuspendResume()* und *isSuspendedByEnabling()* wird in den Zeilen 5-14 geprüft, ob die Aufgabe durch entsprechende temporale Operatoren ausgesetzt wurde oder gestartet werden kann. Die Aufgabe wird gestartet, indem deren Zustand auf *running* gesetzt wird (s. Zeile 16). Darauf folgend werden in den Zeilen 17-22 Seiteneffekte auf verbundene Aufgaben durchgeführt, die auf Basis der entsprechenden temporalen Operatoren geschehen müssen. Zunächst werden die *skip()*-Operationen aufgerufen, die bei optionalen oder mit Choices verbundenen Aufgaben auftreten können. Des Weiteren können mit dem Starten der Aufgabe andere übersprungen oder ausgesetzt werden, wenn diese mit dem *Disabling* bzw. *SuspendResume*-Operator verbunden sind. Dieses passiert mit den *disableTasks()* und *suspendTasks()*-Operationsaufrufen in Zeile 20 und 21. Als letztes wird in Zeile 22 die *setStartStamp()* Operation aufgerufen, welche das *start*-Attribut der Klasse *LeafTask* auf den entsprechenden Zeitpunkt setzt (s. Abbildung 5.4).

Im Gegensatz zum hier vorgestellten SOIL-Code wurde wie bereits erwähnt beim DMWM-Ansatz der imperative Teil mit ASSL umgesetzt. Das DMWM-Metamodell und der ASSL-Code lag in unterschiedlichen Dateien vor. Mit der Verwendung von SOIL hat sich diese Verwendung bei MCTT geändert, da die imperativen Operationen nun direkt im Metamodell integriert sind. Da die Bedingungen und der imperative Code nun in einer Datei vorliegen, sind in Listing 6.1 die Vor- und Nachbedingungen in den Zeilen 25 und 26 mit spezifiziert. Mit diesen Bedingungen wurde die *start()*-Transition der Zustandsautomaten von Abbildung 5.5 umgesetzt.

Ein weiterer Unterschied von der ASSL zur SOIL-Verwendung liegt darin, dass bei den ASSL-Prozeduren, wie in Listing 4.1 angegeben die entsprechende Aufgabe als Parameter übergeben werden musste. Diese ist bei SOIL nicht mehr notwendig, da die Operation auf einem entsprechenden *LeafTask*-Objekt ausgeführt wird. Die Syntax von SOIL liegt damit näher an einer objektorientierten Programmiersprache als dieses noch bei ASSL der Fall war.

6.2.2 Taskmodel-Plugin für USE

In diesem Abschnitt wird die Ausführung des Aufgabenmodells von Abbildung 5.11 im MCTT-Workflow-Plugin thematisiert. Die Aufgabenmodellldarstellung, die Interaktionsschnittstelle für den Nutzer und die Darstellung der Datensicht wird in Abbildung 6.2 präsentiert.

Die Darstellung des Workflows ist ganz ähnlich zum DMWM-Plugin. Das Workflowmodell ist ebenfalls in einem Baum gezeigt und die Ausführungszustände der Aufgaben werden farblich gekennzeichnet. Die Farbzusordnungen sind die gleichen wie bei DMWM, die in Tabelle 4.3 vorgestellt wurden. Jedoch ist mit *suspended* ein zusätzlicher Ausführungszustand bei MCTT hinzugekommen, für den die Farbe *Türkis* verwendet wurde. Die Buttons zur Nutzerinteraktion sind mit *start*, *finish*, *skip* und *fail* die gleichen wie beim DMWM-Plugin geblieben.

In Abbildung 6.2(a) ist die Aufgabe *CheckPatientCondition* gestartet und ausgewählt. Damit werden die entsprechenden Datensichten für diese Aufgabe angezeigt. Das Workflowmodell von Abbildung 5.11 zeigt, dass die Aufgabe mit *PatientData* als Entscheidungsobjekt verbunden ist. Somit wird über die Eingabe von Daten über den weiteren Verlauf des Workflows entschieden. Hier wurde dem Attribute *urgency* der Wert *6* zugeordnet. Mit der Spezifikation des *Datachoice1*-Objekts aus Abbildung 5.11 ist ersichtlich, dass damit die Aufgabe *HandleCriticalPatient* zur Ausführung ausgewählt werden müsste. Nach Beendigung der Aufgabe *CheckPatientCondition* ist in Abbildung 6.2(b) ersichtlich, dass die Aufgabe *HandleStablePatient* übersprungen wurde. Dafür muss dann also *HandleCriticalPatient* ausgeführt werden.

Die Aufgabe ist jedoch in Abbildung 6.2(b) nicht *enabled*, da zwischenzeitlich die Aufgabe *AdjustMedi-*

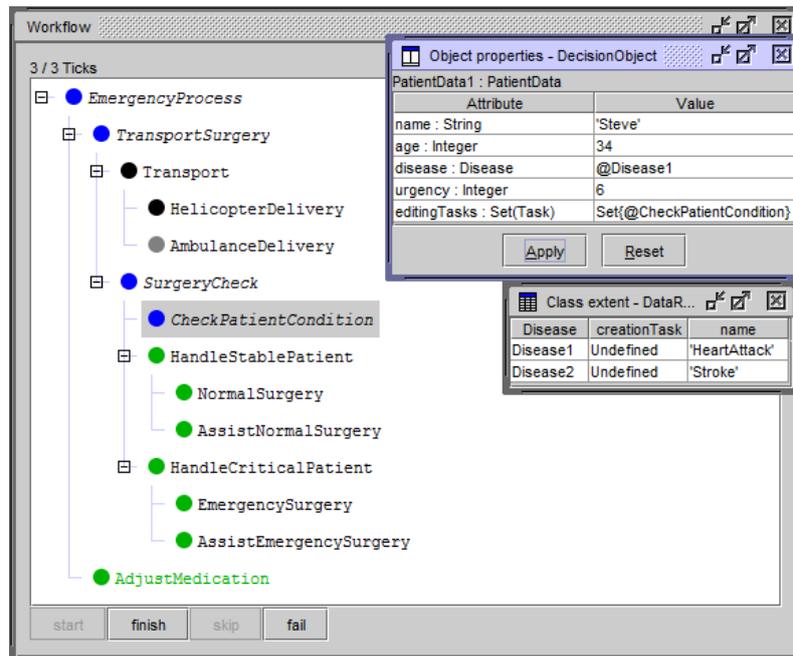
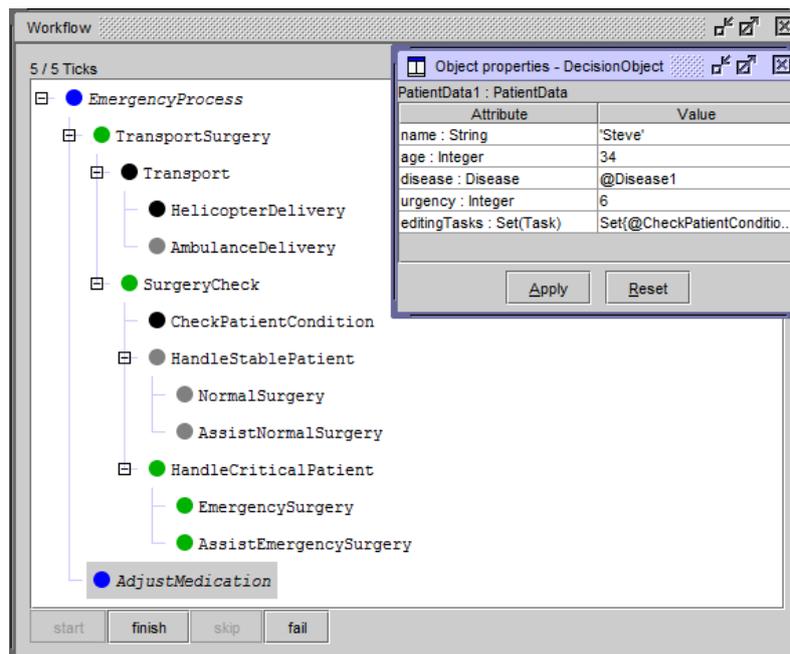
(a) Ausführung der Aufgabe *CheckPatientCondition*(b) Ausführung der Aufgabe *AdjustMedication*

Abbildung 6.2: MCTT-Modelle zur Runtime dargestellt im MCTT-Plugin

tion gestartet wurde. Mit diesem Aufruf wurden die davor gelegenen Aufgaben durch den temporalen Operator *SuspendResume* unterbrochen. Nachdem diese Aufgabe wieder beendet wird, sind die Aufgaben *EmergencySurgery* und *AssistEmergencySurgery* wieder startbar.

Zusätzlich zum *PatientData* Objekt ist in Abbildung 6.2(a) ersichtlich, dass ein weiteres *DataRead*-Fenster geöffnet ist. Dieses wird durch die Interpretation des *DataRead*-Objekts *getAllDiseases* geöffnet. Damit

werden die Krankheiten aus dem Datenbestand des Objektdiagramms abgefragt, die die Datenbasis des Krankenhausinformationssystem repräsentiert bzw. simuliert.

In Abbildung 6.2(b) ist zu sehen, dass die Aufgabe *AdjustMedication* ausgewählt ist, so dass die entsprechenden Datenverbindungen dieser Aufgabe angezeigt werden. Im Workflowmodell von Abbildung 5.11 wurde diese Aufgabe mit dem *PatientData*-Objekt mit einem *flow*-Link verbunden. Somit dient das Datenobjekt *PatientData* sowohl als Entscheidungs- als auch als Datenflussobjekt.

Die Datensichten sind über den Optionsmenü-Eintrag *Show DataViews automatically* analog zu dem von DMWM in Abbildung 4.4(d) ein- bzw. wieder ausblendbar. Die weiteren Auswahlmöglichkeiten im Konfigurationsmenü sind bis auf *Instantiate*, *Load ASSL-Runtime* und *Revert to default ASSL-Runtime* erhalten geblieben. Da die Sprache SOIL anstatt von ASSL gewählt wurde, wurden außerdem die beiden Menüpunkte *Run*- und *Peek ASSL Procedure* entsprechend durch *SOIL* ersetzt.

6.3 TTMS: A Task Tree Based Workflow Management System

In diesem Abschnitt wird das Aufgabenmodell-basierte Workflow Management System TTMS vorgestellt. Abschnitt 6.3.1 führt dafür die verwendete Workflowsprache ein. Zum Erstellen der Modelle wird im gleichen Abschnitt der Editor vorgestellt. Schließlich wird in Abschnitt 6.3.2 das Workflow Management System, das Konzept der Instantiationtime und die Verwendung der umgesetzten Tools in einer Werkzeugkette präsentiert.

6.3.1 TTMS-Workflowsprache und Editor

Die TTMS-Workflowsprache hat mit dem hierarchischen Charakter und den binären temporalen Beziehungen zwischen Geschwisterknoten im Aufgabenbaum die gleiche Charakteristik wie CTT. TTMS setzt jedoch nicht alle temporalen Operatoren um. Lediglich die wichtigsten werden mit *Choice*, *Concurrency*, *Enabling* und *Iteration* von TTMS unterstützt und in Tabelle 6.1 (in der oberen Hälfte über dem Doppelstrich) aufgeführt. Zusätzlich zu der Syntax der Operatoren in TTMS werden dort die äquivalenten CTT-Operatoren und die entsprechenden Bezeichnungen angegeben. Die TTMS-Syntax weicht etwas von der CTT-Syntax ab und verwendet statt mehrerer Zeichen pro Operator wie bei CTT jeweils genau ein Zeichen für TTMS.

TTMS-Operator	CTT-Operator	Bezeichnung
$A \times B$	$A \square B$	XOR-Choice
$A + B$	$A \parallel B$	Concurrency
$A ; B$	$A \gg B$	Enabling
$A\{n\}$	A^*	Iteration
$A X B$	in CTT nicht vorhanden	XOR-Choice (explicit)
$A O B$	in CTT nicht vorhanden	OR-Choice (explicit)
$A I B$	in CTT nicht vorhanden	Instantiationtime XOR-Choice
$A D B$	in CTT nicht vorhanden	Instantiationtime OR-Choice

Tabelle 6.1: Temporale TTMS-Operatoren

Bei dem unären Operator *Iteration* kann bei TTMS statt des n eine konkrete Zahl angegeben werden, mit der die Anzahl der Durchläufe bereits zur Designtime festgelegt wird. Die Anzahl kann aber auch wie bei CTT offen gelassen werden, so dass der Nutzer während der Runtime die Anzahl der Iterationen bestimmt.

Bezüglich der Entscheidungsmodellierung ist anzumerken, dass bei TTMS diverse weitere Varianten

umgesetzt wurden. Die implizite Variante wird analog zum CTT-Choice-Operator weiter angeboten, die sich wie in Abschnitt 5.2.4 beschrieben so wie ein *Deferred Choice* verhält. Im Workflowmodell sind bei der impliziten Variante keine Kriterien angegeben, die ausdrücken, bei welcher Situation welche Alternative der angebotenen Aufgaben auszuwählen ist. Zur Runtime führt die Ausführung einer aktivierten Aktivität zur impliziten Deaktivierung der anderen Alternative. Dadurch muss dem Nutzer nicht zwangsläufig bewusst sein, dass er durch Starten einer Aktivität die andere deaktiviert.

Im unteren Teil (unter dem Doppelstrich) der Tabelle 6.1 sind vier weitere Entscheidungsoperatoren angegeben, die nicht Teil von CTT sind. Bei der expliziten Variante ist dem Nutzer stets bewusst, dass eine Entscheidung zu treffen ist, da die Kriterien explizit vom Nutzer durch das WfMS abgefragt werden. Auf dessen Grundlage werden dann die gewünschte Aktivitäten aktiviert und die anderen deaktiviert. Diese Abfrage wird in Abbildung 6.4 am TTMS-WfMS angegeben und in Abschnitt 6.3.2.1 näher beschrieben.

Die Variante des expliziten *Choices* wird bei TTMS durch *Decisionnodes*, die bereits in Abschnitt 5.4.1 vorgestellt wurden, umgesetzt. Die Unterknoten der *Decisionnodes* müssen dann mit durchgehend genau einer Sorte von binären Entscheidungsoperatoren in einer Kette verbunden werden. Dieser Operator gibt die Art der geforderten Entscheidung an. Die Kriterien zur Entscheidungsauswahl werden des Weiteren an die Unteraufgaben mit sogenannten *Guards* spezifiziert. Ähnlich zur Aktivitätsdiagramm-Syntax werden die *Guards* im Flussdiagramm angezeigt (s. Abbildung 6.3). In der Baumdarstellung des Editors sind die *Decisionnodes* und die darunterliegenden Aufgaben, die die Ausführungsalternativen darstellen, abgebildet. Es fehlt dort jedoch die Darstellung der *Guards*.

Zusätzlich zum exklusiven *Choice*, bei dem genau eine Ausführungsalternative ausgewählt werden muss, wurde bei TTMS noch eine inklusive Art analog zum *OR*-Operator bei den EPKs umgesetzt. Diese Art wurde bereits in Abschnitt 5.4.1 in Abbildung 5.8(b) eingeführt.

Des Weiteren können die expliziten Entscheidungsoperatoren noch als *Instantiationtime*-Operatoren eingesetzt werden, um die Prozessmodelle zur Instanzierungszeit für den konkreten Fall zu konfigurieren. Diese Vorgehensweise wird in Abschnitt 6.3.2.2 näher beleuchtet.

Im linken Teil des Editors aus Abbildung 6.3 ist die hierarchische Dekomposition der Aufgaben abgebildet. Hier können einzelne Unteraufgaben oder ganze Unterbäume aus externen Modellen in die Baumhierarchie neu eingefügt werden. Es fehlt in der Baumdarstellung die visuelle Angabe der temporalen Operatoren, wie sie z.B. bei der CTT-Syntax vorhanden ist. Diese lassen sich im TTMS-Editor über einen Rechtsklick mittels der *Task properties*-Abfrage in einem separaten Fenster darstellen (s. Abbildung 6.3).

Alle inneren Knoten im Aufgabenbaum, die durch Dekomposition Kinderknoten beinhalten, müssen diese in temporale Beziehungen setzen. Die Zeichen aus Tabelle 6.1, die die temporalen Operatoren repräsentieren, werden bei der Spezifikation der Aufgabeneigenschaften im TTMS-Editor benutzt. Das Fenster dafür ist in Abbildung 6.3 für die Aufgabe *EmergencyProcess* angegeben. Die Unteraufgaben haben hier Nummern, die für die im Prozessalgebra-Term zur Spezifikation der temporalen Beziehungen benötigt werden. Hierbei kann entgegengesetzt zu CTT eine Klammerung der Unteraufgaben erfolgen, um damit die Bindungsstärke der verwendeten temporalen Operatoren anzugeben. Eine vorgegebene Priorisierung der temporalen Operatoren, wie bei CTT ist damit zwar nicht notwendig, wird aber vom TTMS-Editor und WfMS umgesetzt und weiter bereitgestellt.

Zusätzlich zur Baumdarstellung werden die temporalen Beziehungen im Editor in einem Flussdiagramm ähnlich zu einem Aktivitätsdiagramm visualisiert. In der Aufgabenhierarchie kann eine Aufgabe ausgewählt werden, auf die sich die Flussdiagrammdarstellung bezieht. Somit wird bei Auswahl des Wurzelknotens der ganze Prozess dargestellt. Bei Auswahl einer Unteraufgabe wird dagegen nur dieser Prozessausschnitt veranschaulicht. Des Weiteren kann über Buttons in der Symbolleiste des Editors entschieden werden, ob

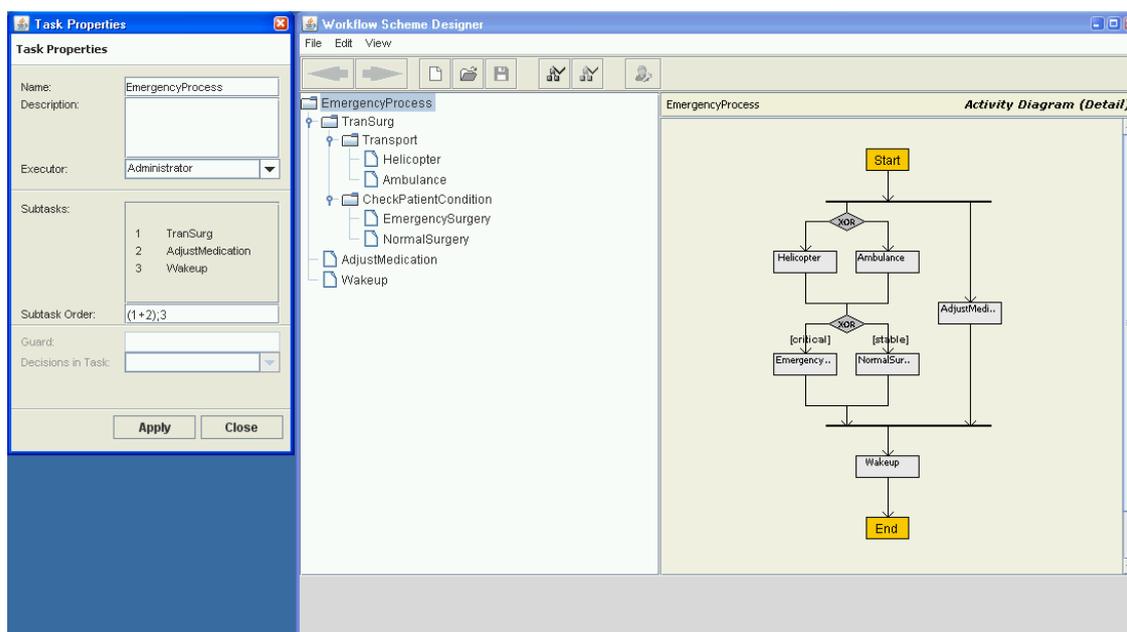


Abbildung 6.3: Der TTMS-Editor

entweder die Knoten der nächsten Hierarchie oder alle Blattknoten betrachtet werden sollen. In Abbildung 6.3 werden die Blattknoten und die entsprechenden temporalen Beziehungen dargestellt. Die inneren Knoten werden hier nicht berücksichtigt.

Die Flussdiagramme werden automatisch angeordnet und es kann keine manuelle Anpassung des Layouts erfolgen. Durch die vorgegebene Strukturierung der Aufgabenmodelle kann dieses zuverlässig automatisch und recht übersichtlich erfolgen. Die kompakte Baum- und Flussdiagrammdarstellung im Editor sind auch für recht große Prozessmodelle geeignet. Hier wurden Prozessmodelle mit ca. 30 Aufgaben entwickelt, die durch den Editor übersichtlich dargestellt werden konnten.

In Abbildung 6.6 ist der Zusammenhang zwischen Workflow-Editor und WfMS zu sehen. Die vom Editor erstellten Modelle werden in einem XML-Dokument abgespeichert, das als Austauschformat zwischen Editor und WfMS dient. Diese Verbindung wird mit *Interface1* vom Workflow-Referenzmodell der WfMC beschrieben [Hol98]. Die als XML-Datei repräsentierten Workflowmodelle können in das zugehörige WfMS hochgeladen, instanziiert und dann interpretiert werden. Hierbei ist eine sogenannte Workflow-Engine für die Interpretation und Abarbeitung verantwortlich.

Das Use Case-Diagramm aus Abbildung 6.5(a) verdeutlicht, dass bei der Ausführung der Prozesse mehrere Rollen mit dem Workflow Management System in Kontakt kommen können. Bei einem Krankenhausprozess können das z.B. der Arzt (*Physican*) und das Pflegepersonal (*Nurse*) sein. Die Rollenzuordnung zur Aufgabe kann zur Designtime vorgenommen werden, wobei eine automatische Synchronisierung der Akteurs- und Rollenmodelle zwischen Editor und WfMS, wie es z.B. YAWL bereitstellt [HAAR09], bei TTMS nicht gegeben ist.

6.3.2 TTMS-WfMS

Das Webinterface und die Funktionalitäten des TTMS-WfMS wird in Abschnitt 6.3.2.1 vorgestellt. Das Konzept der Instantiationtime ist daraufhin Thema in Abschnitt 6.3.2.2. Die Verwendung anderer Tools zur

Aufgabenmodellierung wird im Zusammenhang mit TTMS in Abschnitt 6.3.2.3 behandelt.

6.3.2.1 Nutzerschnittstelle des WfMS

Die Nutzerschnittstelle des TTMS-WfMS ist in Abbildung 6.4 zu sehen. Das WfMS ist mit Java-Webtechnologie umgesetzt worden und läuft auf einem Java-Webserver (in dem Falle war es *Tomcat* [BD07]). Damit ist eine Client-Server-Architektur umgesetzt worden und der Nutzer kann auf das TTMS-WfMS über einen Browser zugreifen. Aktivitäten der Workflows können darüber vom Nutzer aufgerufen und gesteuert werden.

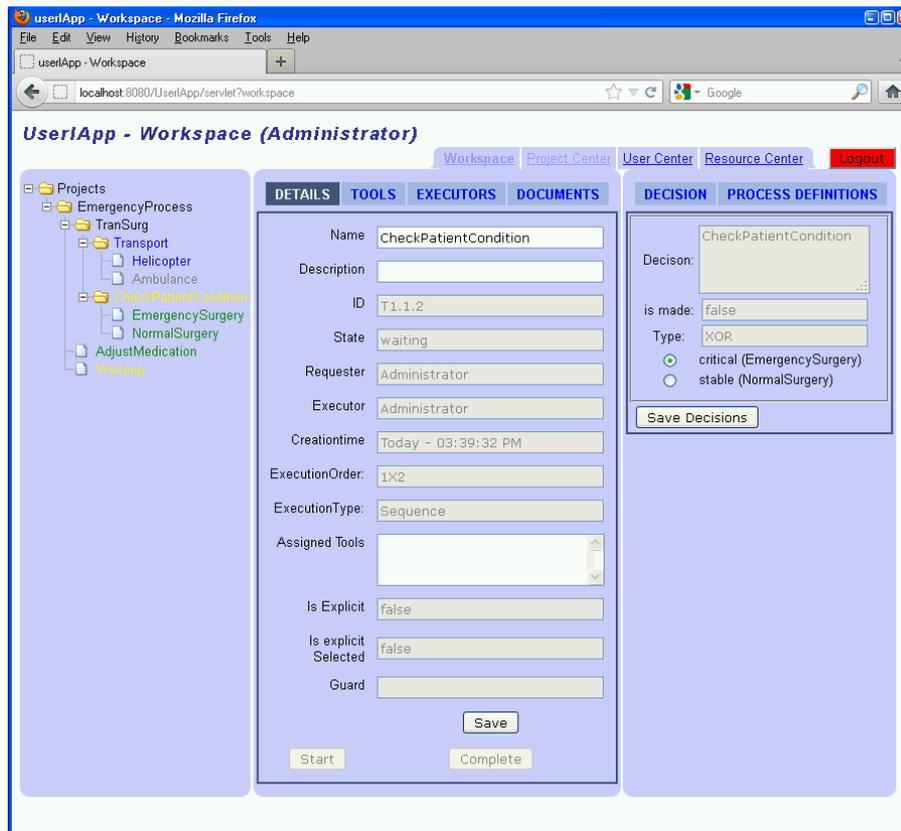


Abbildung 6.4: Die Nutzerschnittstelle des TTMS WfMS bereitgestellt über einen Browser

Über die Client-Server Architektur können des Weiteren mehrere Personen auf das WfMS und somit auf gleiche Prozessinstanzen zugreifen. Auf Basis der Webtechnologie können hier auch mobile Geräte eingesetzt werden, so dass ein flexibler ortsunabhängiger Einsatz des WfMS erfolgen kann. Die Arbeit der Personen, die in einem Geschäftsvorfall involviert sind kann damit koordiniert werden. Diese Funktionsweise über das Web wurde analog in anderen WfMS wie YAWL [HAAR09] umgesetzt.

Im linken Frame des Browserfensters von Abbildung 6.4 werden in einer Baumdarstellung unter der Wurzel *Projects* die laufenden Prozessinstanzen gehängt. Da die Workflowmodelle streng hierarchisch strukturiert und direkt als Baum modelliert sind, können diese unproblematisch als Unterbäume direkt unter der Wurzel eingehängt werden. Die temporalen Beziehungen werden jedoch wie schon bei der Ausführung der MCTT-Modelle in Abschnitt 6.2.2 in der Darstellung weggelassen. Diese ist dadurch deutlich kompakter und auch große Modelle werden benutzbar. Statt der temporalen Operatoren werden die Ausführungszustände der Aktivitäten über Farben ausgedrückt.

In dem Falle von Abbildung 6.4 ist genau eine laufende Workflowinstanz vom *EmergencyProcess* Modell (vgl. Abbildung 6.3) vorhanden. Bei der Instanziierung eines weiteren Prozesses wird ein zweiter Unterbaum unter den Wurzelknoten *Projects* eingehängt. Sollte der Workflow ebenfalls vom Typ *EmergencyProcess* sein, so können die Prozessinstanzen z.B. anhand der assoziierten Dokumente unterschieden werden. TTMS stellt hier über die Schaltfläche *DOCUMENTS* im mittleren Frame des Browserfensters von Abbildung 6.4 die Möglichkeit einer Dokumentensicht für Aktivitäten bereit. Während der Ausführung der entsprechenden Aktivität können hier beliebige Dokumente erstellt, gelöscht oder verändert werden. Das WfMS archiviert diese für die zugeordneten Aktivitäten.

Die operationale Semantik bei TTMS fußt ebenfalls wie bei MCTT auf Zuständen und Zustandsautomaten im Gegensatz zur eventbasierten Prozessalgebra bei CTT [MPS02]. Im vom WfMS dargestellten Aufgabenbaum kann der Nutzer eine gewünschte Blattaufgabe auswählen und diese starten oder beenden. Die entsprechenden Buttons *start* und *finish* sind unten im mittleren Frame von Abbildung 6.4 zu sehen. Wird eine aktivierte Blattaufgabe gestartet, so wird diese in der blauen Farbe im Aufgabenbaum dargestellt. Die von TTMS gebrauchten Farben zur Zustandsdarstellung entsprechen denen von DMWM, die in Tabelle 4.3 vorgestellt wurden. So wie bei DMWM und MCTT werden die Aktivitäten im Zustand *waiting* in *startbare* und *nicht startbare* unterschieden. Dem Nutzer werden die startbaren Blattaufgaben durch eine hellgrüne Farbe markiert. Somit ist eine Führung des Nutzers vom WfMS bei der Ausführung des Workflowmodells gegeben.

Zustandsänderungen der Aufgaben können entsprechend der temporalen Operatoren Seiteneffekte bei anderen Aufgaben des gleichen Modells auslösen. Im Falle von Abbildung 6.4 ist als Beispiel hierfür die Blattaktivität *Helicopter* ausgeführt worden. Der Zustand *done* wird mit der schwarzen Farbe gekennzeichnet. Durch die *Choice*-Beziehung zur Aufgabe *Ambulance* (vgl. Abbildung 6.3) wurde diese Aufgabe damit implizit übersprungen, was durch die graue Farbe ausgedrückt wird.

Zusätzlich zu den Blattaufgaben erfordern die in Abschnitt 5.4.1 und 6.3.1 beschriebenen Decisionnodes die Interaktion des Nutzers. Hierbei müssen diese aus der Aufgabenbaumdarstellung des linken Frames ausgewählt werden. Der Decisionnode *CheckPatientCondition* ist durch eine gelbe Farbe markiert und im Falle von Abbildung 6.4 ausgewählt worden. Die Bezeichnung der selektierten Aufgabe wird im mittleren Frame beim Datenfeld *name* dargestellt. In dem Frame sind des Weiteren Eigenschaften wie die temporalen Beziehungen der Unteraufgaben mit *ExecutionOrder* angegeben. Diese Eigenschanschaften der Aufgaben werden über die dort ausgewählte *DETAIL*-Ansicht dargestellt.

Die Auswahl der zum Decisionnode gehörenden Guards wird im rechten Frame über die dortigen beiden Radiobuttons *critical* und *stable* ausgedrückt. Zusätzlich zu den Guards werden die damit verknüpften Aufgaben in Klammern dahinter angegeben, die auszuführen sind, wenn der entsprechende Guard selektiert wurde. Ist die Auswahl vom Nutzer getroffen worden, so drückt dieser auf den Button *Save Decision*. Die ausgewählten Aufgaben werden daraufhin aktiviert und die restlichen übersprungen.

Es können vier verschiedene Ansichten für die im Aufgabenbaum selektierten Aufgaben ausgewählt werden, die dann im mittleren Frame des WfMS (s. Abbildung 6.4) angezeigt werden. Im Folgenden werden diese aufgelistet und näher erklärt.

- **DETAILS:** Diese Ansicht gibt die Eigenschaften der ausgewählten Aktivität wieder. Sie ist in Abbildung 6.4 zu sehen und die entsprechenden Detailinformationen der Aufgaben sind dort aufgelistet. Diese sind die über die XML-Datei vom Workfloweditor zum WfMS übertragenen Modellinformationen. Abbildung 6.6 verdeutlicht den Austausch zwischen Editor und WfMS über die XML-Datei. Insbesondere die temporalen Beziehungen der Unteraufgaben sind von Bedeutung und werden hier im Feld *ExecutionOrder* abgebildet. Auch die ausführende Person oder Rolle kann bereits im Editor

spezifiziert werden, was im Feld *Executor* ausgedrückt wird. Genauso ist es mit den benötigten Werkzeugen, die im Feld *AssignedTools* angegeben werden.

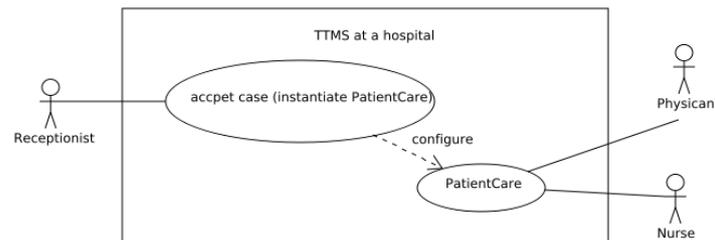
- **TOOLS:** Hierüber wird eine Ansicht über die benötigten Ressourcen für die entsprechende Aufgabe angegeben. Beim Starten der Aufgabe werden die benötigten Ressourcen allokiert. Sollten hier welche fehlen wird dieses über diese Ansicht verdeutlicht. Welche Ressourcen derzeit in Benutzung sind wird im *Ressource Center* vermerkt, der im Folgenden noch näher erläutert wird. Ggf. kann die Zuordnung noch adaptiv zur Laufzeit geändert werden.
- **EXECUTORS:** Hier wird eine detaillierte Ansicht über die ausführende Person angegeben, welche Rolle bzw. Person für die Ausführung der Aufgabe benötigt wird. Ggf. kann die Zuordnung analog zur Tools-Zuordnung noch während der Runtime verändert werden.
- **DOCUMENTS:** Diese Sicht setzt die Verwaltung der zur Aufgabe gehörenden Dokumente um. Es können während der Aufgabenausführung neue Dokumente hinzugefügt, gelöscht oder verändert werden. Ist die Aufgabe noch nicht gestartet oder bereits beendet ist dieser Bereich zwar lesbar, kann aber nicht weiter bearbeitet werden.

Zusätzlich zu den oben aufgeführten Ansichten, die die jeweils ausgewählten Aufgaben betreffen, stehen dem Administrator weitere Ansichten zur Konfiguration des WfMS zur Verfügung. Hierfür sind in der obersten Leiste im Browserfenster von Abbildung 6.4 neben dem roten *Logout*-Button weitere zu sehen. Die Sichten und deren Funktionen werden im Folgenden aufgelistet und kurz die Funktionalität erklärt.

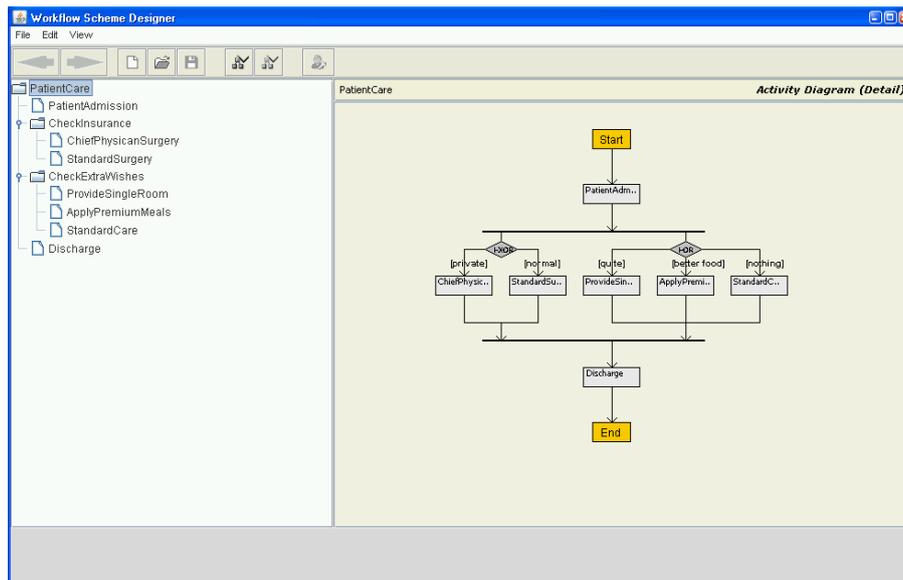
- **Workspace:** Diese Ansicht ist genau die, die in Abbildung 6.4 zu sehen ist. Es werden die Workflowinstanzen im linken Frame und im mittleren die spezifischen Ansichten für die selektierten Aktivitäten dargestellt. Im rechten werden des Weiteren bei selektierten Decisionnodes die geforderten Entscheidungen dem Nutzer in Form von Guards präsentiert. Außerdem ist hier die Möglichkeit gegeben über *PROCESS DEFINITIONS* weitere Prozessinstanzen von hochgeladenen Workflowmodelle zu erzeugen.
- **Project Center:** In diesem Bereich können neue Prozessmodelle in das WfMS geladen werden. Außerdem können hier aus den Prozessmodellen direkt Instanzen erzeugt werden. Das Modell muss also nicht erst ins Modellrepository geladen werden, um es zu instanzieren. Eine persistente Speicherung erfolgt in diesem Falle nur für die Modellinstanz.
- **User Center:** Hier werden die Nutzer und Rollen verwaltet. Es können neue angelegt oder wieder gelöscht werden. Außerdem ist hier die Passwortverwaltung vorhanden. Für Nutzer können des Weiteren Rollenzuordnungen vorgenommen oder auch wieder zurückgenommen werden.
- **Ressource Center:** Hier werden die Ressourcen, die zur Workflowausführung bereitgestellt werden, administriert. Neue Geräte können angelegt und die Anzahl der vorhandenen Geräte angepasst werden. Des Weiteren wird hier angegeben, wie viele Exemplare der entsprechenden Ressource derzeit in Benutzung sind.

6.3.2.2 Konfiguration der Workflowmodelle durch Instanzierungsoperatoren

In diesem Abschnitt wird das für TTMS umgesetzte Konzept der *Instantiationtime* vorgestellt. Es sollen die Prozessmodelle für die Ausführungsfälle bzw. Prozessinstanzen kurz vor der Runtime bzw. der Abarbeitung mit Hilfe des WfMS konfiguriert werden. In Abbildung 6.5 wird das Konzept und deren Nutzung in TTMS veranschaulicht.



(a) Konfiguration der Prozessmodelle durch WfMS zur Instantiationtime



(b) Modellierung von Instantiationtime-Operatoren zur Designtime

Choice: CheckInsurance					
	Guard	ID	Number	Name	Description
<input checked="" type="radio"/>	normal	Wed Jul 18 19:37:17 CEST 2012	2	StandardSurgery	
<input type="radio"/>	private	Wed Jul 18 19:36:52 CEST 2012	1	ChiefPhysicianSurgery	
<input type="radio"/> Decision in Runtime					
Choice: CheckExtraWishes					
	Guard	ID	Number	Name	Description
<input type="checkbox"/>	nothing	Wed Jul 18 19:39:53 CEST 2012	3	StandardCare	
<input checked="" type="checkbox"/>	better food	Wed Jul 18 19:39:15 CEST 2012	2	ApplyPremiumMeals	
<input checked="" type="checkbox"/>	quite	Wed Jul 18 19:38:41 CEST 2012	1	ProvideSingleRoom	
<input type="radio"/> Decision in Runtime					
<input type="button" value="Save Instantiation Decisions"/>					

(c) Ausführung der Instanzierungsoperatoren durch WfMS

Abbildung 6.5: Instantiationtime-Operatoren bei TTMS

Bei Geschäftsprozessmodellen ist das Konzept der Konfiguration von Referenzprozessmodellen schon länger Gegenstand der Forschung. So wurden für EPKs *XOR* und *OR*-Buildtime-Operatoren eingeführt, um allgemeine Referenzprozessmodelle für deren Gebrauch in spezifischen Branchen, Firmen oder Länder nutzbar zu machen [ST05]. Beispielsweise werden spezifische Verwaltungsprozesse mit dem gleichen Ziel

in verschiedenen Branchen oder Ländern unterschiedlich durchgeführt. Referenzprozessmodelle beinhalten hier in der Regel mehr Aktivitäten und Ausführungsmöglichkeiten als diese für bestimmte Firmen nötig wären. Die nicht benötigten Prozessstränge werden auf Basis von Auswertungen der *Buildtime*-Operatoren zur *Designtime* eliminiert. SAP-Referenzprozessmodelle liegen als EPKs vor und werden zur *Designtime* entsprechend der Regeln für das Unternehmen konfiguriert [Sch01]. Auf Basis dieser Modelleinstellungen kann dann die SAP-Standardsoftware für das Unternehmen ebenfalls konfiguriert und eingerichtet werden. Mit C-EPCs (Configurable Event Driven Process Chains) [RA07] wurde das Konzept der *Buildtime*-Operatoren formalisiert. Für das Workflow Management System YAWL wurde C-YAWL (Configurable Yet Another Workflow Language) entwickelt, bei der die YAWL-Referenzprozessmodelle an bestimmte Eigenschaften angepasst werden können [RGDA07]. Die Anpassung bezieht sich jedoch auf Konfigurationen für das Unternehmen während der *Designtime*. Daraufhin können beliebig viele Prozessinstanzen davon erzeugt werden, für die keine weitere Anpassung vorgesehen sind.

Im Gegensatz dazu muss die Anpassung sich nicht nur auf das Unternehmen und dessen Umfeld, sondern kann sich auch auf den spezifischen Geschäftsvorfall beziehen. In diesem Falle ist das Geschäftsprozessmodell bei jeder Instanziierung neu einzustellen. In [BF08b] wurde das *Instantiationtime*-Konzept für EPKs eingeführt, um die Prozessmodelle an kundenspezifische Wünsche anzupassen. Außerdem wurde in [APS09] das Konzept der *Instantiationtime* und die Konfiguration ebenfalls eingeführt, um Prozessmodelle adaptiver und anpassungsfähiger zu machen. Für TTMS sind die *Instantiationtime*-Operatoren umgesetzt und in [BF11] vorgestellt worden.

Im (adaptierten) Use Case-Diagramm von Abbildung 6.5(a) wird an einem Fallbeispiel verdeutlicht, wie das Konzept der *Instantiationtime* im Zusammenhang mit einem Prozess zur Patientenbetreuung im Krankenhaus eingesetzt werden kann. Ein neuer Krankenhausprozess wird anhand des Use Cases *accept case* angenommen. Dafür interagiert der *Receptionist* mit dem TTMS-WfMS und instanziiert einen neuen Prozess *PatientCare* im WfMS.

Bei der Instanziierung (zur *Instantiationtime*) hat der Nutzer den Prozess auf Grundlage von Fragen bzw. Entscheidungsfunktionen, die vom WfMS angegeben werden zu konfigurieren. Danach ist der Prozess *PatientCare* im WfMS erstellt. Die dafür zuständigen Rollen *Physician* und *Nurse* haben die Prozessinstanz daraufhin abzuarbeiten.

Ein Prozessmodell mit *Instantiationtime*-Operatoren sind in Abbildung 6.5(b) zu sehen. Es wurde sowohl die exklusive Variante mit dem *I-XOR* als auch die inklusive mit dem *I-OR*-Operator verwendet. Die exklusive ist mit dem Decisionnode *CheckInsurance* verknüpft. Hier hat der *Receptionist* zu prüfen, ob der Patient privat oder normal versichert ist, was durch die entsprechenden Guards im Prozessmodell ausgedrückt wird. Bei privat Versicherten ist für die Operation eine Chefarztbehandlung (*ChiefPhysicianSurgery*) vorgesehen. Dagegen ist bei einer normalen Versicherung eine Standardbehandlung ohne besonderen Personals (*NormalSurgery*) durchzuführen.

Der zweite *Instantiationtime*-Choice ist mit dem Decisionnode *CheckExtraWishes* verknüpft. Hier ist zu prüfen, ob der Patient eine Einzelzimmerbetreuung wünscht, die durch Guard *quite* ausgedrückt wird. Bessere Verpflegung wird durch die Aktivität *ApplyPremiumMeal* durchgeführt, die mit dem Guard *better food* verknüpft ist. Sind keine extra Wünsche vorhanden (mit dem Guard *nothing* versehen), so wird durch die Aktivität *StandardCare* die normale Betreuung durchgeführt.

Im Prozessmodell *PatientCare* sind des Weiteren die nicht weiter konfigurierbaren Aktivitäten *PatientAdmission* und *Discharge* für die Ein- und Ausweisung des Patienten modelliert.

Bei der Instanziierung des Prozessmodells im WfMS wird dem Nutzer das Fenster von Abbildung 6.5(c) präsentiert. Hier werden *Instantiationtime*-Entscheidungsoperatoren vom Modell aus Abbildung 6.5(b) aufge-

löst. Bei dem Instantiationtime-Operator *CheckInsurance* sind für die exklusive Auswahl zwei Radiobuttons angegeben. Der Nutzer kann sich dadurch nur für genau eine Alternative entscheiden.

Bei der Interpretation des inklusiven *I-OR* Decisionnodes *CheckExtraWhishes* werden die Alternativen mit Checkboxes abgefragt. Hier kann der Nutzer entsprechend mehrere Alternativen auswählen, die daraufhin im Prozessmodell aktiviert werden. Die nicht ausgewählten werden für die auszuführende Prozessinstanz dagegen deaktiviert.

Zusätzlich wird bei beiden Operatoren mit *Decision in Runtime* angeboten, die Entscheidung an dem Ausführungspunkt im Prozessmodell nochmal abzufragen, an der der Decisionnode modelliert ist. Die Verschiebung der Entscheidung in die Runtime ist analog auch bei den *C-EPCs* vorgesehen [RA07]. Dieses Konzept wurde für die Instantiationtime-Operatoren entsprechend bei TTMS übernommen.

6.3.2.3 Toolchain

Im Zusammenhang mit Aufgabenmodellen sind diverse Modellierungs- und Simulationstools entstanden. In dieser Arbeit sind mit MCTT und TTMS im Kontext der Workflowmodellierung zwei weitere hinzugekommen. Um bestehende und die hier entwickelten Tools im Zusammenhang darzustellen, wird auf Basis einer Toolchain in Abbildung 6.6 ein Entwicklungsprozess für Aufgabenmodell-basiertes Workflow Management vorgeschlagen.

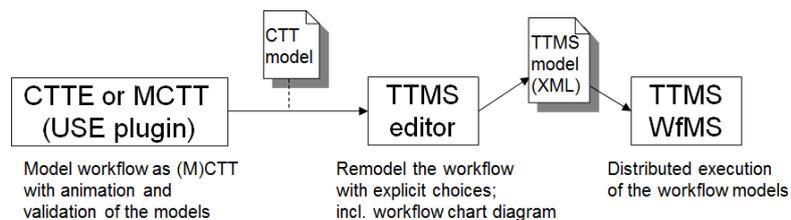


Abbildung 6.6: Eine Werkzeugkette im Zusammenhang mit Verwendung von TTMS

Aufgabenmodelle könnten lokal simuliert und validiert werden, bevor man sie für das WfMS erstellt und ausführt. Abschnitt 6.4 zählt einige Aufgabenmodellierungsansätze mit unterschiedlicher Syntax auf. Der Modellierungssprache für die TTMS-Workflowmodelle ist sehr ähnlich zur CTT-Syntax. Beide verwenden binäre und unäre Operatoren zur Spezifikation der Ablauflogik im Aufgabenbaum. Beachtet man, dass nicht alle CTT-Operatoren in TTMS umgesetzt sind, so lässt sich die entsprechende Untermenge der temporalen Operatoren recht einfach von CTT in TTMS übersetzen. Somit können zunächst im CTTE-Tool CTT-Modelle erstellt und lokal simuliert und validiert werden. Statt des CTTE-Tools kann auch das USE-Tool mit dem in Abschnitt 6.2.2 vorgestellten MCTT-Plugin verwendet werden. Dieser Ansatz wäre sogar näher an der Benutzung eines Workflow Management Systems, da die dortige Ausführung nicht eventbasiert auf der Basis einer Prozessalgebra passiert, sondern wie bei TTMS auf Ausführungszuständen.

Sind die Workflowmodelle validiert und entsprechen den Vorstellungen des Endanwenders, können diese im TTMS-Editor nachentwickelt werden. Ein Austauschformat zwischen CTTE ist nicht umgesetzt worden, wäre hier aber natürlich wünschenswert. Vor allem wenn sich Veränderungsbedarfe herausstellen und Modelle restrukturiert werden sollen, wäre ein Austauschformat nützlich, um die veränderten Modelle konsistent zu halten. Die manuelle Übertragung vom CTTE- bzw. MCTT-Tool zum TTMS-Editor wird mit dem über die gestrichelte Linke verbundenen Artefakt (*M)CTT model* an dem Pfeil in Abbildung 6.6 ausgedrückt.

Ist der Workflow nachmodelliert, so wird das Aufgabenmodell im TTMS-Editor als strukturiertes Fluss-

diagramm ähnlich zu einem strukturierten Aktivitätsdiagramm dargestellt. Entsprechende Beispiele sind in Abbildungen 6.3 und 6.5(b) gegeben. Diese Darstellung mag konventionellen Workflowmodellierern vertrauter sein, als die CTT-Syntax. Im Editor sind des Weiteren die vom Nutzer zu treffenden expliziten Entscheidungen anhand von Decisionnodes einzufügen.

Ist das Prozessmodell daraufhin im XML-Austauschformat von TTMS abgespeichert, so kann es in das TTMS-WfMS hochgeladen und benutzt werden. Dieser Sachverhalt wird über eine Objektflussdarstellung in Abbildung 6.6 ausgedrückt.

6.4 Weitere Ansätze

Es gibt einige Ansätze zur Simulation und Validation von Aufgabenmodellen. Die populärste Aufgabenmodellierungssprache ist CTT und hat eine Entwicklungsumgebung mit CTTE [MPS02]. Eine integrierte Simulationsmöglichkeit zur Validation ist im Tool umgesetzt, um die dynamischen Modelleigenschaften untersuchen zu können.

Eine an CTT angelehnte, aber ohne eine grafische Modellierungs- und Simulationsumgebung ist in [DFHS05, KDF10] entwickelt worden. Hier werden anhand von Formeln die Aufgabenmodelle textuell erstellt. Des Weiteren gibt es ein konsolenbasierte Animationsmöglichkeit, um die Kontrollflusseigenschaften der Modelle zu validieren. Eine ähnliche Ausführung wird in [Ric09] verfolgt.

Ein weiterer diesmal visueller Modellierungsansatz auf Basis der von Eclipse bereitgestellten EMF und GMF-Technologien [Gro09] wurde in [RF08, BDFR07] präsentiert. Die Ausführung von Aufgabenmodellen ist ebenfalls Thema in [Uhr03]. Ein weiterer Einsatz von Aufgabenmodellen wurde im Zusammenhang mit einer von CTT abweichenden Syntax in [Bom07] vorgestellt. Die temporalen Operatoren sind hier nicht mehr binär sondern n-är. Eine ähnliche Aufgabenmodellierungssprache mit n-ären temporalen Operatoren wird in [RF08] präsentiert.

Anstatt einer eventbasierten Prozessalgebra wurde für die Definition der Ausführungssemantik auf Zustandsautomaten bei [Bom07, RF08] zurückgegriffen. Das gleiche gilt für die in dieser Dissertation eingeführten MCTT- und TTMS-Ansätze.

Mit den vielen vorgestellten teilweise sehr ähnlichen Sprachen existieren unterschiedliche Semantiken bei gleichen Operatoren. U.a. ist das strikte Verbot von CTT, einen Sequenzoperator hinter einer Iteration zu verwenden bei den in [DFHS05, KDF10] vorgestellten Ansätzen nicht vorgesehen. Die unterschiedliche Verwendung des *Disabling* Operators von CTT im Vergleich zu LOTOS ist hier ein weiteres Beispiel, das die nicht eindeutige operationale Semantik der temporalen Operatoren in Aufgabenmodellen verdeutlicht. Bei der n-ären Verwendung dieses Operators stellt sich des Weiteren die Frage, welche Aufgabe welche andere beenden darf und ob auch hier alle Aufgaben ausgeführt werden müssen. Die in Tabelle 5.2 vorgestellte Semantik der CTT-Operatoren gilt also sicherlich nicht analog zu anderen Aufgabenmodellierungssprachen. Dort können zwar gleiche oder ähnliche Operatoren genutzt werden, diese können aber u.U. eine andere operationale Semantik haben.

Die Konfiguration vor der Ausführung von Aufgabenmodellen wird in [BGHM11] behandelt. In diesem Zusammenhang gibt es des Weiteren die Decisionnodes zur Konfiguration der Aufgabenbäume [Luy04]. Um die Aufgabenmodelle für spezielle Einsatzzwecke anzupassen, wurde in [WFRS07] ein Konfigurationstool für Task-Patterns entwickelt.

Bisher wurden Aufgabenmodelle im Kontext der User Interface-Entwicklung verwendet. Sie haben aber auch in weiteren Feldern Einsatz gefunden. Beispielsweise wurden Aufgabenmodelle in [Wur11] zur nutzerzentrierten Aktivitätsmodellierung in intelligenten Umgebungen eingesetzt. Mit dem in dieser Dissertation

verfolgten Ansatz finden sie außerdem Anwendung in der Workflowmodellierung.

TTMS ist das erste Workflow Management System, das eine integrierte Zielmodellierung in strukturierten Workflowmodellen verfolgt. Ein Workflow Management mit einer Baum-basierten Workflowmodellierungssprache wird in [Wes99] verwendet. Zwischen den Aktivitäten sind Datenflüsse zu modellieren, die keiner bestimmten Struktur folgen müssen. Im Gegensatz zu den strukturierten CTT-Modellen können diese unübersichtlich wirken.

Das wohl populärste freie Workflow Management System ist mit *Yet Another Workflow Language* umgesetzt worden. Die Workflowsprache fußt auf der Petri-Netz Semantik und setzt alle Workflow Patterns um [HAAR09]. Das Datenmodell ist anhand von XML integriert worden. Außerdem lassen sich die WfMS mit dem Editor verbinden, um das Ressourcen- bzw. Organisationsmodell zu synchronisieren. Damit lassen sich während der Designzeit Rollen- bzw. Personen zu Aktivitäten spezifizieren. Eine auf diesen Regeln basierende Zuordnung kann vom WfMS dann automatisch erfolgen.

ADEPT ist ein Workflow Management System [DRRM11], das den Fokus auf Flexibilität und Adaptivität legt. Es unterstützt den Nutzer während der Ausführung von Modellinstanzen, diese bei Bedarf zu verändern. Gewisse Modellbereiche müssen dabei geschützt, andere dürfen editiert werden. Diese Funktionalität im Zusammenhang mit einer adäquaten Benutzerschnittstelle zum Editieren der Modelle über das WfMS zur Runtime wird von *ADEPT* bereitgestellt. Dabei wird das Prinzip *flexible by change* [SMR⁺07] verfolgt.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurden zwei Ansätze zur Workflowmodellierung verfolgt, die jeweils verschiedene Aspekte der Geschäftsprozessmodellierung in den Vordergrund stellen. Die deklarative UML-metamodellbasierte Workflowmodellierung (DMWM) repräsentiert eine flexible Sprache, mit der Workflowmodelle erstellt und durch ein UML-Tool ausgeführt werden können. Der Ansatz folgt dem deklarativen Prinzip, das beliebige Ausführungsreihenfolgen der Aktivitäten erlaubt, solange sie nicht durch Constraints verboten sind.

Die Hierarchisierung, Strukturierung und integrierte Zielmodellierung charakterisieren die zweite Herangehensweise zur Workflowmodellierung. Im Bereich der Aufgabenmodellierung gibt es bereits etablierte Ansätze dafür. Darauf aufbauend wurden die *metamodellbasierten ConcurTaskTrees* (MCTT) und für das *Task Tree Based Workflow Management System* die TTMS-Workflowsprache entwickelt. Sie enthalten für die Workflowmodellierung wesentliche Erweiterungen zur Entscheidungsmodellierung, Datenmodellintegration und Soundness-Analyse.

UML-Metamodelle dienen als Basis für die Umsetzung der Sprachen DMWM und MCTT. OCL-Invarianten aus dem Metamodell werden mit Hilfe des UML-Tools USE ausgewertet, um Soundness-Eigenschaften der Workflowmodelle zur Designzeit zu prüfen. Darüber hinaus erlaubt der Metamodellansatz, die operationale Semantik der entwickelten Sprache in das Metamodell zu integrieren. Die imperativen OCL-basierten Sprachen ASSL und SOIL dienen dazu, die Effekte der Nutzerinteraktion und die operationale Semantik der Workflowsprachen umzusetzen. Diese Herangehensweise zeichnet sich gegenüber anderen metamodellbasierten Ansätzen aus, die nur die Syntax der entwickelten Sprachen ausdrücken (Beispiele dafür sind die Metamodelle für UML-Aktivitätsdiagramme und BPMN). Zwei Schnittstellen für unterschiedliche Szenarien stehen zur Verfügung: Für die lokale Modellausführung in Verbindung mit dem UML-Werkzeug USE erweitert ein Plugin die Oberfläche um eine Nutzerschnittstelle zur MCTT-Workflowsteuerung. Kooperative, verteilte Ausführung von Aufgabenmodellen ist im Workflow Management System TTMS umgesetzt.

Der folgende Abschnitt 7.1 zieht das Fazit für diese Arbeit und bewertet die erzielten Ergebnisse. Die zwei metamodellbasierten Sprachen zur Workflowmodellierung DMWM und MCTT werden verglichen. Daraufhin diskutiert Abschnitt 7.2 die Ergebnisse dieser Arbeit. Es wird abgewogen, ob sich die Ziele der beiden Ansätze in einer integrierten Sprache, die durch die Vereinigung der Metamodelle erreicht werden könnte, verbinden lassen. Abschnitt 7.3 behandelt schließlich Fragen zur Erweiterung und Verbesserung der neuen Modellierungsweisen und Tools als Anregung für aufbauende Arbeiten.

7.1 Fazit

Die deklarative UML metamodelbasierte Workflowmodellierung (*DMWM*) verfolgt einen flexibleren Ansatz als konventionelle, imperative Sprachen. Sie setzt die deklarative Modellierungsphilosophie um, die dem Nutzer zunächst mehr Freiheiten bei der Modellausführung einräumt. Die Grundannahme ist, dass alle Aktivitäten unabhängig voneinander sind. Weitere Sprachelemente können demnach den Entscheidungsraum für die Ausführung nicht erweitern bzw. aufspannen, sondern nur einschränken.

Demgegenüber sehen konventionelle Sprachen (EPKs oder BPMN) die Modellierung bestimmter Reihenfolgen von Aktivitäten als Prozessketten von einem definierten Start bis zum definierten Ende vor. Das Hinzufügen von Modellierungselementen (z.B. *forks* und *joins*) entspricht hier einer Erweiterung der Ausführungsmöglichkeiten. Das definierte Ende gibt explizit an, wann der Prozess terminiert. Die Herangehensweise bei *DMWM* ist insofern flexibler, als dass Anfang und Ende nur implizit definiert werden. Die Aktivitäten, die zu Beginn auf Basis der Ausführungseinschränkungen ausführbar sind, geben den Start an. Das Ende wird implizit dadurch definiert, dass keine weiteren Aktivitäten ausgeführt werden können.

Die Workflow Pattern-Analyse hat gezeigt, dass die Sprache *DMWM* mächtig ist, indem sich alle Kontrollfluss-Patterns ausdrücken lassen. Die Patterns dienen als akzeptierter Benchmark für Workflowsprachen und nicht viele Sprachen können alle ausdrücken [AHKB03]. Daraufhin wurde die Modellierungsweise und die Anwendung des Modellierungswerkzeugs gezeigt. Einige Patterns (z.B. *Deferred Choice* und *Interleaved Parallel Routing*) lassen sich einfacher und mit weniger Modellierungselementen als bei konventionellen Modellierungssprachen ausdrücken. Mit OCL-Invarianten und Tools zur Auswertung konnte eine Soundness-Analyse der Workflowmodelle zur Designzeit ermöglicht werden, ohne dass der Modellierer zur Identifizierung der Fehler über spezifischen OCL-Kenntnisse verfügen muss. Über verschiedene Sichten werden die Invarianten und die problematischen Modellobjekte angegeben, die die Inkonsistenz hervorrufen. Für OCL-versierte Nutzer steht ein OCL-Debugger bereit, der detaillierte Angaben über die Inkonsistenzen vermittelt.

Der Ansatz schließt mit der Daten- und Organisationsmodellierung zwei weitere wichtige Teilbereiche der Workflowmodellierung ein. Der Modellierer definiert dazu u.a. mit OCL Datenanfragen zur Designzeit, die vom Runtime-Plugin dann interpretiert werden. Eine Analyse zeigte, dass z.B. bei Aktivitätsdiagrammen und BPMN Uneindeutigkeiten hinsichtlich der Datenmodellierung bestehen, die u.a. eine Ausführung dieser Prozessmodelle verhindert.

Diese Arbeit folgte dem Grundsatz, dass Modellierungssprachen visuell sein müssen. Der zunächst verfolgte Ansatz, (OCL-)Formeln anzuwenden, um Workflows deklarativ zu modellieren, erwies sich im Laufe der Entstehung dieser Arbeit als nicht praktikabel. Eine Modellierungssprache muss intuitiv und leicht verständlich sein, wofür die Visualisierung einen wichtigen Beitrag leistet. Diesbezüglich ist für die entwickelten Modellierungssprachen *DMWM* und *MCTT* ein Mangel festzustellen. Sie haben zwar mit den verwendeten UML-Objektdiagrammen eine visuelle Syntax, die jedoch recht abstrakt ist und der Typenvielfalt der Modellierungselemente noch nicht gerecht wird. Es stehen dem Modellierer ausschließlich UML-Objekte und Verbindungen zwischen diesen (Links) zur Verfügung. Die verschiedenen Modellelemente lassen sich im UML-Objektdiagramm nur anhand der Typangabe identifizieren, nicht aber durch unterschiedliche Symbole. Eine deutlich intuitivere Visualisierung ruft eine konkrete Syntax der Sprachen hervor, bei der u.a. Aktivitäten, Datenobjekte und Akteure sich visuell voneinander unterscheiden.

Ein Vorteil der Sprachen *DMWM* und *MCTT* ist mit der unmittelbaren Möglichkeit der Modellausführbarkeit gegeben. Die Modelle können sogar während der Runtime verändert werden. Somit sind sie bei *DMWM* auf zwei Arten flexibel: Erstens durch die bereits diskutierte deklarative Modellierungsweise und zweitens durch die Adaptivität zur Runtime.

Das Metamodell gibt für die DMWM-Workflowmodelle keine festgelegte Struktur vor. Es liegt hier analog zu EPKs, UML-Aktivitätsdiagrammen und BPMN am Modellierer, übersichtliche und verständliche Modelle zu erstellen. Hierbei hat sich herausgestellt, dass sie recht schnell unstrukturiert und unübersichtlich werden können. Im Gegensatz zu DMWM gibt das Metamodell von MCTT eine gewisse Struktur für die Workflowmodelle vor. Insbesondere bei großen Prozessmodellen ist diese notwendig, um die Modelle verständlich zu gestalten.

Ein alternatives Layout für MCTT-Modelle, mit horizontaler Dekomposition der Aufgaben, kann große Modelle kompakter darstellen. Die Baumdarstellung hat bei der vertikalen Dekomposition den Nachteil, dass bei vielen Blattaufgaben nebeneinander geschrieben der Platzbedarf des Baums sehr groß werden würde. Bei einer Anordnung, die die Blattknoten untereinander aufführt, kann eine kompaktere Darstellung erfolgen. Somit wurde das Ziel mit MCTT erreicht, auch größere Modelle verständlich und strukturiert visualisieren zu können.

Die meisten Sprachen zur Aufgabenmodellierung sind informell und es werden mit bestimmten temporalen Operatoren unterschiedliche Semantiken assoziiert, was eine verlässliche Soundness-Analyse der Modelle verhindert. Transformationen zu Aktivitätsdiagrammen verdeutlichen die Semantiken der CTT-Operatoren genauer. Auf dieser Basis wurde ein metamodellbasierter Ansatz für CTT (MCTT) entwickelt, der die Modellierungselemente so formalisiert, dass die Soundness zur Designtime überprüft werden kann. Für Workflowmodelle ist diese Möglichkeit sehr wichtig und ermöglicht dem aufgabenbasierten Ansatz neue Anwendungsgebiete im Bereich der Workflowmodellierung.

Sowohl bei DMWM als auch bei MCTT wurden die Workflowmodelle mit visuellen Datenmodellen verbunden. Zur Designtime dienen UML-Objektdiagramme der Workflowmodellierung und UML-Klassendiagramme der Datenmodellierung. Zur Runtime wird dann ein integriertes UML-Objektdiagramm sowohl für Workflowinstanzen als auch für die Ausprägung des Datenmodells verwendet. Ein Nachteil liegt hierbei in der Verwendung unterschiedlicher Diagrammart für die relevanten Modelle zur Designtime.

Das Plugin zeigt zur Runtime die Prozessinstanzen aus dem UML-Objektdiagramm kompakt und strukturiert in einer Baumdarstellung an. Im Fokus der Betrachtung stehen zur Runtime die aktivierten und laufenden Aktivitäten, die dem Nutzer anzuzeigen sind. Die Darstellung des gesamten Workflowmodells spielt hierbei keine Rolle. Das Plugin hat sowohl Modelle von DMWM als auch MCTT für die Runtime dargestellt. Die operationale Semantik der einzelnen Sprachen ist im Metamodell integriert, so dass das Plugin nur die Nutzerschnittstelle bereitstellt und somit weitestgehend unabhängig vom Metamodell ist. Dadurch ist es möglich, weitere Sprachen über neue Metamodelle umzusetzen und die Ausführung von Instanzen der modellierten Workflows mit Hilfe des Plugins zu steuern.

Die bisher diskutierten metamodellbasierten Ansätze waren nur lokal ausführbar. Ein Workflow Management System, das DMWM und MCTT als Sprachen nutzt und auf USE als Workflow Engine zur Interpretation der Modelle zugreift, ist nicht umgesetzt worden. Stattdessen wurde mit dem *Task Tree Based Workflow Management System* (TTMS) ein Workflow Management System implementiert, das in den Workflowmodellen Prozessalgebra-Terme in der hierarchischen Anordnung der Aufgaben verwendet. Aufgaben können mit den vier temporalen Operatoren *Sequence*, *Choice*, *Concurrency* und *Iteration* verknüpft werden, was die Ausdrucksmächtigkeit der Sprache sehr beschränkt. Es hat sich vor allem beim *Choice* herausgestellt, dass die implizite Entscheidungsmodellierung für Workflowmodelle bei der Verwendung mit Workflow Management Systemen nicht ausreicht. Dem Nutzer sollte während der Ausführung des Prozesses bewußt sein, wenn er bei einer Prozessverzweigung über den einzuschlagenden Prozesspfad entscheidet. Mit impliziten Choices muss das nicht der Fall sein, was für Verwirrung bei dem Nutzer sorgen kann, wenn Aktivitäten plötzlich implizit übersprungen werden.

Bei anderen Workflow Management Systemen wie z.B. YAWL ist die explizite Entscheidungsmodellierung

über das Datenmodell gelöst worden. Das Workflowmodell bezieht also Datenelemente in die Spezifikation der Ablauflogik mit ein, was sehr nützlich und wichtig ist. Bei TTMS ist der Datenaspekt mit einer Dokumentensicht zwar integriert, hat aber keinen Einfluss auf die Ausführungslogik. Hier wäre analog zu YAWL oder MCTT ein datenbasierter Choice wünschenswert gewesen. Bei TTMS wurde die explizite Entscheidungsmodellierung stattdessen durch Decisionnodes implementiert, die an die Entscheidungsmodellierung von EPKs angelehnt ist.

Die Praktikabilität der vorgestellten Ansätze zur Workflowmodellierung auf Basis der entwickelten Softwareprototypen konnte anhand zahlreicher Modelle unterschiedlicher Komplexität evaluiert und bestätigt werden. Auch größere Modelle wurden damit erzeugt. So wurde z.B. ein Kfz-Inspektionsprozess einer Autowerkstatt mit DMWM und MCTT modelliert, der jeweils aus ca. 60 Aktivitäten und insgesamt über 100 Modellelemente bestand. Die Ausführung der Workflowmodelle ist bei den metamodellbasierten Ansätzen direkt im Modellierungstool realisiert worden. Auch die Ausführung der großen Modelle war mit dem jeweiligen Runtime-Plugin möglich, wobei sich gezeigt hat, dass die Ausführung auf Basis von SOIL (bei MCTT) deutlich schneller war als die mit ASSL (bei DMWM).

7.2 Weiterführende Diskussion

Metamodelle sind dazu geeignet, beliebige Aspekte bzw. Eigenschaften für Workflowsprachen zu definieren. Alle Aspekte in einem allumfassenden Metamodell aufzunehmen, wäre mit dem hier gewählten Ansatz und Tool technisch durchaus realisierbar. Jedoch stellt sich nicht nur die Frage nach der Sinnhaftigkeit dieser Idee, sondern auch die nach der Vereinbarkeit der Ziele, welche die zugrundeliegenden Philosophien der einzelnen Sprachen verfolgen. Folgende Gründe führen zu einer skeptischen Beurteilung dieser Fragen.

Eine wichtige Eigenschaft bei den MCTT-Modellen ist die Strukturiertheit, die durch das MCTT-Metamodell vorgegeben ist. Werden nun Elemente aus anderen Sprachen wie z.B. DMWM aufgenommen, so geht die vorher garantierte Strukturiertheit zugunsten der Mächtigkeit verloren. Die Situation würde sich weiter verschärfen, wenn zusätzlich Teile aus den Metamodellen von UML-Aktivitätsdiagrammen oder BPMN für eine imperative Prozessmodellierung in das integrierte Metamodell aufgenommen werden würde. Ein solches Metamodell stellt zur deklarativen und hierarchieorientierten Workflowmodellierung die Möglichkeit bereit, auch konventionelle Prozessmodelle zu erstellen. Somit wäre eine Sprache, die über ein solches Metamodell definiert wird, sehr mächtig und könnte weit mehr ausdrücken als die einzelnen einfachen Sprachen.

Jedoch birgt ein solcher Ansatz gravierende Nachteile. Zusätzlich zur Gefahr, dass die Prozessmodelle recht unübersichtlich werden, ergibt sich das Problem inkonsistenter Semantik. Ein Widerspruch ist beispielsweise mit der expliziten Angabe des Prozessendes bei imperativen und der impliziten Angabe bei deklarativen Prozessmodellen gegeben. Es erscheint daher sinnvoller, Metamodelle zu verwenden, bei denen die Modellierungsphilosophie und Semantik der Modellierungselemente klar definiert und konsistent ist. Ein integriertes allumfassendes Metamodell kann eine solche Sprache nicht bereitstellen.

Dagegen ist eine einfach zu realisierende Zusammenstellung der Modellierungselemente über die Verwendung von Metamodellen durchaus möglich. Metamodelle können zur Definition von domänenspezifische Sprachen eingesetzt werden und damit genau die vom Nutzer gewünschten Modellierungselemente definieren und bereitstellen [Kle08]. Genauso können Workflowmodellierungssprachen an Nutzeranforderungen recht einfach adaptiert werden. Da sich Modellierungselemente widersprechen können, ist aber auf die Konsistenz der Sprache zu achten. OCL-Invarianten können hier helfen, wie es für DMWM und MCTT in dieser Arbeit gezeigt wurde.

7.3 Ausblick

Einige Fragestellungen und offene Punkte sind während der Bearbeitung des Themas aufgetreten, die hier erläutert werden. Die Erweiterungsmöglichkeiten betreffen sowohl die Design- als auch die Runtime der entwickelten Sprachen. Des Weiteren lassen sich viele weitere Ansätze mit den hier vorgestellten Konzepten bzw. Tools verbinden.

Wie bereits im Fazit angesprochen und in [MRR10] analysiert, sind Symbole zur Workflowmodellierung wichtig und für das Modellverständnis förderlich. So wäre eine Umsetzung einer konkreten Syntax für die Sprachen DMWM und MCTT sehr wünschenswert. Das UML-Objektmodell kann in USE zwar weiterhin genutzt werden und das Workflowmodell abstrakt darstellen, zusätzlich sollte aber ein USE-Plugin für die Designtime das Objektmodell mit einer konkreten Syntax darstellen. Dort werden die Objekte je nach Typ mit unterschiedlichen Symbolen visualisiert. Für die Designtime ist die Darstellung des kompletten Modells als flexible anzuordnender Graf wichtig. Für die Ausführung setzt das entwickelte Runtime-Plugin die Prozessdarstellung und Nutzerschnittstelle um. Hier werden die relevanten Aktivitäten dagegen über verschiedenen Sichten in einem Baum oder Listen (Worklists) angeordnet.

Es können diverse weitere Sprachen durch Metamodelle ausgedrückt werden. Auch Petri-Netze gehören dazu, für die u.a. in [JKBL10] ein UML-Metamodell entwickelt wurde. Speziell auf Workflows bezogen ist eine Untersuchung interessant, die analysiert, ob OCL in der Lage ist, den von van der Aalst definierten Soundness-Begriff für Workflow-Netze [AS11] metamodellbasiert zu spezifizieren. Auf Grundlage dessen ließe sich mit dem UML-Tool USE Workflow-Netze modellieren, die just-in-time während der Designtime auf Konsistenzeigenschaften geprüft werden könnten.

Auch eine Ausführung der Workflow-Netze ist mit den Techniken, die in dieser Arbeit verwendet wurden, denkbar. Die operationale Semantik der Petri-Netze basiert jedoch nicht auf Zustandsdiagrammen, so wie bei den in dieser Arbeit entwickelten Sprachen. In [Wac08] ist hierfür ein Petri-Netz-Metamodell entwickelt worden, das auch die Ablaufsemantik für die Runtime umsetzt. Die operationale Semantik eines solchen Metamodells lässt sich auch durch die OCL-basierten imperativen Sprachen ASSL oder SOIL umsetzen. Das Plugin zur Darstellung der Prozessinstanzen und Nutzerschnittstelle ist zwar weitestgehend generisch gehalten, basiert jedoch auf Zustandsänderungen, die vom Nutzer aufgerufen werden. Die Darstellung und Ausführung der Modelle ist damit nicht unmittelbar adaptierbar, wie das z.B. ausgehend von der Sprache DMWM zu MCTT möglich war.

Um die in dieser Arbeit entwickelten Sprachen und Tools für eine nutzerorientierte Validierung der Prozessmodelle zu erweitern, können die Prozessmodelle mit Kontextinformationen, wie z.B. Bildern, angereichert werden. Die *Executable Use Case* Modellierung verfolgt einen solchen Ansatz [JTF09]. Mit so einer Erweiterung wird die Anschaulichkeit der Modellausführung für den Nutzer erhöht und damit die Diskussionsgrundlage verbessert. Um dieses zu erreichen, ist eine Erweiterung sowohl der Modellierungs- als auch der Ausführungsumgebung für DMWM bzw. MCTT notwendig.

Die Modelle sind bei DMWM und MCTT adaptiv, indem sie zur Runtime editiert werden können. Da jedoch gewisse Modellierungselemente zur Runtime geschützt werden müssen, ist hier zunächst eine genauere Analyse sinnvoll, die diese Elemente identifiziert. Auf dieser Basis kann dann ein Runtime-Modelleditor für USE entwickelt werden, der die entsprechenden Elemente schützt und die anderen zum Editieren freigibt.

Bezüglich der Modellausführung gibt es bei DMWM und MCTT nicht mehrere Rollen-Perspektiven für die Prozessausführung. Mit DMWM wurde eine Organisationsmodellierung integriert, jedoch werden die unterschiedlichen Perspektiven bei der Prozessausführung nicht berücksichtigt. Hier ist eine Erweiterung des Plugins sinnvoll, die jeder Rolle eine Perspektive auf den Prozess bereitstellt und die ihr zugeordneten

Aktivitäten darstellt.

Bei MCTT fehlt eine Organisationsmodellierung bereits zur Designtime. Um diese konzeptionell zu integrieren, ist ein an CCTT angelehnter Ansatz denkbar [Pat99]. Es können hier unterschiedliche Aufgaben bestimmten Rollen zugeordnet werden. Die Kooperation wird über einen separaten Baum modelliert analog zu CCTT. Die CCTT-Modelle sind im CTTE-Tool simulierbar [MPS02]. Eine ähnliche Erweiterung ist für MCTT und das Runtime-Plugin auch denkbar.

Abschließend ist zu sagen, dass diese Arbeit vielversprechende Perspektiven aufzeigt, wie Metamodelle und OCL im Bereich der Workflowmodellierung eingesetzt werden können. Insbesondere im Zusammenhang mit Aufgabenmodellen leisten sie wichtige Grundlagen zur Formalisierung. Ebenso können sie Beiträge für andere Sprachen liefern. Diese zu untersuchen wäre sehr interessant geht aber über den Rahmen dieser Arbeit hinaus.

Tabellenverzeichnis

4.1	Die für DMWM verwendeten ASSL-Befehle und Erklärungen dazu	70
4.2	Die verwendeten UML-Diagramme für DMWM	72
4.3	Die Farbzusordnungen zu den Zuständen	76
5.1	Temporale CTT-Operatoren	96
5.2	Transformation der CTT-Operatoren in Aktivitätsdiagramme	97
5.3	Transformation der unären CTT-Operatoren in Aktivitätsdiagramme	99
6.1	Temporale TTMS-Operatoren	121

Abbildungsverzeichnis

2.1	Eine Kreditantragsprozesse modelliert mit einer einfache Ereignisgesteuerten Prozesskette	18
2.2	Das ARIS Konzept und Beispielmodelle dafür	19
2.3	Aktivitätsdiagramm und zugehörige UML Klassen- und Zustandsdiagramme	20
2.4	Ein Kreditantragsprüfungsprozess als BPMN Geschäftsprozessmodell	22
2.5	Ausschnitte der Metamodelle für EPKs, Aktivitätsdiagramme und BPMN	26
2.6	Ausschnitt des ARIS Metamodells für Funktionssicht	28
3.1	Bankenbeispiel als UML-Klassendiagramm zur OCL-Einführung	33
3.2	Das Metamodel für den <i>DMWM</i> -Ansatz	34
3.3	Lebenszyklen von Aktivitätsobjekten modelliert in UML-Zustandsautomaten	37
3.4	Basic Control Flow Patterns in <i>DMWM</i> -Prozessmodellen	43
3.5	Advanced Branching and Synchronization Patterns	45
3.6	Multiple Instance Patterns	46
3.7	State-based Patterns	47
3.8	Iteration, Cancellation and Force Completion Patterns	48
3.9	Ein <i>DMWM</i> -Workflowmodell für einen Krankenhaus-Notfallprozess als Objektdiagramm	49
3.10	Modell, das die zwei OCL-Soundness-Invarianten verletzt	50
3.11	Datenintegration in den Modellierungssprachen EPK, BPMN, Aktivitäts- und Datenflussdiagrammen	52
3.12	Metamodellerweiterung für Datenintegration in <i>DMWM</i>	54
3.13	Daten- und Workflowmodell mit Datenintegration	55
3.14	<i>DMWM</i> -Metamodel für die Organisationsmodellierung	58
3.15	Organisationsmodell für Krankenhaus und Aktivitäts-Allokationsmodell	59
3.16	Einsatz des USE Tools zur Workflowmodellierung	62
3.17	Modellierungsoberfläche vom UML-Werkzeug USE	63
3.18	Toolunterstützung zur Identifizierung von Soundness-Problemen	65
4.1	Anwendungsfälle für <i>DMWM</i> zur Runtime und das entwickelte Workflow-Plugin	68
4.2	<i>DMWM</i> -Designtime-Plugin	74
4.3	Übersicht über Entwicklungsprozess von <i>DMWM</i> -Modellen und Nutzung der unterschiedlichen Werkzeuge	75
4.4	Verschiedene Sichten auf Workflow im Runtime-Plugin und Optionsauswahlmenü	77
4.5	Anzeigen der Entscheidungsaktivitäten während der Runtime	79

4.6	Ausführung der Iterationaktivität	80
4.7	Workflow-Modell mit Multiinstanz-Aktivitäten und Ausführung im Runtime-Plugin	81
4.8	Ausführung der Iterationsgruppe	82
4.9	Darstellung von Daten während der Workflowausführung	83
4.10	Das Log-Fenster vom Runtime-Plugin	85
4.11	Ein Szenario des Notfallprozesses in einem Sequenzdiagramm dargestellt	86
5.1	Hierarchische Modelle	92
5.2	Darstellung der Hierarchie in Aktivitätsmodellen	95
5.3	Problematische CTT-Modelle	101
5.4	Das Metamodel für den <i>MCTT</i> -Ansatz	102
5.5	Lebenszyklen von Aufgabenobjekten modelliert in UML-Zustandsautomaten	104
5.6	Inkonsistente <i>MCTT</i> -Aufgabenmodelle	106
5.7	<i>MCTT</i> -Prozessmodell für die Notfallaufnahme	107
5.8	Entscheidungsknoten zur expliziten Entscheidungsmodellierung in <i>CTT</i> -Aufgabenmodellen	109
5.9	<i>MCTT</i> -Metamodellerweiterung für <i>DataChoice</i>	111
5.10	Das <i>MCTT</i> -Metamodel mit Datenmodellintegration	112
5.11	Das <i>MCTT</i> -Modell mit Datenintegration	113
6.1	Klassen aus dem <i>MCTT</i> -Metamodel inkl. der Hilfsoperationen	117
6.2	<i>MCTT</i> -Modelle zur Runtime dargestellt im <i>MCTT</i> -Plugin	120
6.3	Der <i>TTMS</i> -Editor	123
6.4	Die Nutzerschnittstelle des <i>TTMS WfMS</i> bereitgestellt über einen Browser	124
6.5	Instantiationtime-Operatoren bei <i>TTMS</i>	127
6.6	Eine Werkzeugkette im Zusammenhang mit Verwendung von <i>TTMS</i>	129

Listingsverzeichnis

3.1	OCL-Invariante und Vor- und Nachbedingungen für das Bankbeispiel	33
3.2	OCL Term zur Berechnung aller Quadratzahlen zwischen 1 und 100	33
3.3	OCL-Vor- und Nachbedingungen für Transitionen im Zustandsdiagramm der Klasse <i>Activity</i>	39
3.4	OCL-Vor- und Nachbedingungen für die Operation <i>finish()</i> der Klasse <i>Iteration</i>	40
3.5	OCL-Operationen <i>getSuccObjects()</i> und <i>getFlowObjects()</i> zur Berechnung der transitiven Hülle	41
3.6	Die OCL Invarianten zur Definition der <i>Parallel</i> -Beziehung und expliziten Entscheidungsmodellierung	42
3.7	Definition der Sequenz (WCP1) und Exclusive Choice (WCP4)	44
3.8	Definition vom Deferred Choice, Interleaved Parallel Routing und Milestone Pattern	47
3.9	OCL-Invarianten zur Soundness-Überprüfung zur Designtime	50
3.10	OCL-Invarianten zur Datenintegritätsprüfung im Zusammenhang mit Workflowausführung	56
3.11	OCL-Invariante zur Allokation von Personen für die Aktivitätsausführung	58
4.1	<i>start()</i> -Operation der Klasse <i>Activity</i> umgesetzt mit ASSL	71
4.2	Allokationsprozedur umgesetzt mit ASSL	72
4.3	Ausschnitt einer ASSL-Instanzierungsprozedur generiert vom Designtime-Plugin	75
4.4	Konsolenausgabe bei Peek-Aufruf der <i>finish()</i> Operation bei <i>AdjustMedication</i>	83
4.5	Konsolenausgabe für OCL-Mininganfrage	87
5.1	OCL-Invarianten zur Festlegung der strukturellen Eigenschaften von Aufgabenbäumen	105
6.1	SOIL-Umsetzung der <i>start()</i> -Operation der Klasse <i>LeafTask</i>	118

Literaturverzeichnis

- [Aal11] AALST, Wil M. P. d.: *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011
- [AAM⁺11] AALST, Wil M. P. d. ; ADRIANSYAH, Arya ; MEDEIROS, Ana Karla A. ; ARCIERI, Franco ; BAIER, Thomas ; BLICKLE, Tobias ; BOSE, R. P. Jagadeesh C. ; BRAND, Peter van d. ; BRANDTJEN, Ronald ; BUIJS, Joos C. A. M. ; BURATTIN, Andrea ; CARMONA, Josep ; CASTELLANOS, Malú ; CLAES, Jan ; COOK, Jonathan ; COSTANTINI, Nicola ; CURBERA, Francisco ; DAMIANI, Ernesto ; LEONI, Massimiliano de ; DELIAS, Pavlos ; DONGEN, Boudewijn F. ; DUMAS, Marlon ; DUSTDAR, Schahram ; FAHLAND, Dirk ; FERREIRA, Diogo R. ; GAALOUL, Walid ; GEFFEN, Frank van ; GOEL, Sukriti ; GÜNTHER, Christian W. ; GUZZO, Antonella ; HARMON, Paul ; HOFSTEDE, Arthur H. M. ; HOOGLAND, John ; INGVALDSEN, Jon E. ; KATO, Koki ; KUHN, Rudolf ; KUMAR, Akhil ; ROSA, Marcello L. ; MAGGI, Fabrizio M. ; MALERBA, Donato ; MANS, R. S. ; MANUEL, Alberto ; MCCREESH, Martin ; MELLO, Paola ; MENDLING, Jan ; MONTALI, Marco ; NEZHAD, Hamid R. M. ; MUEHLEN, Michael zur ; MUÑOZ-GAMA, Jorge ; PONTIERI, Luigi ; RIBEIRO, Joel ; ROZINAT, Anne ; PÉREZ, Hugo S. ; PÉREZ, Ricardo S. ; SEPÚLVEDA, Marcos ; SINUR, Jim ; SOFFER, Pnina ; SONG, Minseok ; SPERDUTI, Alessandro ; STILO, Giovanni ; STOEL, Casper ; SWENSON, Keith D. ; TALAMO, Maurizio ; TAN, Wei ; TURNER, Chris ; VANTHIENEN, Jan ; VARVARESOS, George ; VERBEEK, Eric ; VERDONK, Marc ; VIGO, Roberto ; WANG, Jianmin ; WEBER, Barbara ; WEIDLICH, Matthias ; WEIJTERS, Ton ; WEN, Lijie ; WESTERGAARD, Michael ; WYNN, Moe T.: Process Mining Manifesto. In: DANIEL, Florian (Hrsg.) ; BARKAOUI, Kamel (Hrsg.) ; DUSTDAR, Schahram (Hrsg.): *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I* Bd. 99, Springer, 2011 (Lecture Notes in Business Information Processing). – ISBN 978-3-642-28107-5, S. 169–194
- [ADG⁺07] AALST, Wil M. P. d. ; DONGEN, Boudewijn F. ; GÜNTHER, Christian W. ; MANS, R. S. ; MEDEIROS, Ana Karla A. ; ROZINAT, Anne ; RUBIN, Vladimir ; SONG, Minseok ; VERBEEK, H. M. W. (. ; WEIJTERS, A. J. M. M.: ProM 4.0: Comprehensive Support for *eal* Process Analysis. In: KLEIJN, Jetty (Hrsg.) ; YAKOVLEV, Alexandre (Hrsg.): *Petri Nets and Other Models of Concurrency - ICATPN 2007, 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings* Bd. 4546, Springer, 2007 (Lecture Notes in Computer Science), S. 484–494
- [ADG⁺09] AALST, Wil M. P. d. ; DONGEN, Boudewijn F. ; GÜNTHER, Christian W. ; ROZINAT, Anne ; VERBEEK, Eric ; WEIJTERS, Ton: ProM: The Process Mining Toolkit. In: MEDEIROS, Ana Karla A. (Hrsg.) ; WEBER, Barbara (Hrsg.): *Proceedings of the Business Process Management*

- Demonstration Track (BPMDeMos 2009), Ulm, Germany, September 8, 2009* Bd. 489, CEUR-WS.org, 2009 (CEUR Workshop Proceedings)
- [AG07] AALST, Wil M. P. d. ; GÜNTHER, Christian W.: Finding Structure in Unstructured Processes: The Case for Process Mining. In: BASTEN, Twan (Hrsg.) ; JUHÁS, Gabriel (Hrsg.) ; SHUKLA, Sandeep K. (Hrsg.): *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007), 10-13 July 2007, Bratislava, Slovak Republic*, IEEE Computer Society, 2007, S. 3–12
- [AH05] AALST, W. M. P. d. ; HOFSTEDE, A. H. M.: YAWL: Yet Another workflow language. In: *Information Systems* 30 (2005), Nr. 4, S. 245–275
- [AHKB03] AALST, W.M.P. van d. ; HOFSTEDE, A.H.M. ter ; KIEPUSZEWSKI, B. ; BARROS, A.P.: Workflow Patterns. In: *Distributed and Parallel Databases* 14 (2003), S. 5–51
- [AK01] AALST, Wil M. P. d. ; KUMAR, Akhil: A reference model for team-enabled workflow management systems. In: *Data Knowl. Eng.* 38 (2001), Nr. 3, S. 335–363
- [Ann03] ANNETT, John: Hierarchical Task Analysis. In: *[DS03]* (2003), S. 5–48
- [APS09] AALST, W. van d. ; PESIC, M. ; SCHONENBERG, H.: Declarative workflows: Balancing between flexibility and support. In: *Computer Science - Research and Development* 23 (2009), Nr. 2, S. 99–113
- [AS11] AALST, Wil Van D. ; STAHL, Christian: *Modeling Business Processes: A Petri Net-Oriented Approach*. MIT Press, 2011
- [Bae04] BAETEN, J. C. M.: A brief history of process algebra / *Theor. Comput. Sci.* 2004. – Forschungsbericht
- [Bal09] BALZERT, Helmut: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. Heidelberg : Spektrum, 2009
- [Bas09] BASTIDE, Rémi: An Integration of Task and Use-Case Meta-models. In: JACKO, Julie A. (Hrsg.): *Human-Computer Interaction. New Trends, 13th International Conference, HCI International 2009, San Diego, CA, USA, July 19-24, 2009, Proceedings, Part I* Bd. 5610, Springer, 2009 (Lecture Notes in Computer Science), S. 579–586
- [BB89] *Kapitel* Introduction to the ISO Specification Language LOTOS. In: BOLOGNESI, T. ; BRINKSMA, E.: *The Formal Description Technique LOTOS*. Elsevier Science Publishers B. V., (North-Holland), 1989
- [BB06] BASTIDE, Rémi ; BASNYAT, Sandra: Error Patterns: Systematic Investigation of Deviations in Task Models. In: CONINX, Karin (Hrsg.) ; LUYTEN, Kris (Hrsg.) ; SCHNEIDER, Kevin A. (Hrsg.): *Task Models and Diagrams for Users Interface Design, 5th International Workshop, TAMODIA 2006, Hasselt, Belgium, October 23-24, 2006. Revised Papers* Bd. 4385, Springer, 2006 (Lecture Notes in Computer Science), S. 109–121
- [BBC⁺10] BARZDINS, Janis ; BARZDINS, Guntis ; CERANS, Kalis ; LIEPINS, Renars ; SPROGIS, Arturs: UML Style Graphical Notation and Editor for OWL 2. In: FORBRIG, Peter (Hrsg.) ; GÜNTHER, Horst (Hrsg.): *Perspectives in Business Informatics Research (BIR2010)* Bd. 64, Springer, 2010 (LNBIP), S. 130–145

- [BCC07] BRAMBILLA, Marco ; CABOT, Jordi ; COMAI, Sara: Automatic Generation of Workflow-Extended Domain Models. In: ENGELS, Gregor (Hrsg.) ; OPDYKE, Bill (Hrsg.) ; SCHMIDT, Douglas C. (Hrsg.) ; WEIL, Frank (Hrsg.): *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings* Bd. 4735, Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978-3-540-75208-0, S. 375–389
- [BD07] BRITAIN, Jason ; DARWIN, Ian F.: *Tomcat: The Definitive Guide*. O'Reilly, 2007
- [BDFR07] BRÜNING, Jens ; DITTMAR, Anke ; FORBRIG, Peter ; REICHARD, Daniel: Getting SW Engineers on Board: Task Modelling with Activity Diagrams. In: GULLIKSEN, Jan (Hrsg.) ; HARNING, Morten B. (Hrsg.) ; PALANQUE, Philippe A. (Hrsg.) ; VEER, Gerrit C. d. (Hrsg.) ; WESSON, Janet (Hrsg.): *Engineering Interactive Systems (EIS2007)* Bd. 4940, Springer, 2007 (LNCS)
- [Bec00] BECK, Kent: *Extreme Programming. Das Manifest*. 2. Auflage. Addison-Wesley, 2000
- [BF08a] BRÜNING, Jens ; FORBRIG, Peter: Behaviour of flow operators connected with object flows in workflow specifications. In: WRYCZA, Stanislaw (Hrsg.): *Perspectives of Business Informatics Research (BIR2008)*, University of Gdansk, 2008
- [BF08b] BRÜNING, Jens ; FORBRIG, Peter: Methoden zur Adaptiven Anpassung von EPKs an Individuelle Anforderungen vor der Abarbeitung. In: LOOS, Peter (Hrsg.) ; NÜTTGENS, Markus (Hrsg.) ; TUROWSKI, Klaus (Hrsg.) ; WERTH, Dirk (Hrsg.): *Proceedings of the Workshops collocated with the MobIS2008 Conference: Including EPK2008, KobAS2008 and ModKollGP2008, Saarbrücken, Germany, November 27-28, 2008* Bd. 420, CEUR-WS.org, 2008 (CEUR Workshop Proceedings), S. 31–43
- [BF11] BRÜNING, Jens ; FORBRIG, Peter: TTMS: A Task Tree Based Workflow Management System. In: HALPIN, Terry A. (Hrsg.) ; NURCAN, Selmin (Hrsg.) ; KROGSTIE, John (Hrsg.) ; SOFFER, Pnina (Hrsg.) ; PROPER, Erik (Hrsg.) ; SCHMIDT, Rainer (Hrsg.) ; BIDER, Iliia (Hrsg.): *Enterprise, Business-Process and Information Systems Modeling - 12th International Conference, BPMDS 2011, and 16th International Conference, EMMSAD 2011, held at CAiSE 2011, London, UK, June 20-21, 2011. Proceedings* Bd. 81, Springer, 2011 (Lecture Notes in Business Information Processing), S. 186–200
- [BFSZ12] BRÜNING, Jens ; FORBRIG, Peter ; SEIB, Enrico ; ZAKI, Michael: On the Suitability of Activity Diagrams and ConcurTaskTrees for Complex Event Modeling. In: ASEEVA, Natalia (Hrsg.) ; BABKIN, Eduard (Hrsg.) ; KOZYREV, Oleg (Hrsg.): *Perspectives in Business Informatics Research (BIR2012)* Bd. 128, Springer, 2012 (LNBIP), S. 54–69
- [BG11] BÜTTNER, Fabian ; GOGOLLA, Martin: Modular Embedding of the Object Constraint Language into a Programming Language. In: SILVA SIMÃO, Adenildo da (Hrsg.) ; MORGAN, Carroll (Hrsg.): *Formal Methods, Foundations and Applications - 14th Brazilian Symposium, SBMF 2011, São Paulo, Brazil, September 26-30, 2011, Revised Selected Papers* Bd. 7021, Springer, 2011 (Lecture Notes in Computer Science). – ISBN 978-3-642-25031-6, S. 124–139
- [BGF10] BRÜNING, Jens ; GOGOLLA, Martin ; FORBRIG, Peter: Modeling and Formally Checking Workflow Properties Using UML and OCL. In: FORBRIG, Peter (Hrsg.) ; GÜNTHER, Horst (Hrsg.): *Perspectives in Business Informatics Research (BIR2010)* Bd. 64, Springer, 2010 (LNBIP), S. 130–145

- [BGHM11] BOMSDORF, Birgit ; GRAU, Stefan ; HUDASCH, Martin ; MILDE, Jan-Torsten: Configurable Executable Task Models Supporting the Transition from Design Time to Runtime. In: JACKO, Julie (Hrsg.): *Human-Computer Interaction. Design and Development Approaches* Bd. 6761. Springer, 2011, S. 155–164
- [BHW11] BRÜNING, Jens ; HAMANN, Lars ; WOLFF, Andreas: Extending ASSL: Making UML Metamodel-based Workflows executable. In: CABOT, Jordi (Hrsg.) ; CLARISÓ, Robert (Hrsg.) ; GOGOLLA, Martin (Hrsg.) ; WOLFF, Burkhart (Hrsg.): *OCL and Textual Modelling (OCL2011)* Bd. 44, ECEASST, 2011
- [BM04] BIDOIT, Michel ; MOSSES, Peter D.: *CASL User Guide*. Springer, 2004
- [Bom07] BOMSDORF, Birgit: The WebTaskModel Approach to Web Process Modelling. In: WINCKLER, Marco (Hrsg.) ; JOHNSON, Hilary (Hrsg.) ; PALANQUE, Philippe (Hrsg.): *Task Models and Diagrams for User Interface Design* Bd. 4849. Springer, 2007. – ISBN 978–3–540–77221–7, S. 240–253
- [BPE07] *Web Services Business Process Execution Language Version 2.0*. OASIS Standard, 2007. – <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>
- [BPM09] *Business Process Model and Notation (BPMN) Version 1.2*. <http://www.omg.org/spec/BPMN/1.2/PDF>, Januar 2009. – öffentliche Spezifikation
- [BPM11] *Business Process Model and Notation (BPMN) Version 2.0*. <http://www.omg.org/spec/BPMN/2.0/PDF>, Januar 2011. – öffentliche Spezifikation
- [Büt11] BÜTTNER, Fabian: *Reusing OCL in the definition of imperative languages*, Universität Bremen, Diss., 2011
- [Cam89] CAMERON, John R.: *Jsp and Jsd: The Jackson Approach to Software Development*. second. IEEE Computer Society Press, U.S., 1989
- [Che76] CHEN, Peter Pin-Shan: The entity-relationship model—toward a unified view of data. In: *ACM Trans. Database Syst.* 1 (1976), Nr. 1, S. 9–36. <http://dx.doi.org/http://doi.acm.org/10.1145/320434.320440>. – DOI <http://doi.acm.org/10.1145/320434.320440>. – ISSN 0362–5915
- [CMN83] CARD, Thomas K. ; MORAN, Thomas P. ; NEWELL, Allen: *The psychology of human-computer interaction*. Lawrence Erlbaum Associates Inc., 1983
- [DA05] DONGEN, Boudewijn F. ; AALST, Wil M. P. d.: A Meta Model for Process Mining Data. In: MISSIKOFF, Michele (Hrsg.) ; NICOLA, Antonio D. (Hrsg.): *EMOI - INTEROP'05, Enterprise Modelling and Ontologies for Interoperability, Proceedings of the Open Interop Workshop on Enterprise Modelling and Ontologies for Interoperability, Co-located with CAiSE'05 Conference, Porto (Portugal), 13th-14th June 2005* Bd. 160, CEUR-WS.org, 2005 (CEUR Workshop Proceedings)
- [Dat07] DATABASE SYSTEMS GROUP, BREMEN UNIVERSITY (Hrsg.): *USE - A UML based Specification Environment (manual)*. Database Systems Group, Bremen University, 2007. <http://www.db.informatik.uni-bremen.de/projects/USE/use-documentation.pdf>

- [DeM79] DEMARCO, Tom: *Structured analysis and system specification*. Upper Saddle River, NJ, USA : Yourdon Press, 1979. – 409–424 S. – ISBN 0–917072–14–6
- [DF09] DITTMAR, Anke ; FORBRIG, Peter: Task-based design revisited. In: *EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. New York, NY, USA : ACM, 2009. – ISBN 978–1–60558–600–7, S. 111–116
- [DFHS05] DITTMAR, Anke ; FORBRIG, Peter ; HEFTBERGER, Simone ; STARY, Chris: Support for Task Modeling – A "Constructive" Exploration. In: BASTIDE, Rémi (Hrsg.) ; PALANQUE, Philippe (Hrsg.) ; ROTH, Jörg (Hrsg.): *Engineering Human Computer Interaction and Interactive Systems* Bd. 3425. Springer Berlin / Heidelberg, 2005, S. 137–137
- [DG11] DIJKMAN, Remco M. ; GORP, Pieter V.: BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules. In: MENDLING, Jan (Hrsg.) ; WEIDLICH, Matthias (Hrsg.) ; WESKE, Mathias (Hrsg.): *Business Process Modeling Notation - Second International Workshop, BPMN 2010, Potsdam, Germany, October 13-14, 2010. Proceedings* Bd. 67, Springer, 2011 (Lecture Notes in Business Information Processing), S. 16–30
- [Dij68] DIJKSTRA, Edsger W.: Go To Statement Considered Harmful. In: *Communications of the ACM* 11, No. 3 (1968), S. 147–148
- [DM09] DECKER, Gero ; MENDLING, Jan: Process instantiation. In: *Data Knowl. Eng.* 68 (2009), Nr. 9, S. 777–792
- [DMV⁺05] DONGEN, B. F. ; MEDEIROS, A. K. A. ; VERBEEK, H. M. W. ; WEIJTERS, A. J. M. M. ; AALST, W. M. P. d.: The ProM Framework: A New Era in Process Mining Tool Support. Version: 2005. http://dx.doi.org/10.1007/11494744_25. In: CIARDO, Gianfranco (Hrsg.) ; DARONDEAU, Philippe (Hrsg.): *Applications and Theory of Petri Nets 2005* Bd. 3536. Berlin, Heidelberg : Springer Berlin / Heidelberg, 2005. – DOI 10.1007/11494744_25. – ISBN 978–3–540–26301–2, Kapitel 25, 444–454
- [DMW09] DONGEN, Boudewijn F. ; MEDEIROS, Ana Karla A. ; WEN, L.: Process Mining: Overview and Outlook of Petri Net Discovery Algorithms. In: JENSEN, Kurt (Hrsg.) ; AALST, Wil M. P. d. (Hrsg.): *Transactions on Petri Nets and Other Models of Concurrency II, Special Issue on Concurrency in Process-Aware Information Systems* Bd. 2, Springer, 2009 (Lecture Notes in Computer Science). – ISBN 978–3–642–00898–6, S. 225–242
- [DR09] DADAM, Peter ; REICHERT, Manfred: The ADEPT project: A decade of research and development for robust and flexible process support - challenges and achievements. In: *Computer Science - Research and Development* 22 (2009), S. 81–97
- [DRRM11] DADAM, Peter ; REICHERT, Manfred ; RINDERLE-MA, Stefanie: Prozessmanagementsysteme - Nur ein wenig Flexibilität wird nicht reichen. In: *Informatik Spektrum* 34 (2011), Nr. 4, S. 364–376
- [DS03] DIAPER, Dan (Hrsg.) ; STANTON, Neville (Hrsg.): *The Handbook of Task Analysis for Human-Computer Interaction*. Mahwah : Lawrence Erlbaum Assoc. Inc., 2003
- [FCJ96] FOWLER, Martin ; CUNNINGHAM, Ward ; JOHNSON, Ralph: *Analysis Patterns: Reusable Object Models*. Addison-Wesley Longman, 1996

- [FL11] FIGL, Katharina ; LAUE, Ralf: Cognitive Complexity in Business Process Modeling. In: MOURATIDIS, Haralambos (Hrsg.) ; ROLLAND, Colette (Hrsg.): *Advanced Information Systems Engineering (CAiSE2011)* Bd. 6741, Springer, 2011 (LNCS)
- [For07] FORBRIG, Peter: *Objektorientierte Softwareentwicklung mit UML*. Carl Hanser Verlag, 2007
- [FR07] FORBRIG, Peter ; REICHART, Daniel: Spezifikation von "Multiple User Interfaces" mit Dialoggraphen. In: KOSCHKE, Rainer (Hrsg.) ; HERZOG, Otthein (Hrsg.) ; RÖDIGER, Karl-Heinz (Hrsg.) ; RONTHALER, Marc (Hrsg.): *INFORMATIK 2007: Informatik trifft Logistik. Band 1. Beiträge der 37. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 24.-27. September 2007 in Bremen* Bd. 109, GI, 2007 (LNI), S. 449–453
- [Gad07] GADATSCH, Andreas: *Grundkurs Geschäftsprozess-Management: Methoden und Werkzeuge für die IT-Praxis: Eine Einführung für Studenten und Praktiker*. Vieweg+Teubner Verlag, 2007
- [GBR05] GOGOLLA, Martin ; BOHLING, Jörn ; RICHTERS, Mark: Validating UML and OCL models in USE by automatic snapshot generation. In: *Journal on Software and System Modeling* 4 (2005), S. 2005
- [GBR07] GOGOLLA, Martin ; BÜTTNER, Fabian ; RICHTERS, Mark: USE: A UML-based specification environment for validating UML and OCL. In: *Science of Computer Programming* 69 (2007), Nr. 1-3, S. 27 – 34
- [GHJV02] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns - Elements of Reusable Object-Oriented Software*. 24th. Addison-Wesley, 2002
- [GLKK08] GRUHN, Volker ; LAUE, Ralf ; KERN, Heiko ; KÜHNE, Stefan: EPK-Validierung zur Modellierungszeit in der bflow* Toolbox. In: LOOS, Peter (Hrsg.) ; NÜTTGENS, Markus (Hrsg.) ; TUROWSKI, Klaus (Hrsg.) ; WERTH, Dirk (Hrsg.): *Modellierung betrieblicher Informationssysteme - Modellierung zwischen SOA und Compliance Management - 27.-28. November 2008 Saarbrücken, Germany* Bd. 141, GI, 2008 (LNI), S. 181–194. – MobIS
- [Gog04] GOGOLLA, Martin: Benefits and Problems of Formal Methods. In: LLAMOSAS, Albert (Hrsg.) ; STROHMEIER, Alfred (Hrsg.): *Reliable Software Technologies - Ada-Europe 2004* Bd. 3063. Springer Berlin / Heidelberg, 2004, S. 1–15
- [Gro09] GRONBACK, Richard C.: *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Longman, 2009
- [HAAR09] HOFSTEDTE, Arthur H. M. ; AALST, Wil M. P. d. ; ADAMS, Michael ; RUSSELL, Nick: *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2009
- [Har84] HAREL, David: Statecharts: A Visual Approach to Complex Systems / Department of Applied Mathematics, The Weizmann Institute of Science. 1984. – Forschungsbericht
- [Hoa69] HOARE, C. A. R.: An axiomatic basis for computer programming. In: *Communications of the ACM* Bd. 12. ACM, 1969, S. 576–583
- [Hol98] HOLLINGSWORTH, David: *The Workflow Reference Model: TC00-1003, Issue 1.1*. Workflow Management Coalition, Januar 1998. – <http://www.wfmc.org/Download-document/TC00-1003-The-Workflow-Reference-Model.html>
- [IYO] *IYOPRO - Improve your process*. <http://www.iyopro.de>, 2012-04-06,

- [JABK08] JOUAULT, Frédéric ; ALLILAIRE, Freddy ; BÉZIVIN, Jean ; KURTEV, Ivan: ATL: A model transformation tool. In: *Sci. Comput. Program.* 72 (2008), Nr. 1-2, S. 31–39
- [Jac82] JACKSON, Michael: A System Development Method. In: NÉELE, D. (Hrsg.): *Tools and notions for program construction: An advanced course*, Cambridge University Press, 1982, S. 1–25
- [Jac03] JACKSON, Daniel: Alloy: A Logical Modelling Language. In: BERT, Didier (Hrsg.) ; BOWEN, Jonathan (Hrsg.) ; KING, Steve (Hrsg.) ; WALDÉN, Marina (Hrsg.): *ZB 2003: Formal Specification and Development in Z and B* Bd. 2651. Springer Berlin / Heidelberg, 2003, S. 629–629
- [JJ91] JOHNSON, Hilary ; JOHNSON, Peter: Task knowledge structures: Psychological basis and integration into system design. In: *Acta Psychologica* 78 (1991), Nr. 1-3, 3 - 26. [http://dx.doi.org/DOI:10.1016/0001-6918\(91\)90003-I](http://dx.doi.org/DOI:10.1016/0001-6918(91)90003-I). – DOI DOI: 10.1016/0001-6918(91)90003-I. – ISSN 0001-6918
- [JKBL10] JABRI, Sana ; KOURSI, ElMiloudi ; BOURDEAUD’HUY, Thomas ; LEMAIRE, Etienne: European railway traffic management system validation using UML/Petri nets modelling strategy. In: *European Transport Research Review* 2 (2010), 113-128. <http://dx.doi.org/10.1007/s12544-010-0030-5>. – DOI 10.1007/s12544-010-0030-5. – ISSN 1867-0717
- [JTF09] JØRGENSEN, Jens B. ; TJELL, Simon ; FERNANDES, João M.: Formal requirements modelling with executable use cases and coloured Petri nets. In: *Innovations Syst Softw Eng* 5 (2009), S. 13–25
- [KDF10] KÜHN, Robert ; DITTMAR, Anke ; FORBRIG, Peter: Alternative Representations of Workflow Control-Flow Patterns Using HOPS. In: FORBRIG, Peter (Hrsg.) ; GÜNTHER, Horst (Hrsg.) ; AALST, Wil (Hrsg.) ; MYLOPOULOS, John (Hrsg.) ; ROSEMANN, Michael (Hrsg.) ; SHAW, Michael J. (Hrsg.) ; SZYPERSKI, Clemens (Hrsg.): *Perspectives in Business Informatics Research* Bd. 64. Springer, 2010, S. 115–129
- [KHB00] KIEPUSZEWSKI, Bartek ; HOFSTEDE, Arthur H. M. ; BUSSLER, Christoph: On Structured Workflow Modelling. In: WANGLER, Benkt (Hrsg.) ; BERGMAN, Lars (Hrsg.): *Advanced Information Systems Engineering, 12th International Conference CAiSE 2000, Stockholm, Sweden, June 5-9, 2000, Proceedings* Bd. 1789, Springer, 2000 (Lecture Notes in Computer Science), S. 431–445
- [KHGB] KUHLMANN, Mirco ; HAMANN, Lars ; GOGOLLA, Martin ; BÜTTNER, Fabian: A benchmark for OCL engine accuracy, determinateness, and efficiency. In: *Software and Systems Modeling* <http://dx.doi.org/10.1007/s10270-010-0174-8>. – ISSN 1619-1366. – 10.1007/s10270-010-0174-8
- [Kie03] KIERAS, David: GOMS Models for Task Analysis. In: *[DS03]* (2003), S. 83–116
- [Kle08] KLEPPE, Anneke: *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Longman, 2008
- [KNS92] KELLER, Gerhard ; NÜTTGENS, Markus ; SCHEER, August-Wilhelm: Semantische Prozeßmodellierung auf der Grundlage „Ereignisgesteuerter Prozeßketten (EPK)“. In: SCHEER, August-Wilhelm (Hrsg.): *Veröffentlichungen des Instituts für Wirtschaftsinformatik, Universität des Saarlandes*. Saarbrücken, 1992 (89)

- [LG08] LAUE, Ralf ; GRUHN, Volker: Good and Bad Excuses for Unstructured Business Process Models. In: HVATUM, Lise B. (Hrsg.) ; SCHÜMMER, Till (Hrsg.): *Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP '2007), Irsee, Germany, July 4-8, 2007*, UVK - Universitätsverlag Konstanz, 2008. – ISBN 978-3-87940-819-1, S. 279–290
- [LM10] LAUE, Ralf ; MENDLING, Jan: Structuredness and its significance for correctness of process models. In: *Inf. Syst. E-Business Management* 8 (2010), Nr. 3, S. 287–307
- [Luy04] LUYTEN, Kris: *Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development*, Limburgs Universitair Centrum, transnational University Limburg: School of Information Technology, Diss., 2004
- [LVM⁺05] LIMBOURG, Quentin ; VANDERDONCKT, Jean ; MICHOTTE, Benjamin ; BOUILLON, Laurent ; LÓPEZ-JAQUERO, Víctor: USIXML: A Language Supporting Multi-path Development of User Interfaces. In: BASTIDE, Rémi (Hrsg.) ; PALANQUE, Philippe A. (Hrsg.) ; ROTH, Jörg (Hrsg.) ; BASTIDE, Rémi (Hrsg.) ; PALANQUE, Philippe A. (Hrsg.) ; ROTH, Jörg (Hrsg.): *Engineering Human Computer Interaction and Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004, Hamburg, Germany, July 11-13, 2004, Revised Selected Papers* Bd. 3425, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3-540-26097-8, S. 200–220
- [LW94] LISKOV, Barbara ; WING, Jeannette M.: A Behavioral Notion of Subtyping. In: *ACM Trans. Program. Lang. Syst.* 16 (1994), Nr. 6, S. 1811–1841
- [Mal03] MALIK, Ayesha: Design XML schemas using UML / IBM. 2003. – Forschungsbericht
- [May99] MAYHEW, Deborah: *The usability engineering lifecycle*. Morgan Kaufmann, 1999
- [Men08] MENDLING, Jan: Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness, Springer, 2008 (Lecture Notes in Business Information Processing)
- [Mey92] MEYER, Bertrand: Applying "Design by Contract". In: *IEEE Computer* 25 (1992), Nr. 10, S. 40–51
- [MFJ05] MULLER, Pierre-Alain ; FLEUREY, Franck ; JÉZÉQUEL, Jean-Marc: Weaving Executability into Object-Oriented Meta-languages. In: BRIAND, Lionel C. (Hrsg.) ; WILLIAMS, Clay (Hrsg.): *MoDELS* Bd. 3713, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3-540-29010-9, S. 264–278
- [MMS⁺03] MAEDCHE, Alexander ; MOTIK, Boris ; STOJANOVIC, Ljiljana ; STUDER, Rudi ; VOLZ, Raphael: Ontologies for Enterprise Knowledge Management. In: *IEEE Intelligent Systems* 18 (2003), Nr. 2, S. 26–33
- [MNN05] MENDLING, Jan ; NEUMANN, Gustaf ; NÜTTGENS, Markus: Yet Another Event-driven Process Chain - Modelling Workflow Patterns with yEPCs. In: *Enterprise Modelling and Information Systems Architectures* 1 (2005), Nr. 1, S. 03–13
- [MOF10] Object Management Group (OMG): *OMG MOF Facility Object Lifecycle (MOFFOL)*, v2.0. <http://www.omg.org/spec/MOFFOL/2.0/PDF>. Version: March 2010

- [MPS02] MORI, G. ; PATERNO, F. ; SANTORO, C.: CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. In: *IEEE Transactions on Software Engineering* volume 28 Issue 8 (2002), S. 797–813
- [MPV11] MEIXNER, Gerrit ; PATERNO, Fabio ; VANDERDONCKT, Jean: Past, Present, and Future of Model-Based User Interface Development. In: *i-com* 10 (2011), Nr. 3, S. 2–11
- [MR06] MUTSCHLER, B. ; REICHERT, M.U.: Aktuelles Schlagwort: Business Process Intelligence. In: *EMISA Forum* 26 (2006), January, Nr. 1, 27–31. <http://doc.utwente.nl/66218/>
- [MRR10] MENDLING, Jan ; RECKER, Jan ; REIJERS, Hajo A.: On the Usage of Labels and Icons in Business Process Modeling. In: *IJISMD* 1 (2010), Nr. 2, S. 40–58
- [Net06] *Enterprise Service Oriented Architecture Project Home*. 2006. – <http://soa.netbeans.org/soa/>
- [OCL10] *Object Constraint Language, Version 2.2*. <http://www.omg.org/spec/OCL/2.2/>, 2010-10-01, 2010. – öffentliche Spezifikation
- [ODHA06] OUYANG, Chun ; DUMAS, Marlon ; HOFSTEDE, Arthur H. M. ; AALST, Wil M. P. d.: From BPMN Process Models to BPEL Web Services. In: *ICWS*, IEEE Computer Society, 2006, S. 285–292
- [OMG10] OMG: *Concrete Syntax for UML Action Language (Action Language for Foundational UML - ALF)*. <http://www.omg.org/spec/ALF/1.0/Beta1,ptc/2010-10-05>, 2010
- [OR99] ORMEROD, Thomas C. ; RIDGWAY, James: Developing task design guides through cognitive studies of expertise. In: *European Conference on Cognitive Science (ECCS1999)*, 1999. – <http://www.psych.lancs.ac.uk/people/uploads/TomOrmerod20041013T092208.pdf>
- [OWS+03] OESTEREICH, Bernd ; WEISS, Christian ; SCHRÖDER, Claudia ; WEILKIENS, Tim ; LENHARD, Alexander: *Objektorientierte Geschäftsprozessmodellierung mit der UML*. dpunkt Verlag, 2003
- [Par07] PARR, Terence: *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Pragmatic Programmers, 2007
- [Pat99] PATERNO, Fabio: *Model-Based Design and Evaluation of Interactive Applications*. Springer, 1999
- [Peš08] PEŠIĆ, Maja: *Constraint-Based Workflow Management Systems: Shifting Control to Users*, Technische Universiteit Eindhoven, Proefschrift, 2008. <http://www.win.tue.nl/~mpesic/papers/Proefschrift.pdf>
- [Pet62] PETRI, C.A.: *Kommunikation mit Automaten*, Institut für instrumentelle Mathematik der Universität Bonn, Diss., 1962
- [Pic07] PICHLER, Roman: *Scrum - Agiles Projektmanagement erfolgreich einsetzen*. dpunkt.verlag, 2007
- [Por85] PORTER, Michael: *Competitive Advantage*. The Free Press, 1985

- [PSSA10] PESIC, M. ; SCHONENBERG, M. ; SIDOROVA, N. ; AALST, W. van d.: Constraint-Based Workflow Models: Change Made Easy. In: MEERSMAN, Robert (Hrsg.) ; TARI, Zahir (Hrsg.): *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS* Bd. 4803. Springer Berlin / Heidelberg, 2010, S. 77–94
- [PW05] PUHLMANN, Frank ; WESKE, Mathias: Using the π -Calculus for Formalizing Workflow Patterns. In: AALST, Wil M. P. d. (Hrsg.) ; BENATALLAH, Boualem (Hrsg.) ; CASATI, Fabio (Hrsg.) ; CURBERA, Francisco (Hrsg.): *Business Process Management* Bd. 3649, Springer, 2005, S. 153–168
- [PWZ⁺11] PICHLER, Paul ; WEBER, Barbara ; ZUGAL, Stefan ; PINGGERA, Jakob ; MENDLING, Jan ; REIJERS, Hajo A.: Imperative versus Declarative Process Modeling Languages: An Empirical Investigation. In: DANIEL, Florian (Hrsg.) ; BARKAOUI, Kamel (Hrsg.) ; DUSTDAR, Schahram (Hrsg.): *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I* Bd. 99, Springer, 2011 (Lecture Notes in Business Information Processing). – ISBN 978–3–642–28107–5, S. 383–394
- [QVT08] *Query View Transformation*. <http://www.omg.org/spec/QVT/1.0/>, 2010-08-11, 2008. – öffentliche Spezifikation
- [RA07] ROSEMANN, Michael ; AALST, Wil M. P. d.: A configurable reference modelling language. In: *Inf. Syst.* 32 (2007), Nr. 1, S. 1–23
- [RAHW06] RUSSELL, Nick ; AALST, Wil M. P. d. ; HOFSTEDE, Arthur H. M. ; WOHED, Petia: On the suitability of UML 2.0 activity diagrams for business process modelling. In: STUMPTNER, Markus (Hrsg.) ; HARTMANN, Sven (Hrsg.) ; KIYOKI, Yasushi (Hrsg.): *Conceptual Modelling 2006, Third Asia-Pacific Conference on Conceptual Modelling (APCCM 2006), Hobart, Tasmania, Australia, January 16-19 2006* Bd. 53, Australian Computer Society, 2006 (CRPIT), S. 95–104
- [Ree79] REENSKAUG, Trygve: The original MVC reports / University of Oslo. Version: 1979. http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf. 1979. – Forschungsbericht
- [Rei00] REICHERT, Manfred: *Dynamische Ablaufänderungen in Workflow-Management-Systemen*, Universität Ulm, Diss., 2000
- [RF08] REICHAERT, Daniel ; FORBRIG, Peter: Transactions in Task Models. In: FORBRIG, Peter (Hrsg.) ; PATERNÒ, Fabio (Hrsg.): *Engineering Interactive Systems* Bd. 5247. Springer, 2008, S. 299–304
- [RG00] RICHTERS, Mark ; GOGOLLA, Martin: Validating UML Models and OCL Constraints. In: EVANS, Andy (Hrsg.) ; KENT, Stuart (Hrsg.) ; SELIC, Bran (Hrsg.): *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings* Bd. 1939, Springer, 2000 (Lecture Notes in Computer Science), S. 265–277
- [RGDA07] ROSA, Marcello L. ; GOTTSCHALK, Florian ; DUMAS, Marlon ; AALST, Wil M. P. d.: Linking Domain Models and Process Models for Reference Model Configuration. In: HOFSTEDE, Arthur H. M. (Hrsg.) ; BENATALLAH, Boualem (Hrsg.) ; PAIK, Hye-Young (Hrsg.): *Business*

- Process Management Workshops, BPM 2007 International Workshops, BPI, BPD, CBP, ProHealth, RefMod, semantics4ws, Brisbane, Australia, September 24, 2007, Revised Selected Papers* Bd. 4928, Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–78237–7, S. 417–430
- [RHAM06] RUSSELL, Nick ; HOFSTEDE, Arthur H. M. ; AALST, Wil M. P. d. ; MULYAR, Nataliya: *Workflow ControlFlow Patterns: A Revised View*. BPM Center Report, 2006. – <http://www.workflowpatterns.com/documentation/documents/BPM-06-22.pdf>
- [RHEA04a] RUSSELL, N. ; HOFSTEDE, A.H.M. ter ; EDMOND, D. ; AALST, W.M.P. van d.: *Workflow Data Patterns / Queensland University of Technology, Brisbane*. 2004. – QUT Technical report, FIT-TR-2004-01
- [RHEA04b] RUSSELL, N. ; HOFSTEDE, A.H.M. ter ; EDMOND, D. ; AALST., W.M.P. van d.: *Workflow Resource Patterns / Eindhoven University of Technology*. 2004. – BETA Working Paper Series, WP 127
- [Ric01] RICHTERS, Mark: *A Precise Approach to Validating UML Models and OCL Constraints*. Logos Verlag, Berlin, BISS Monographs, No. 14, Universität Bremen, Diss., 2001
- [Ric09] RICH, Charles: *Building Task-Based User Interfaces with ANSI/CEA-2018*. In: *IEEE Computer* 42 (2009), Nr. 8, S. 20–27
- [Sch01] SCHEER, August-Wilhelm: *ARIS - Modellierungsmethoden, Metamodelle, Anwendungen*. Springer, 2001
- [Sch02] SCHEER, August-Wilhelm: *ARIS - Vom Geschäftsprozess zum Anwendungssystem*. Springer, 2002
- [SMR⁺07] SCHONENBERG, M. H. ; MANS, R. S. ; RUSSELL, N. C. ; MULYAR, N. A. ; AALST, W. M. P. d.: *Towards a Taxonomy of Process Flexibility (Extended Version)*. BPM Center Report. <http://www.wis.win.tue.nl/~wvdaalst/BPMcenter/reports/2007/BPM-07-11.pdf>. Version: 2007
- [SMR⁺08] SCHONENBERG, Helen ; MANS, Ronny ; RUSSELL, Nick ; MULYAR, Nataliya ; AALST, Wil M. P. d.: *Towards a Taxonomy of Process Flexibility*. In: BELLAHSENE, Zohra (Hrsg.) ; WOO, Carson (Hrsg.) ; HUNT, Ela (Hrsg.) ; FRANCH, Xavier (Hrsg.) ; COLETTA, Remi (Hrsg.): *CAiSE Forum* Bd. 344, CEUR-WS.org, 2008 (CEUR Workshop Proceedings), S. 81–84
- [ST05] SCHEER, August-Wilhelm ; THOMAS, Oliver: *Geschäftsprozessmodellierung mit der ereignis-gesteuerten Prozesskette*. In: *Das Wirtschaftsstudium* 34 (2005), Nr. 8-9, S. 1069–1078
- [Sta94] STARY, Christian: *Interaktive Systeme: Software-Entwicklung und Software-Ergonomie*. Vieweg, 1994
- [STA05] SCHEER, A.-W. ; THOMAS, O. ; ADAM, O.: *Process Modeling Using Event-driven Process Chains*. In: DUMAS, M. (Hrsg.) ; AALST, W. M. P. d. (Hrsg.) ; HOFSTEDE, A. H. M. (Hrsg.): *Process-aware Information Systems : Bridging People and Software through Process Technology*. Wiley, 2005, S. 119–145
- [TF07] THOMAS, Oliver ; FELLMANN, Michael: *Semantic EPC: Enhancing Process Modeling Using Ontology Languages*. In: HEPP, Martin (Hrsg.) ; HINKELMANN, Knut (Hrsg.) ;

- KARAGIANNIS, Dimitris (Hrsg.) ; KLEIN, Rüdiger (Hrsg.) ; STOJANOVIC, Nenad (Hrsg.): *Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management held in conjunction with the 3rd European Semantic Web Conference (ESWC 2007), Innsbruck, Austria, June 7, 2007* Bd. 251, CEUR-WS.org, 2007 (CEUR Workshop Proceedings)
- [Tou07] TOUSSAINT, Frederic: *Grafische Benutzungsunterstützung auf Befehlsebene für die Entwicklung massivparalleler Programme*, Universität Karlsruhe, Diss., 2007
- [Uhr03] UHR, Holger: Tombola: Simulation and user-specific presentation of executable task models. In: *In Proceedings of HCI International*, 2003
- [UML10] Object Management Group (OMG): *OMG Unified Modeling Language (OMG UML), Superstructure: Version 2.3*. <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/>. Version: Mai 2010
- [VLB96] VEER, Gerrit C. Van D. ; LENTING, Bert F. ; BERGEVOET, Bas A. J.: GTA: Groupware Task Analysis - Modeling Complexity. In: *Acta Psychologica* 91 (1996), S. 297–322
- [VVK09] VANHATALO, J. ; VÖLZER, H. ; KOEHLER, J.: The refined process structure tree. In: *Data & Knowledge Engineering* 68 (2009), S. 793–818
- [VWC02] VEER, Gerrit C. d. ; WELIE, Martijn van ; CHISALITA, Cristina: Introduction to Groupware Task Analysis. In: PRIBEANU, Costin (Hrsg.) ; VANDERDONCKT, Jean (Hrsg.): *Task Models and Diagrams for User Interface Design: Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design - TAMODIA 2002, 18-19 July 2002, Bucharest, Romania*, INFOREC Publishing House Bucharest, 2002. – ISBN 973–8360–01–3, S. 32–39
- [Wac08] WACHSMUTH, Guido: Modelling the Operational Semantics of Domain-Specific Modelling Languages. In: LÄMMEL, Ralf (Hrsg.) ; VISSER, Joost (Hrsg.) ; SARAIVA, João (Hrsg.): *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers* Bd. 5235, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–88642–6, S. 506–520
- [WAD⁺06] WOHEDE, Petia ; AALST, Wil M. P. d. ; DUMAS, Marlon ; HOFSTEDDE, Arthur H. M. ; RUSSELL, Nick: On the Suitability of BPMN for Business Process Modelling. In: DUSTDAR, Schahram (Hrsg.) ; FIADEIRO, José Luiz (Hrsg.) ; SHETH, Amit P. (Hrsg.): *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings* Bd. 4102, Springer, 2006 (Lecture Notes in Computer Science), S. 161–176
- [WADH03] WOHEDE, Petia ; AALST, Wil M. P. d. ; DUMAS, Marlon ; HOFSTEDDE, Arthur H. M.: Analysis of Web Services Composition Languages: The Case of BPEL4WS. In: SONG, Il-Yeol (Hrsg.) ; LIDDLE, Stephen W. (Hrsg.) ; LING, Tok W. (Hrsg.) ; SCHEUERMANN, Peter (Hrsg.): *Conceptual Modeling - ER 2003, 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October 13-16, 2003, Proceedings* Bd. 2813, Springer, 2003 (Lecture Notes in Computer Science). – ISBN 3–540–20299–4
- [Wes99] WESKE, Mathias: *Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects*, Mathematisch-Naturwissenschaftliche Fakultät der Westfälischen Wilhelms-Universität Münster, Habilitationsschrift, Dezember 1999

- [Wes07] WESKE, Mathias: *Business Process Management: Concepts, Languages, Architectures*. Springer Berlin Heidelberg, 2007
- [WFRS07] WURDEL, Maik ; FORBRIG, Peter ; RADHAKRISHNAN, T. ; SINNIG, Daniel: Patterns for Task- and Dialog-Modeling. Version: 2007. http://dx.doi.org/10.1007/978-3-540-73105-4_133. In: JACKO, Julie (Hrsg.): *Human-Computer Interaction. Interaction Design and Usability* Bd. 4550. Springer, 2007. – ISBN 978-3-540-73104-7, 1226-1235
- [Whi04] WHITE, Stephen A.: Process Modeling Notations and Workflow Patterns / IBM Corp. United States, Januar 2004. – Forschungsbericht
- [WK04] WARMER, Jos ; KLEPPE, Anneke: *Object Constraint Language 2.0*. Mitp-Verlag, 2004
- [Wol11] WOLFF, Andreas: *Modellbasierte Generierung von Benutzungsoberflächen*, Universität Rostock, Diss., 2011. – <http://rosdok.uni-rostock.de/resolve?urn=urn:nbn:de:gbv:28-diss2011-0110-1&pdf>
- [Wur11] WURDEL, Maik: *An Integrated Formal Task Specification Method for Smart Environments*, Universität Rostock, Diss., 2011. – 1-226 S
- [XMI05] *XMI Version 2.1*. <http://www.omg.org/spec/XMI/2.1/>, 2005-09-01, 2005

Thesen

1. Es gibt mit imperativen, deklarativen und hierarchischen Sprachen viele Ansätze zur Modellierung von Geschäftsprozessen. Imperative Modellierungssprachen sind mit Ereignisgesteuerten Prozessketten, UML-Aktivitätsdiagrammen und BPMN die populärsten.
2. Die verschiedenen Ansätze zur Prozessmodellierung haben alle ihre spezifischen Eigenschaften und damit Vor- und Nachteile. Ein Nachteil kann bei den imperativen Modellierungssprachen z.B. in der zu starken Restriktivität liegen. Es sind nur die angegebenen Prozesspfade aus dem Modell erlaubt und alle anderen Aktivitätsabfolgen untersagt.
3. Deklarative Prozessmodelle können Workflows flexibler beschreiben als imperative Sprachen. Bei deklarativen Modellierungssprachen sind alle Aktivitätsabfolgen erlaubt, solange sie nicht durch Constraints untersagt sind. Zur UML gehört die textuelle Constraintsprache OCL, die metamodellobasiert für eine deklarativen Modellierung für Geschäftsprozesse genutzt werden kann. Dafür ist der Ansatz zur *Deklarativen metamodellobasierten Workflowmodellierung* (DMWM) in dieser Arbeit umgesetzt worden.
4. Die meisten Modellierungssprachen haben eine grafische Syntax. Sie fördert das Verständnis und ist intuitiver als eine textuelle Syntax. OCL lässt sich mit dem Konzept der Invarianten und Vor- und Nachbedingungen hinter grafischen UML-Modellierungselementen verbergen. Damit wird mit DMWM ein grafischer, deklarativer Modellierungsansatz auf Basis von UML und OCL verfolgt.
5. Weitere Sprachen können analog zur deklarativen Workflowsprache DMWM über UML-Metamodelle definiert werden. Mit diesem Metamodel-Ansatz können genauso domänenspezifische Sprachen beschrieben werden. OCL-Invarianten legen sprachspezifische Konsistenzeigenschaften im Metamodel fest.
6. Ein Nachteil der imperativen Modellierungssprachen kann des Weiteren darin bestehen, dass eine integrierte Zielmodellierung fehlt. Die im HCI-Bereich etablierten Aufgabenmodelle besitzen eine leicht einzusetzende, simple Zielmodellierung, die mit einer Prozessmodellierung kombiniert ist. ConcurTaskTree (CTT) ist die derzeit am weit verbreitetste Aufgabenmodellierungssprache.
7. CTT-Aufgabenmodelle haben Defizite in der Formalisierung. Eine umfangreiche Konsistenzprüfung ist zur Designzeit anhand des Modellierungswerkzeugs CTTE nur teilweise möglich. Mit dem Einsatz von präzisen Metamodellen lässt sich der Ansatz formalisieren. Dafür sind die *metamodellobasierten ConcurTaskTrees* (MCTT) in dieser Arbeit eingeführt worden. Anhand des UML-Tools USE lässt sich eine genauere Konsistenzprüfung von CTT-Modellen durchführen.
8. Mit einer präzisen Soundness-Analyse zur Designzeit können Aufgabenmodelle eingesetzt werden, um Workflows zu beschreiben. Die Baumstruktur stellt sicher, dass die Prozessmodelle strukturiert sind. Ähnlich zum Entwurf von strukturierten Programmen mit Bäumen bei den *Jackson Structured Diagrams* (JSD) können Aufgabenbäume zur strukturierten Workflowmodellierung eingesetzt werden.
9. Die OCL kann bei UML metamodellobasierten Ansätzen für mehrere Zwecke verwendet werden. Zusätzlich zum Prüfen der Soundness-Eigenschaften der Prozessmodelle kann sie zweitens zum plattformunabhängigen Spezifizieren der operationalen Semantik der Workflowsprache und drittens zum Process Mining und der Analyse ausgeführter Prozessinstanzen eingesetzt werden.
10. Eine UML-Action Language, deren imperative Code im Metamodel hinterlegt wird, kann die operationale Semantik der neu definierten Workflowsprache umsetzen. Das UML-Tool USE unterstützt

die zwei Action Languages ASSL und SOIL. Sie werden eingesetzt, um die beiden aufgeführten metamodellbasierten Ansätze DMWM und MCTT ausführbar zu machen.

11. Auf Basis des UML-Metamodells lässt sich eine Daten- und Organisationsmodellierung in die Workflowmodelle integrieren. Somit kann eine grafische Workflow-, Daten- und Organisationsmodellierung vorgenommen werden. Diese Modelle lassen sich dann über UML-Action Languages direkt ohne Transformationsschritte zur Ausführung nutzen.
12. Für die Ausführung bzw. Simulation der Workflowmodelle bedarf es einer geeigneten Nutzerschnittstelle, die für DMWM und MCTT mit dem UML-Tool USE umgesetzt wurde. Sie stellt die Workflowmodelle zur Runtime dar, ohne dass der Nutzer sich um Layoutdarstellungen des Modells zu kümmern braucht. Durch die toolbasierte Modellausführung können die dynamischen Modelleigenschaften überprüft werden. Dies hat bessere und validierte Modelle zur Folge.
13. Die Vorteile von Aufgabenmodellen lassen sich auch mit einem Workflow Management System (WfMS) verbinden, um Workflowmodelle verteilt ausführen zu können. Strukturierte Modelle mit integrierter Zielmodellierung dienen einem besseren Modellverständnis insbesondere für den Endnutzer. Das *Task Tree Based Workflow Management System* (TTMS) setzt Aufgabenmodellkonzepte in einem WfMS um.
14. Aufgabenmodelle haben Defizite in der Entscheidungsmodellierung. Populäre Geschäftsprozessmodellierungssprachen bieten mit expliziten und impliziten Entscheidungsoperatoren zwei Arten dafür an. Hier wurden Aufgabenmodelle zur expliziten Entscheidungsmodellierung mit Decisionnodes und datenbasierten Entscheidungsoperatoren erweitert.