CARL VON OSSIETZKY universität OLDENBURG

Diplomarbeit
im Studiengang Informatik

Zeitanalyse als Aspekt der Usability-Bewertung im Kontext komplexer Aufgaben

Jutta Fortmann

Oldenburg, den 29. November 2010

Gutachter:
Dr. Andreas Lüdtke
Prof. Dr. Susanne Boll

Zusammenfassung

In der vorliegenden Arbeit wurde – angelehnt an ein Szenario aus der Luftfahrt – ein Konzept zur Zeitanalyse anhand von Aufgabenmodellen entwickelt. Dazu wurden zunächst Zeitkonzepte definiert, die im Kontext des Szenarios sinnvolle Modellierungsmöglichkeiten bieten. Dabei erwiesen sich vor allem mengenbezogene, zeitliche Bedingungen zwischen Aufgaben als passend, da diese im sicherheitskritischen Anwendungsbereich besonders relevant sind, in dem die sekundengenaue Bearbeitung von Aufgaben einen entscheidenden Sicherheitsfaktor darstellen kann. Schließlich wurde dieses Konzept als Erweiterung der Aufgabenmodellierungsumgebung PED (Procedure Editor) implementiert. PED wird am OFFIS-Institut für Informatik entwickelt. Die Erweiterung ermöglicht die Berechnung der minimalen, maximalen und mittleren Ausführungszeiten einzelner Aufgabensequenzen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Einleitung

Dieses Kapitel führt den Leser in das Themenfeld dieser Arbeit ein. Außerdem werden Aufgabenstellung und Motivation der Arbeit formuliert. Abschließend folgt eine Beschreibung des Aufbaus dieser Arbeit.

1.1 Einordnung des Themas

Das Design einer Benutzungsoberfläche ist entscheidend für den Erfolg eines Produkts und für die Möglichkeit, mit diesem effizient zu arbeiten. Auch die leistungsstärkste Soft- und Hardware ist nutzlos, wenn der Nutzer auf Grund schlechten Designs oder Ablaufs des Systems langsam, unzufrieden oder sogar ineffektiv arbeiten muss. Auch der Grad, wie sicher ein System bedienbar ist, kann von der Usability dieses Systems abhängen. Die häufigsten Unfälle, die heutzutage in sicherheitskritischen Bereichen, wie z.B. der Luftfahrt geschehen, sind auf die fehlerhafte Bedienung der Benutzungsschnittstelle durch den Bediener zurückzuführen. Um dauerhafte, effiziente und sichere Nutzung von Software zu gewährleisten, ist es deshalb wichtig ihre Usability zu beachten. Usability wird laut DIN EN ISO 9241-11 mit Gebrauchstauglichkeit übersetzt und beschreibt „[...] das Ausmaß in dem ein Produkt durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele **effektiv**, **effizient** und **zufriedenstellend** zu erreichen.“. Diese drei Begriffe sind also als Maßstab für die Gebrauchstauglichkeit zu sehen.

Um die Usability von Benutzungschnittstellen zu verbessern, werden diese hinsichtlich ihrer allgemeinen Gebrauchstauglichkeit oder bestimmten Usability-Kriterien evaluiert und entsprechend gewonnener Erkenntnisse diesbezüglich optimiert. Eine Usability-Evaluation kann mit Hilfe von Aufgabenanalyse-Techniken

durchgeführt werden. Das Ziel der Aufgabenanalyse ist die Analyse und Beschreibung, wie eine Aufgabe erreicht wird. Dabei werden Aufgaben in Teilaufgaben zerlegt. Diese Teilaufgaben werden zunehmend präziser und schließlich in Form elementarer Aktionen formuliert. Die korrekte Zusammenführung dieser elementaren Aktionen führt dann zur Erreichung der Aufgabe bzw. des Ziels auf der obersten Abstraktionsebene. Bei der Evaluation von Benutzungsschnittstellen sind die Aufgaben aus Sicht eines potentiellen Systemnutzers zu formulieren und können so z.B. dabei helfen, aufzudecken, ob sich die tatsächlichen Nutzerziele mit den für das Design angenommenen Nutzerzielen decken. Auch weitere Schwächen hinsichtlich der Gebrauchstauglichkeit des Systems, wie beispielsweise Inkonsistenzen und Schwierigkeiten bei der Aufgabendurchführung, können so ggfs. aufgezeigt werden. Die Zerlegung von Aufgaben wird häufig in Form hierarchischer Strukturen notiert, die zusätzliche Informationen über Abhängigkeiten und Ausführungsreihenfolge liefern können. Bei dieser Struktur spricht man deshalb auch von *Hierarchical Task Analysis* (HTA). Die Modellierung und anschließende Analyse von Aufgaben kann zusätzlich durch Informationen wie beispielsweise Ausführungszeiten, Optionen oder Bedingungen angereichert werden. Dadurch sind umfangreichere und aussagekräftigere Analysen möglich.

Diese Arbeit legt ihren Fokus auf solche komplexen Aufgabenanalysen. Als Anwendungsbeispiel wird die Modellierung von zwei parallel durchzuführenden Aufgaben im Flugzeug-Cockpit betrachtet. Der Pilot soll einerseits die Bordinstrumente kontrollieren und andererseits eine neue Flugroute mit Hilfe einer interaktiven Benutzungsschnittstelle generieren. Um solche komplexen Aufgaben modellieren zu können, wurde am OFFIS - Institut für Informatik ein spezielles Regelsystem entwickelt. Dieses ermöglicht in Verbindung mit Zielhierarchien die Modellierung von Aufgaben im Rahmen einer Aufgabenanalyse. Um die Modellierung solcher Ziele und Regeln zu erleichtern, wird derzeit ebenfalls am OFFIS ein sogenannter *Procedure Editor* (PED) entwickelt. Mit Hilfe dieses Editors können Aufgabenmodelle, bestehend aus u.a. Zielen und Regeln, graphisch erstellt und veranschaulicht werden. Es fehlt allerdings die Möglichkeit, den zeitlichen Ablauf zur Ausführung der Aufgaben explizit zu modellieren. Bisher wird der Ablauf von mit PED modellierten Aufgaben eines Aufgabenmodells so interpretiert, dass die Aufgaben in Modellierungsreihenfolge nacheinander ausgeführt werden, ohne dass explizite zeitliche Abhängigkeiten definiert werden können. Da der Mensch jedoch fähig ist, bestimmte Aufgaben z.B. seriell, sequentiell oder parallel durchzuführen, soll

das Regelsystem von PED in dieser Arbeit um die Darstellung eines Zeitfaktors erweitert werden. Unter Berücksichtigung des Zeitfaktors soll zudem ein Konzept entworfen werden, das die Berechnung zu erwartender Gesamt-Ausführungszeiten modellierter Aufgaben ermöglicht. Dabei sollen jeweils Berechnungen zu Best-/Schlechtest- und Durchschnittsfall ausgegeben werden. Das Konzept orientiert sich an Aufgabenmodellen, die mit PED erstellt wurden und wird schließlich prototypisch implementiert.

1.2 Aufgabenstellung

Ziel dieser Diplomarbeit ist es, ein umfassendes Konzept zur Zeitanalyse im Rahmen Hierarchischer Aufgabenanalysen zu entwickeln. Mittels Literaturrecherche soll dazu zunächst das Thema Usability insbesondere hinsichtlich der verschiedenen Metriken zur Bewertung der Usability eines Systems aufgearbeitet werden. Außerdem soll die Modellierung und Analyse von Aufgaben im Hinblick auf Zeitkonzepte untersucht und in das Forschungsfeld Usability eingeordnet werden. Dabei sollen unterschiedliche Zeitkonzepte, wie z.B. die parallele Ausführung von Aufgaben betrachtet werden. Angelehnt an ein Szenario aus dem Flugzeug-Cockpit, soll das o.g. Regelsystem dahingehend erweitert werden, dass verschiedene Zeitkonzepte (z.B. sequentiell, seriell, parallel) modelliert werden können. Darauf aufbauend soll ein Konzept zur Zeitanalyse inklusive der Berechnung von Ausführungszeiten einzelner Wege innerhalb eines Aufgabenmodells unter Berücksichtigung von zeitlichen Bedingungen erstellt werden. Dabei sollen jeweils Ausführungszeiten für Best-, Schlechtest- und Durchschnittsfall berechnet werden. Schließlich soll das Konzept durch eine prototypische Implementierung umgesetzt werden, die zu einem späteren Zeitpunkt als Erweiterung von PED dienen soll.

1.3 Motivation

Die in dieser Arbeit behandelte Thematik „Zeitanalyse zur Usability-Bewertung“ ist in einem sehr aktuellen und wachsenden Forschungsfeld angesiedelt. Auch in der freien Wirtschaft gewinnt die Berücksichtigung der Usability von Systemen zunehmend an Bedeutung. Für das Anwendungsfeld „Sicherheitskritische Systeme“,

wozu u.a. auch die computergestützten Systeme eines Flugzeug-Cockpits zählen, sind besondere Aspekte relevant, die z.T. über das Kernverständnis von Usability hinausgehen. Denn im sicherheitskritischen Bereich geht es primär um Aspekte, die zur Gewährleistung von Sicherheit beitragen und weniger um jene, die z.B. den Spaß an der Benutzung[1] beschreiben. Die Identifikation relevanter Zeitkonzepte und deren Integration in Zeitanalysen ist in diesem sicherheitskritischen Kontext besonders interessant, da insbesondere die Zeit, die für einen Prozess benötigt wird bzw. die ein Prozess andauert einen entscheidenden Sicherheitsfaktor darstellen kann. Das im Rahmen dieser Arbeit zu entwickelnde Konzept widmet sich Zeitkonzepten, die speziell für Abläufe innerhalb eines Flugzeug-Cockpits relevant sind und schließt mit der Integration dieser Konzepte in eine umfassende Zeitanalyse eine Lücke des aktuellen Forschungsstands.

Darüber hinaus liegt ein starkes persönliches Interesse für das Forschungsfeld *Human-Computer Interaction* und darunter insbesondere der Usability vor. Diesem wurde u.a. bereits in einer Bachelorarbeit nachgegangen, die sich mit ähnlicher Thematik befasst (Fortmann, 2008).

1.4 Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen dieser Arbeit erklärt. Es wird zunächst auf die Normung von Usability eingegangen, bevor Grundlagen der Aufgabenanalyse und -modellierung beschrieben werden. Anschließend werden das im Rahmen dieser Arbeit zu erweiternde Regelsystem und der *Procedure Editor* vorgestellt. Das 3. Kapitel beschreibt den Stand der Forschung. Nachdem kurz auf unterschiedliche Maße und Methoden zur Evaluation von Usability eingegangen wurde, wird die Hierarchische Aufgabenanalyse als spezielle Evaluationsmethode vorgestellt. Es folgt eine ausführliche Beschreibung aktueller Ansätze zur Berücksichtigung und Modellierung von Zeitkonzepten innerhalb von Aufgabenanalysen. Im 4. Kapitel wird das Anwendungsszenario dieser Arbeit vorgestellt. In Kapitel 5 wird die Erweiterung des Regelsystems hinsichtlich zeitlicher Abfolgen präsentiert. Dabei wird auf die ausgewählten Zeitkonzepte und die Erweiterung der Prozeduren-Syntax um diese Zeitkonzepte eingegangen. Das 6. Kapitel umfasst eine detaillierte Beschreibung des Konzepts zur Zeitanalyse. Es beginnt mit der Formulierung

[1] engl.: „Joy of Use“

der Anforderungen und endet mit der Identifikation möglicher Probleme des Konzepts. Im 7. Kapitel wird letztlich die algorithmische Umsetzung des Konzepts präsentiert. Diese beinhaltet neben der Erläuterung der Programmstruktur und der Implementierung ausgewählter Methoden auch die Darlegung eines möglichen Speicherproblems, welches zur Laufzeit des Programms auftreten kann. Abgerundet wird diese Arbeit durch eine Zusammenfassung und einen Ausblick im 8. Kapitel.

Der Anhang dieser Arbeit befasst sich u.a. ausführlich mit verschiedenen Usability-Metriken und Methoden zur Usability-Evaluation, deren Erläuterung zur genauen Einordnung der in dieser Arbeit betrachteten Metrik „Ausführungszeit“ und Methode „Hierarchische Aufgabenanalyse“ in den Gesamtkontext dienen soll.

2 Grundlagen

Dieses Kapitel befasst sich mit der internationalen Normung der Usability und der im Fokus dieser Arbeit stehenden Aufgabenanalyse als Bewertungsverfahren für die Usability eines Systems. Darüber hinaus werden die Aufgabenmodellierungsumgebung PED und ein spezielles Regelsystem vorgestellt, welches intern von PED verwendet wird und im Rahmen dieser Arbeit erweitert wird.

2.1 Die Usability-Normung

Der Begriff Usability wurde in der Normenreihe DIN EN ISO 9241: „Ergonomie der Mensch-System-Interaktion“ international definiert und spezifiziert. Diese Reihe bestand bis März 2006 aus insgesamt 17 Teilen, wurde seitdem aber um einige Normen erweitert. Wie in Kapitel 1.1 bereits zitiert, sind laut DIN EN ISO 9241-11 Effektivität, Effizienz und Zufriedenstellung zentrale Begriffe bei der Definition von Gebrauchstauglichkeit.
Der Titel der Normenreihe lautet erst seit dem 1.4.2006 mit der Überarbeitung der Norm 9241-10 so. Zuvor hieß sie „Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten“. Da die Beachtung von Usability mittlerweile längst nicht mehr nur bei Büroarbeit wichtig ist, wurde diese Einschränkung aufgehoben. Des Weiteren werden nun alle interaktiven Systeme, d.h. sowohl Software als auch Hardware, angesprochen. Dies wird in der Neudefinition der Benutzungsschnittstelle[1] deutlich, wie sie in der DIN EN ISO 9241-110 formuliert ist. Danach zählen zu einer Benutzungsschnittstelle[2] „[...] alle Bestandteile eines interaktiven Systems (Software oder Hardware), die Informationen und Steuerelemente zur Verfügung stellen, die für den Benutzer notwendig sind, um eine bestimmte Arbeitsaufgabe

[1] Dieser Begriff wird äquivalent zu dem Begriff „Benutzungsoberfläche“ verwendet
[2] engl.: User Interface

mit dem interaktiven System zu erledigen.“.
Diese grundlegenden Überarbeitungen der Normenreihe zeigen, dass die Einhaltung von Usability als zunehmend wichtiger erachtet wird. Zudem ist die Relevanz von Usability durch die rasante Weiterentwicklung in der Informationstechnologie längst nicht mehr nur im anfänglichen Bürobereich präsent, sondern bereits in die unterschiedlichsten Anwendungsfelder vorgedrungen, nämlich in solche, bei denen eine Interaktion zwischen Mensch und Maschine eine zentrale Rolle spielt, was heutzutage in nahezu jedem Anwendungsfeld der Fall ist. Die DIN EN ISO 9241-10 legt in sieben Grundsätzen allgemeingültige Verhaltensregeln für interaktive Systeme fest. Dazu gehören die Aufgabenangemessenheit, die Selbstbeschreibungsfähigkeit, die Erwartungskonformität, die Fehlertoleranz, die Steuerbarkeit, die Individualisierbarkeit sowie die Lernförderlichkeit eines interaktiven Systems. Aus der Überarbeitung dieser Norm ist die DIN EN ISO 9241-110 „Grundsätze der Dialoggestaltung“ als Nachfolgerin entstanden, die die sieben Grundsätze grundsätzlich beibehält, aber weiter präzisiert. Die sieben Grundsätze beschreiben im Einzelnen Folgendes:

- **Aufgabenangemessenheit:** Ein interaktives System ist aufgabenangemessen, wenn es seinen Benutzer dabei unterstützt, seine Aufgabenziele vollständig und korrekt und mit einem annehmbaren Aufwand zu erledigen. Nicht aufgabenangemessen wäre z.B. ein Formular, bei dem mehrere Eingabefelder in einer für den Menschen untypischen Reihenfolge angeordnet sind. Sollte z.B. eine Adresse mit Namen, Straße, Hausnummer, Postleitzahl und Wohnort eingegeben werden und die Reihenfolge wäre nicht standardmäßig wie oben angegeben, sondern der Ort käme beispielsweise vor der Straße, dann würde sich das System hier unangemessen verhalten und eine ineffiziente Erledigung der Aufgabe verursachen.
- **Selbstbeschreibungsfähigkeit:** Ein interaktives System ist selbstbeschreibungsfähig, wenn für den Benutzer jederzeit offensichtlich ist, in welchem Dialog und an welcher Stelle im Dialog er sich befindet. Außerdem muss stets deutlich sein, welche Handlungen der Nutzer unternehmen kann und wie diese auszuführen sind. Wird beispielsweise in einem Formular keine Anmerkung zu einem Eingabefeld gegeben, dessen Formatierung unklar ist (Beispiel: Datumseingabe), so wird dieser Grundsatz hier missachtet.
- **Erwartungskonformität:** Ein interaktives System ist erwartungskonform,

wenn es konsistent ist und den Merkmalen des Benutzers entspricht. Diese Merkmale sind beispielsweise seine Kenntnisse aus dem Arbeitsgebiet, aus seinen Erfahrungen und aus allgemein anerkannten Koventionen, wie z.B. der Norm 9241. Das Programm „WinZip“ zum Komprimieren von Dateien beispielsweise weist keine Funktion „Komprimieren“ auf, sondern versteckt diese hinter der Funktion „Neu“. Der Nutzer hat die Erwartungshaltung, bei einem Komprimierprogramm auch eine „Komprimieren-“Funktion zu finden, wird aber enttäuscht und ist gezwungen, Vermutungen nachzugehen und auszuprobieren. Das Dialogprinzip der Erwartungskonformität wird hier also verletzt.

- **Fehlertoleranz:** Ein interaktives System ist fehlertolerant, wenn es den Benutzer vor Fehleingaben bewahrt und im Fehlerfall unmittelbar auf die Fehleingabe hinweist und den Nutzer dann konstruktiv bei der Behebung des Fehlers unterstützt. Ein Beispiel hierfür ist die Reaktion des Betriebssystemes Microsoft Windows Vista, wenn man versucht, einer Datei im Explorer oder auf dem Desktop einen Namen mit bestimmten Sonderzeichen, wie z.B. dem Slash („/“), zu geben. Es erscheint eine Meldung, die darauf hinweist, dass solche Zeichen unzulässig sind, und die Eingabe des Zeichens wird gleichzeitig nicht angenommen bzw. verarbeitet.

- **Steuerbarkeit:** Ein interaktives System ist steuerbar, wenn der Nutzer in den Dialogablauf eingreifen kann, d.h. über Start, Richtung und Rückschritte bestimmen kann. Der Nutzer muss also jederzeit alle Aktivitäten des Systems steuern und auch vorangegangene Schritte rückgängig machen können. Ein Beispiel wäre die Rückgängig-Funktion in Graphik- oder Textverarbeitungsprogrammen, die z.B. einen gerade gelöschten Absatz wiederherstellen kann.

- **Individualisierbarkeit:** Ein interaktives System sollte sich individuell auf die Bedürfnisse und Vorlieben seines Benutzers einstellen können. Beispiele wären hier die Möglichkeit, die Schrift zu vergrößern, oder zusätzliche Menüs und Funktionen an- und abschalten zu können. Vor zu starker Anpassung ist allerdings auch Vorsicht geboten. Menüelemente oder Icons, die sich ständig, je nach Häufigkeit der Verwendung durch den Nutzer unterschiedlich anordnen, können zu mehr Irritation als Nutzen führen. Ein Beispiel für eine Anwendung, die dieses Kriterium umsetzt, ist Microsoft Word. Die Anord-

nung der Icons in der Werkzeugleiste von Microsoft Word passt sich ständig an die Zugriffe des Nutzers an.

- **Lernförderlichkeit:** Ein interaktives System sollte das Erlernen von Bedienung und Funktionalität fördern, d.h., der Benutzer sollte durch intuitive Darstellung und Hilfestellungen möglichst schnell und einfach den Umgang mit dem System erlernen können. Dabei helfen z.B. aussagekräftige Icons und Bezeichner für bestimmte Funktionen, oder die Anordnung von Standardfunktionen in oberster Menüebene. Ein weiteres Beispiel ist die Verwendung von Unterstrichen unter Funktionsbezeichnern in Menüs, um Kommandokürzel anzugeben, wie es bei vielen Microsoft-Programmen angewendet wird. Lernförderlichkeit bezieht sich also auch auf das Erlernen alternativer Methoden zur Systemsteuerung.

Neben der oben erläuterten Normenreihe spielt hier ebenfalls die Norm ISO 13407 eine wichtige Rolle. ISO 13407 „Benutzer-orientierte Gestaltung interaktiver Systeme“ stellt ein grundsätzliches Vorgehensmodell vor, wie der Benutzer durchgängig in den Entwicklungsprozess mit einzubinden ist und fordert dieses gleichzeitig. Durch die frühzeitige und dauerhafte Einbeziehung der Nutzerbedürfnisse sollen Usability-Mängel und daraus resultierender zusätzlicher Optimierungsaufwand, verbunden mit hohen Kosten, vermieden werden. Laut der Norm soll ein iterativer Prozess, der parallel zur Entwicklung der Software abläuft, die Gebrauchstauglichkeit dieser gewährleisten. Dabei soll zunächst der Nutzungskontext erfasst werden, um darauffolgend die Anforderungen an die Software und auch organisatorische Anforderungen zu ermitteln. Damit können dann Gestaltungsvorschläge für die Software gemacht und diese letztlich bewertet werden. Allerdings handelt es sich hierbei lediglich um ein allgemeines Vorgehensmodell, nicht jedoch hält die ISO 13407 konkrete Gestaltungsvorschläge für den Entwicklungsprozess vor. Hierfür eignet sich der DATech[3] Prüfbaustein Usability-Engineering-Prozess, ein Verfahren zur Überprüfung der Normkonformität mit DIN EN ISO 9241-10/-11 und DIN EN ISO 13407. Solch ein definiertes Prüfverfahren erleichtert Prüfern den Vergleich von Prüfergebnissen und deren Reproduktion. Verbindlich angewendet wird dieses Prüfverfahren von akkreditierten Prüfstellen. Es gilt allerdings nicht als allgemeingültiges Prüfverfahren, sondern vielmehr als eine geeignete Richtung, die sich in der Konformitätsprüfung bewährt hat. Das Prüfverfahren ist im Einzelnen in dem

[3]Deutsche Akkreditierungsstelle Technik e.V.

„Leitfaden Usability“ (DATech, 2009) veröffentlicht.

2.2 Aufgabenanalyse und -modellierung

Um die Usability von interaktiven Systemen, u.a. anhand der o.g. Kriterien messen zu können, werden verschiedene Verfahren angewendet. Die Aufgabenanalyse und -modellierung als ein Verfahren zur Usability-Bewertung ist zentraler Bestandteil dieser Arbeit und wird im Folgenden kurz vorgestellt.

2.2.1 Aufgabenanalyse

Bei der Aufgabenanalyse wird untersucht, wie Menschen bestimmte Arbeiten, d.h. Aufgaben verrichten. Dabei werden die Aktionen, die sie ausführen und die Objekte, auf die sie dabei einwirken, untersucht. Außerdem wir das Wissen ermittelt, das sie dafür benötigen. Aufgabenanalysen stellen den Nutzer eines Systems in den Vordergrund und nicht das System selbst.

Es gibt verschiedene Ansätze der Aufgabenanalyse, die ihre Schwerpunkte hinsichtlich der o.g. Informationen jeweils unterschiedlich legen. Bei der *Aufgabenzerlegung* wird die Struktur der Aufgabe, d.h. in welche Unteraufgaben die Aufgabe zerlegt werden kann und in welcher Reihenfolge diese ausgeführt werden, betrachtet. Die *wissensbasierten Techniken* fokussieren das Wissen, welches der Nutzer über aufgabenbeteiligte Objekte und Aktionen haben muss und untersuchen zudem, wie dieses Wissen organisiert ist. Bei der *Entität-Relations-basierten Analyse* werden die bei der Aufgabe beteiligten Akteure, Objekte und Aktionen identifiziert und ihre Zusammenhänge untereinander dargestellt (Dix et al., 2004). Der Fokus dieser Arbeit liegt auf der Aufgabenzerlegung, auf die in Abschnitt 3.2 genauer eingegangen wird.

Aufgabenanalysen können bei der Erstellung von Übungsmaterialien, Handbüchern und Dokumentationen bzgl. eines Systems bzw. einer Software helfen. Aber auch bei der Verbesserung und Entwicklung von Systemen können sie unterstützen, indem sie z.B. zur Erfassung und Organisation des Wissens über eine bestimmte Situation beitragen und dadurch helfen, Schwachstellen eines Systems aufzudecken, die die menschliche Leistung beeinflussen (Dix et al., 2004).

2.2.2 Aufgabenmodellierung

Ein Aufgabenmodell stellt die strukturelle Repräsentation einer Aufgabenanalyse dar. In dem Aufgabenmodell werden die Aktionen, die für die Durchführung einer Aufgabe erforderlich sind, strukturiert festgehalten. Zur graphischen Darstellung eines Aufgabenmodells eignet sich ein gerichteter Graph mit Knoten und Kanten. Die Hierarchie ergibt sich durch die Zerlegung der Aufgaben in immer feinere Teilaufgaben, die durch die Knoten repräsentiert werden und über Kanten miteinander verbunden sind. An oberster Stelle steht das Wurzelelement, welches die oberste Aufgabe in abstrakter Form repräsentiert. An unterster Stelle stehen Operatoren, d.h. feingranulare Aktionen. Die Leserichtung ist üblicherweise von oben nach unten und damit von grob zu fein bzw. abstrakt zu konkret. Zusätzlich können Bedingungen an den Kanten festlegen, dass bestimmte Unteraufgaben nur unter bestimmten Voraussetzungen ausgeführt werden. Zur Durchführung der obersten Aufgabe, dargestellt durch den Wurzelknoten, müssen alle darunter liegenden Teilaufgaben durchgeführt werden, sofern dies nicht durch Bedingungen anders spezifiziert ist. Zu unterscheiden ist dies von einem einzelnen Pfad durch die Aufgabenhierarchie, welcher nur einen geringen Teil der Aufgaben berücksichtigt, die zur Durchführung der obersten Aufgabe nötig sind. Ist ein Aufgabenmodell erstellt, dient dieses wiederum zur weiteren Analyse. Somit kann ein Aufgabenmodell sowohl Ergebnis einer ersten, als auch Grundlage einer weiteren Aufgabenanalyse sein. Der Konstruktionsvorgang eines Aufgabenmodells wird Aufgabenmodellierung genannt.

Es existieren bereits zahlreiche Werkzeuge, die die Modellierung und Analyse von Aufgaben unterstützen. Im Rahmen dieser Arbeit wird das Modellierungswerkzeug PED erweitert, welches in Kapitel 2.4 vorgestellt wird. Auf einige weitere Werkzeuge zur Aufgabenanalyse im Hinblick auf Zeitmodellierung wird in Kapitel 3.3.4 eingegangen.

2.2.3 Ordnungen zur groben Ablaufbestimmung in Aufgabenmodellen

Eine Ordnung bedeutet die Sortierung einer Menge von Elementen. Die einzelnen Elemente der Menge stehen in einer bestimmten Relation zueinander, sodass sie

vergleichbar sind. Die Reihenfolge zwischen Aufgaben in Aufgabenanalysen kann durch Ordnungen angeben werden, sodass dadurch der Ablauf eines Aufgabenmodells spezifiziert wird. Dabei werden die einzelnen Aufgaben als Elemente einer Menge interpretiert, sodass Relationen zwischen den Aufgaben definiert werden können. Bei einer totalen Ordnung sind z.B. alle Elemente der Menge in einer bestimmten Reihenfolge angeordnet. Es gibt folglich keine Spielräume bei der Positionierung eines Elements in der Menge. Eine Sequenz kann z.B. durch eine totale Ordnung modelliert werden. Sie legt eine bestimmte Reihenfolge fest, in der die einzelnen Aufgaben nacheinander ausgeführt werden müssen. Im Gegensatz dazu können auch keine Ordnungsrelationen zwischen Elementen definiert sein. Dies ist z.B. der Fall, wenn das Aufgabenmodell Zyklen enthält. Trotz Zyklen repräsentiert der Graph zwar eine mathematische Relation, aber dabei handelt es sich nicht um eine Ordnungsrelation. Ein Spezialfall einer Ordnung ist die partielle Ordnung (auch: Halbordnung). Sie beschreibt eine teilweise geordnete Menge von Elementen, d.h., dass nicht alle Elemente zueinander in einer bestimmten Ordnung stehen: Die Menge ist nur teilweise eindeutig sortierbar. Serielle oder parallele Abläufe können grob durch partielle Ordnungen definiert sein, da nur teilweise eine Ausführungsreihenfolge festgelegt wird.

Das im Rahmen dieser Arbeit zu erweiternde Aufgabenmodellierungswerkzeug PED (s. Abschnitt 2.4) definiert bisher noch keine expliziten Ordnungen zwischen Aufgaben. Um realistische Aufgaben modellieren zu können, ist die Definition von zeitlichen und damit Ordnungsrelationen jedoch unabdingbar. Aus diesem Grund wird im Rahmen dieser Arbeit ein Konzept entwickelt, das es ermöglicht, Aufgaben zueinander in zeitliche Relationen zu setzen.

2.3 Regelsystem zur Aufgabenmodellierung

Das im Fokus dieser Arbeit stehende Regelsystem (Lüdtke et al., 2009) dient zur Modellierung komplexer Aufgaben im Rahmen einer Aufgabenanalyse. Dieses Regelsystem basiert auf der bekannten GOMS-Sprache (John und Kieras, 1994; s. a. Anhang III). Es setzt sich aus mehreren Regeln zusammen, die als Produktionsregeln interpretiert werden können, d.h. sie folgen einem Wenn-Dann-Muster (s. a. Anhang III). Die linke Seite einer Regel (left-hand side, abgek. LHS) kann

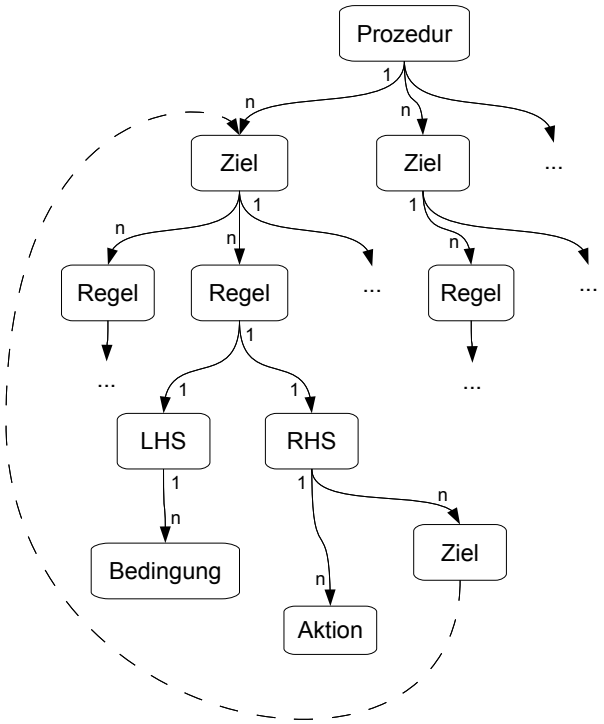


Abb. 2.1: Der vereinfachte Aufbau des Regelsystems

Bedingungen spezifizieren, die erfüllt sein müssen, um den rechten Teil der Regel (right-hand side, abgek. RHS) ausführen zu können bzw. die Regel feuern zu können. Bedingungen können auch solche sein, die immer erfüllt sind. Die rechte Seite definiert die auszuführenden Aktionen bzw. Unterziele. Abbildung 2.1 veranschaulicht diesen vereinfachten Aufbau des Regelsystems.

Eine Regel ist genau einem Ziel zugeordnet, welches auf der LHS der Regel steht, und beschreibt alle Unterziele und atomaren Aktionen, die erforderlich sind, um das Ziel zu erreichen. Diese benötigten Ziele und Aktionen stehen auf der RHS der Regel. Atomare Aktionen werden nicht weiter verfeinert und sind somit auf der untersten Hierarchieebene angesiedelt. Ein Ziel kann durch mehrere Regeln beschrieben werden. Ist dies der Fall, so beschreiben die unterschiedlichen Regeln Alternativen, die jeweils zur Erreichung des Ziels führen. Ein oder mehrere Ziele beschreiben wiederum die Prozedur. Diese definiert das oberste Ziel und beinhaltet somit die gesamte Aufgabenhierarchie.

2.4 Aufgabenmodellierungsumgebung PED

PED[4] und ist eine Modellierungsumgebung zur Erstellung von formalen Aufgabenmodellen, welche dafür das im vorherigen Abschnitt vorgestellte Regelsystem nutzt, d.h. u.a., dass Aufgaben in PED als zu erreichende Ziele interpretiert werden. PED wird seit 2006 am OFFIS–Institut für Informatik von der Gruppe *Human Centered Design* entwickelt.

PED unterstützt die graphische und textuelle Modellierung von Prozeduren, d.h. Aufgaben. Das Werkzeug wurde mit Hilfe des Eclipse Modeling Framework (EMF) und des Eclipse Graphical Modeling Framework (GMF) erstellt und wird derzeit weiterentwickelt. Abbildung 2.2 zeigt einen Screenshot der aktuellen Version 4.0 von PED.

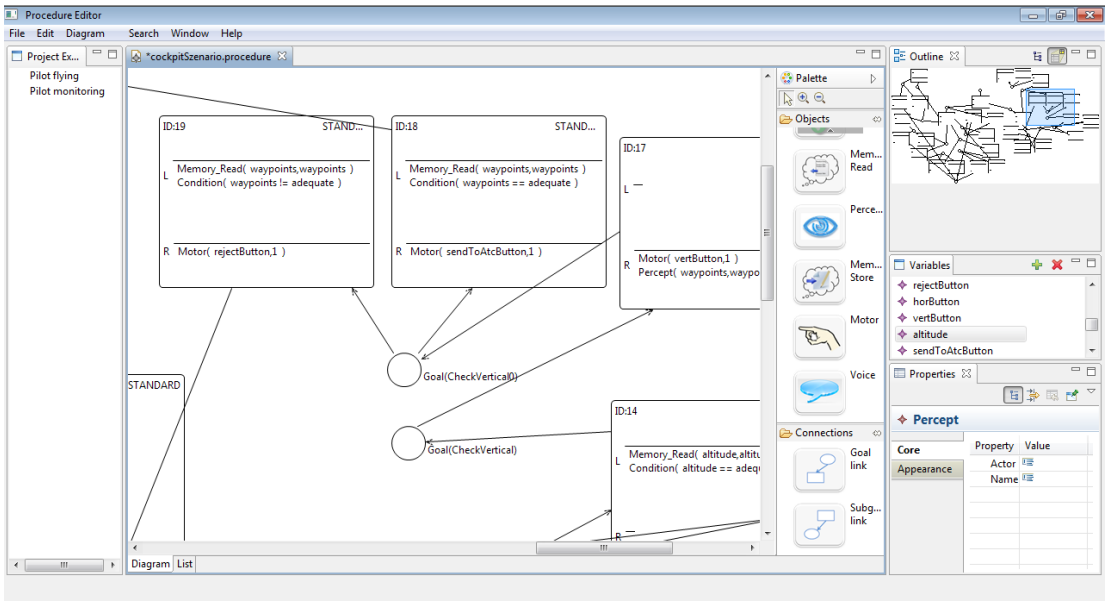


Abb. 2.2: Screenshot des Aufgabenmodellierungswerkzeugs PED

Zu sehen ist ein geöffnetes Dokument im Procedure Editor. Die graphische Modellierung wird im Hauptfenster, das sich in der Mitte befindet, angezeigt und vorgenommen. Rechts daneben ist die Palette angeordnet, die die einzelnen Modellierungselemente wie z.B. Ziele, Regeln und Aktionen, bereithält. Per Drag and Drop werden diese Elemente in das Modellierungsfenster gezogen. So wird eine Prozedur mit Zielen und Regeln modelliert. Ist eine Regel im Modellierungsfenster

[4] Procedure Editor

ausgewählt, erscheinen die einzelnen Variablen und deren Werte im rechten *Variables*-Fenster. Im darunter liegenden *Properties*-Fenster können alle Eigenschaften einer Regel inkl. beteiligter Ziele, Unterziele und Verbindungen eingesehen und editiert werden. Der *Project Explorer* am linken Bildschirmrand listet alle Prozeduren zu den jeweiligen Projekten auf. Das Fenster oben rechts stellt den *Outline* dar, eine Miniaturskizze der gesamten modellierten Prozedur. Dieses Fenster dient zur schnellen Navigation innerhalb der Modellierung. Mit Hilfe der Tabs *Diagram* und *List* kann zwischen der graphischen und der textuellen Ansicht gewechselt werden. Die textuelle Ansicht wird automatisch aus dem graphischen Modell generiert und zeigt die Modellierungselemente in einer hierarchischen Struktur.

PED stellt Ziele als Kreise und Regeln als Rechtecke dar. Die Unterziele werden graphisch aus den Regeln herausgezogen, um die Zusammenhänge zwischen den Zielen zu verdeutlichen.

Im Rahmen dieser Arbeit wird das intern von PED genutzte Regelsystem um zeitliche Relationen erweitert. Darauf aufbauend wird ein algorithmisches Konzept entwickelt, welches automatische Zeitanalysen anhand von in PED modellierten Aufgabenmodellen ermöglicht.

3 Stand der Forschung

In diesem Kapitel werden einige für diese Arbeit relevanten Kriterien vorgestellt, anhand derer die Usability eines Systems gemessen werden kann. Außerdem wird die in dieser Arbeit betrachtete Hierarchische Aufgabenanalyse als Usability-Evaluationsverfahren vorgestellt. Im letzten Abschnitt wird ausführlich auf die Berücksichtigung und Modellierung von Zeitkonzepten innerhalb von Aufgabenanalysen eingegangen.

3.1 Maße zur Evaluation von Usability

Um Aussagen über den Grad der Usability eines Systems treffen zu können, muss diese messbar gemacht werden. Es gibt eine Reihe von Usability-Maßen, die genau dieses Ziel verfolgen. Mit Hilfe entsprechender Verfahren und Methoden können die gewünschten Maßzahlen erhoben und analysiert werden.

3.1.1 Definition von Metrik und Maß

Eine Metrik bezeichnet allgemein ein Maßzahl-System, d.h. die Abbildung von Eigenschaften in Zahlenwerte (auch Maßzahlen). Die abzubildende Eigenschaft kann als Maß bezeichnet werden, wobei der quantitative Zahlenwert, der die Ausprägung des Maßes darstellt, die Maßzahl ist. Die einem Maß zugeordneten Maßzahlen quantifizieren oder messen folglich Eigenschaften, sodass sie zu Vergleichs- und Bewertungszwecken herangezogen werden können.

In dem IEEE Standard 1061 ist eine Softwaremetrik wie folgt definiert:

> *„Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.“*

Qualitätseigenschaften können dabei z.B. neben Erstellungsaufwand, Code-Komplexität, Wartbarkeit oder Fehlerfreiheit auch Zuverlässigkeit, Effizienz und Benutzungsfreundlichkeit sein.

In dieser Arbeit liegt der Fokus auf Usability-Metriken, d.h., die zu messenden Qualitätseigenschaften beziehen sich ausschließlich auf Aspekte der Usability. Wie in Abschnitt 1.1 beschrieben, zählen zu diesen Aspekten die Effizienz und Effektivität der Interaktion, sowie die Zufriedenheit des Nutzers mit bzw. während der Interaktion. Im Folgenden werden verschiedene Maße (vgl. Tullis und Albert, 2008) vorgestellt, mit denen die Usability von Systemen gemessen werden kann.

3.1.2 Klassifizierung von Usability-Maßen

Es gibt eine Reihe von Maßen, anhand derer versucht wird, die Usability eines Systems zu messen. Diese orientieren sich an den in DIN EN ISO 9241-11 aufgeführten Usability-Leitkriterien: “Effizienz, Effektivität und Zufriedenheit”. Allerdings erschweren u.a. die unpräzisen Definitionen von Usability die Ernennung eines allgemeingültigen Usabilitymaßes. Erstens ist die Gebrauchstauglichkeit eines Systems von zahlreichen, vor allem kontextbezogenen Faktoren abhängig und zweitens spielen dabei auch subjektive Empfindungen der Zielnutzer eine entscheidende Rolle.

Usability-Maße sind quantitativ messbare Kriterien, deren Messwerte es ermöglichen, Aussagen hinsichtlich der Usability eines Systems zu machen. Die Quantitativität ermöglicht zudem, verschiedene Systeme hinsichtlich bestimmter Usability-Maße zu vergleichen. Abbildung 3.1 zeigt eine Aufteilung von gängigen Usability-Maßen nach performanzorientiert, problembasiert, selbsteinschätzend, verhaltensorientiert und physiologisch, sowie kombinatorisch und vergleichend.

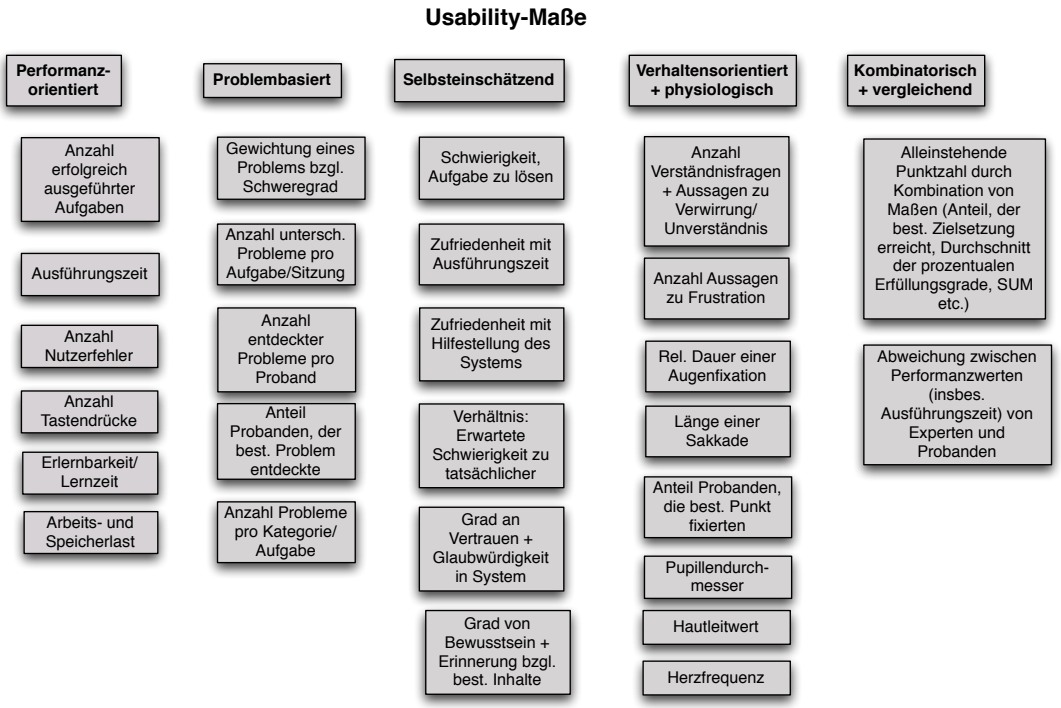


Abb. 3.1: Übersicht einiger gängiger Usability-Maße

Performanz-Maße

Maße, die Aussagen hinsichtlich der Performanz der Interaktion zwischen Nutzer und System treffen, werden häufig zur Usability-Bewertung herangezogen. Dies liegt vor allem daran, dass sie anhand des Verhaltens von Nutzern erhoben werden und somit i.d.R. ein direkter Zusammenhang zwischen Messung und Verbesserungspotentialen erkennbar ist. Performanz-Maße werden gemessen, während Nutzer festgelegte Aufgaben mit dem zu testenden System bearbeiten, wie es bei Usability-Tests (s. Abschnitt 3.1.3) der Fall ist. Es folgt die Auflistung einiger gängiger Performanz-Maße.

Anzahl erfolgreich ausgeführter Aufgaben Dieses Maß bewertet die Effektivität des Systems. Effektiv ist ein System dann, wenn der Nutzer in der Lage ist, seine Aufgaben erfolgreich auszuführen. Die Definition vom Aufgabenerfolg kann dabei unterschiedlich sein. Dieser kann z.B. in der Beendigung und kompletten Zielerfüllung einer Aufgabe liegen oder in Form von Abstufungen die Erreichung unterschiedlicher Teilziele oder Inanspruchnahme von Hilfe berücksichtigen.

Ausführungszeit Die Ausführungszeit, die ein Nutzer zur Durchführung einer Aufgabe benötigt, gibt Hinweise auf die Effizienz eines Systems. Braucht ein Nutzer länger als angemessen für die Durchführung einer Aufgabe, so deutet dies i.d.R. auf eine ineffiziente Interaktion hin. Besonders wichtig ist die Effizienz bei wiederholt ausgeführten oder sicherheitskritischen Aufgaben, bei denen im schlimmsten Fall Menschenleben von der Ausführungszeit abhängen können.

Anzahl Nutzerfehler Fehler sind inkorrekte Aktionen, die zum Misserfolg einer Aufgabe führen können. Im ungünstigsten Fall können sie folglich die Effektivität einer Interaktion beeinflussen. Aber auch wenn Fehler nicht zum Misserfolg einer Aufgabe führen, können sie die Durchführung dieser erschweren und zeitlich verlängern. Sie sind daher insbesondere ein Maß für die Effizienz eines Systems. Macht der Nutzer Fehler während der Interaktion, kann dies für ein ineffizientes System sprechen.

Anzahl Tastendrücke Der physische Aufwand, ausgedrückt z.B. in der Anzahl an Tastendrücken oder Mausklicks, kann Auskunft darüber geben, wie effizient eine Interaktion ist. Sind zur Durchführung einer Aufgabe viele Tastendrücke notwendig, kann dies auf eine ineffiziente Interaktion hindeuten. Je mehr Tastendrücke ein Nutzer benötigt, desto mehr Zeit wird die Bearbeitung der Aufgabe voraussichtlich auch in Anspruch nehmen.

Erlernbarkeit/Lernzeit Die Erlernbarkeit eines Systems wird anhand des Aufwands ausgedrückt, mit dem ein Nutzer über einen Zeitraum Fertigkeiten im Umgang mit dem System erlernt. Für das Maß ist dabei wichtig, wieviel Zeit und kognitiven Aufwand der Nutzer investieren muss, um geübt mit dem System umgehen zu können. Das Maß macht folglich Aussagen über die Effizienz einer Interaktion. Erhoben werden kann die Lernzeit z.B. empirisch, indem die Ausführungszeiten von Nutzern bzgl. einer bestimmten Aufgabe mehrmals gemessen und addiert werden, bis die einzelne Ausführungszeit nahe der Zeit liegt, die ein Experte benötigt. Die Summe der bis dahin gemessenen Ausführungszeiten ergibt dann die Lernzeit, die der Nutzer für den erreichten Geübtheitsgrad benötigt hat.

Arbeits- und Speicherlast Die kognitive Komplexität einer Interaktion kann durch die Arbeits- und die Speicherlast ausgedrückt werden, die bei der Durchführung von Aufgaben anfällt. Fallen diese Lasten gering aus, so unterstützt

das System den Nutzer bei einer effizienten Interaktion. Die Arbeits- und Speicherlast eines Nutzers erhöht sich beispielsweise, wenn dieser Funktionen auf der Benutzungsoberfläche des Systems sucht, eine Entscheidung bzgl. der Ausführung einer Aktion trifft oder Ergebnisse ausgeführter Aktionen interpretiert. Gemessen werden kann diese Komplexität mit Hilfe formaler Analysen, bei denen z.B. die Anzahl von Informationsblöcken (sog. *Chunks*[1]), die sich zu einem bestimmten Zeitpunkt im Kurzzeitgedächtnis des Nutzers befinden, ermittelt wird.

Da in dieser Arbeit auschließlich Performanzmaße betrachtet werden, wird an dieser Stelle auf die Erläuterung weiterer in Abbildung 3.1 aufgeführten Maße verzichtet. Die Abbildung soll lediglich einen groben Überblick über die Einordnung der Performanzmaße in das Forschungsfeld der Usability-Maße geben, ohne vom Argumentationsfluss dieser Arbeit abzulenken. Eine ausführliche Erläuterung aller weiteren abgebildeten Maße ist in Anhang II zu finden.

Diese Arbeit legt ihren Schwerpunkt auf das Performanzmaß Ausführungszeit. Dies ist darin begründet, dass der im Rahmen dieser Arbeit zu entwickelnde Algorithmus in einen Editor integriert werden soll, der auf Basis Hierarchischer Aufgabenanalysen formale Usability-Bewertungen ermöglicht. Performanzmessungen lassen sich sinnvoll in Hierarchische Aufgabenanalysen einfügen, da die hierachische Struktur der Aufgabenmodelle durch die Aufgliederung in einzelne Operatoren die Berechnung solcher Maße, insbesondere von Ausführungszeiten, vereinfacht. Die Messung anderer Maße involviert größtenteils empirische Usability-Tests, d.h. die Beteiligung von Probanden. Da in dieser Arbeit formale Usability-Analysen fokussiert werden, die gänzlich auf Nutzerbeteilung verzichten, sind solche Maße ungeeignet, und es bietet sich stattdessen die Eingliederung von Performanzmaßen zur Stärkung der Usability-Analyse an.

3.1.3 Methoden zur Evaluation von Usability-Maßen

Bei der Erhebung von Maßen mit Hilfe sogenannter Usability-Evaluationen kann neben den o.g. Einflussfaktoren auch die Erfahrung und das Engagement der Ana-

[1] Chunks sind Speichereinheiten im Kurzzeitgedächtnis, die aus Gruppierungen von zusammenhängenden Informationen bestehen. Es wird angenommen, dass maximal 7 +/- 2 Chunks im Kurzzeitgedächtnis gespeichert werden können, wobei die Informationsmenge pro Chunk sehr hoch sein kann.

lysten entscheidend für den Ausgang der Messung sein (de Kock et al., 2009). Die Usability-Bewertung eines Systems kann grob von zwei unterschiedlichen Perspektiven aus durchgeführt werden. Einerseits können konkrete Usability-Probleme identifiziert werden, andererseits kann eine Aussage über die Gesamt-Usability des Systems vorgenommen werden. Zur Aufdeckung konkreter Usability-Probleme dient die formative Evaluation, die meistens während der Entwicklung eines Systems an Prototypen vorgenommen wird und Hinweise auf sinnvolles Redesign gibt. Bei der summativen Evaluation wird i.d.R. ein existierendes System hinsichtlich seiner allgemeinen Gebrauchstauglichkeit analysiert (Sarodnick und Brau, 2006, S. 20).

In dieser Arbeit wird das Evaluationsverfahren der Hierarchischen Aufgabenanalyse angewendet, das im Folgenden erläutert wird.

3.2 Hierarchische Aufgabenanalysen zur Usability-Evaluation

Die Aufgabenanalyse wurde bereits in Abschnitt 2.2 vorgestellt. Im Rahmen dieses Abschnitts ist sie insbesondere als Verfahren zur Usability-Evaluation interessant. Aufgabenanalysen können helfen, Schwächen in der Usability eines Systems aufzudecken. Anhand der Ergebnisse einer solchen Aufgabenanalyse muss der Analyst Hypothesen aufstellen, die die Fehler erklären und letztlich zu Verbesserungsvorschlägen führen, die vom Analysten formuliert werden. Eine anschließende Validierung der aufgestellten Hypothesen ist empfehlenswert. Eine Aufgabenanalyse ist u.a. folglich nur ein Hilfsmittel zur Aufdeckung von Problemen, liefert aber keine Erklärungen oder Lösungsvorschläge. Somit sind die Erfahrung und das Wissen des Analysten entscheidend für den Erfolg einer solchen Evaluation (Diaper und Stanton, 2004).

Bei der Hierarchischen Aufgabenanalyse (Diaper und Stanton, 2004) wird eine Aufgabe funktional in Form von Operationen bzw. Zielen betrachtet. Die Aufgabe stellt das oberste Ziel dar, welches in Subziele zerlegt wird, deren Erreichung bzw. Ausführung zusammen wiederum das oberste Ziel ergibt. Subziele können in weitere Subziele zerlegt werden, sodass eine hierarchische Struktur entsteht, bei der die unterste Ebene die feinsten, d.h. konkretesten Ziele beinhaltet, die zur Erreichung

des obersten Ziels notwendig sind. Je nach Ziel der Analyse kann der Detailgrad der untersten Ebene unterschiedlich ausgeprägt sein. Häufig sind die elementaren Ziele Operationen wie motorische oder kognitive Handlungen. An die Ausführung einer Operation können Konditionen geknüpft sein. Eine Kondition kann beispielsweise darin bestehen, dass erst ein Ziel erreicht sein oder dass eine Variable einen bestimmten Wert haben muss, bevor ein bestimmtes anderes Ziel bearbeitet werden darf. Auch das Verhalten an Abzweigungen innerhalb der Hierarchie wird durch solche, an Bedingungen geknüpfte Regeln gesteuert. Daneben ist auch die Modellierung von Zeitkonzepten denkbar. So kann z.B. spezifiziert werden, ob bestimmte Ziele untereinander nur sequentiell oder auch parallel ausgeführt werden können. Da die Modellierung von Zeitkonzepten eine zentrale Stellung in dieser Arbeit einnimmt, wird hierauf separat in Abschnitt 3.3 eingegangen.

Die Durchführung einer Hierarchischen Aufgabenanalyse geschieht i.d.R. in sieben Schritten (Diaper und Stanton, 2004, S. 73), die jeweils, je nach Ziel der Analyse, stärker oder schwächer berücksichtigt werden können:

1. Stelle die Absichten der Analyse fest.
2. Definiere mit allen Interessenvertretern einheitliche Aufgabenziele und Kriterien-Maße.
3. Identifiziere verfügbare Quellen für die Sammlung von Aufgabeninformationen und wähle Mittel zur Datenakquise aus.
4. Erwerbe die Daten und zerlege diese. Halte diese Zerlegung in Form eines Diagramms oder einer Tabelle fest.
5. Überprüfe die Validität der Zerlegung zusammen mit den Interessenvertretern.
6. Ermittle signifikante Operationen angesichts der Absichten der Analyse.
7. Erstelle und, sofern möglich teste Hypothesen bzgl. Faktoren, die das Erlernen und die Leistung beeinflussen.

Die Absichten der Analyse haben Einfluss auf die Art der Datenerhebung, die Ergebnisse und die Granularität der Zielhierarchie. Sind die Absichten definiert, so können die groben Ziele mit allen Interessenvertretern gemeinsam formuliert werden. Auch Leistungsindikatoren und -kriterien sollten geklärt werden. Dabei ist z.B. zu diskutieren, welches objektive Kriterium belegt, dass ein Ziel erreicht

wurde oder welche Auswirkungen es hat, wenn ein Ziel nicht erreicht wird. Im nächsten Schritt werden Datenquellen ermittelt, anhand derer die Aufgabenanalyse durchgeführt werden soll. Das können z.B. Aufzeichnungen früherer Interaktionen (gewählte Pfade, Fehler- und Erfolgsraten), Interviews mit Systemdesignern und -experten, Nutzerobservationen oder Aufzeichnungen von Simulationen und Experimenten sein. Schritt 4 sieht die Erstellung der Zielhierarchie mit Hilfe der erworbenen Daten vor. Um aussagekräftig zu sein, muss die Zerlegung stark nutzerorientiert vorgenommen werden. Dabei gilt es zuzuordnen, was die Benutzer machen (Operationen/Ziele), warum sie dies tun (Erreichen von Zielen) und welche Auswirkungen es hat, wenn sie dies nicht korrekt machen (Bedingungen). Valide gilt eine Zerlegung, wenn alle Subziele untereinander exklusiv und zusammen vollständig sind, d.h. dass alle Subziele zusammen das entsprechende übergeordnete Ziel definieren. Notiert wird die Zielhierarchie i.d.R. in Form eines Hierarchie-Diagramms oder einer Tabelle. Auch Ablaufpläne und Haltepunkte[2] sollten in der Notation verdeutlicht werden. Im Anschluss sollte die Überprüfung der Zerlegung unter Absprache mit den Interessenvertretern vorgenommen werden. Bestenfalls geschieht dies in einem iterativen Prozess, der letztlich eine verlässliche und ausgereifte Analyse ermöglicht. Nach der Überprüfung ermittelt der Analyst signifikante Operationen bzw. Ziele unter Berücksichtigung der Analyseabsicht. Solche Operationen sind z.B. jene, deren Scheitern erhebliche negative Konsequenzen auf den gesamten Systemablauf hat. Auch Operationen, die eine hohe Arbeitsbelastung, Teamarbeit oder spezielles Wissen erfordern, können signifikante Operationen sein. Der letzte Analyseschritt besteht darin, die aufgedeckten Fehlerquellen bestimmten Fähigkeiten, Regeln oder Wissen zuzuordnen. Diese Zuordnungen helfen anschließend plausible Lösungen bzw. Verbesserungen vorzuschlagen. Sofern möglich, sollten die Vorschläge hinterher validiert werden.

Eine bekannte Evaluationsmethode, die auf der Hierarchischen Aufgabenanalyse aufbaut, ist GOMS[3]. In GOMS wird ein Aufgabenmodell anhand von Zielen, Operatoren, Methoden und Selektionsregeln repräsentiert. Ziele sind die Aufgaben, die der Nutzer durchführen möchte. Das oberste Ziel stellt die Hauptaufgabe dar und wird in Unterziele aufgebrochen, die die Teilaufgaben repräsentieren, die der Nutzer erfüllen muss um das oberste Ziel zu erreichen. So entsteht die für eine Hierarchische Aufgabenanalyse typische Aufgaben- bzw. Zielhierarchie. Die Ope-

[2] Ein Haltepunkt definiert die Ebene mit dem höchsten Detailgrad.
[3] Goals, Operators, Methods, Selection Rules

ratoren sind die detailliertesten Ziele, d.h. die Ziele der untersten Hierarchieebene, die z.B. einzelne Motoroperationen repräsentieren. Methoden sind Sequenzen von Operatoren und Unterzielen, die ausgeführt werden müssen um ein bestimmtes Ziel zu erreichen. Selektionsregeln werden benötigt, wenn mehrere Methoden bzw. Unterziele geeignet sind um ein bestimmtes Ziel zu erreichen. Die Regeln klären dann welche Methode ausgeführt bzw. welches Ziel verfolgt wird.

Das ursprüngliche CMN[4]-GOMS wurde zu einer Familie von GOMS-Verfahren weiterentwickelt. Die Verfahren der GOMS-Familie (John und Kieras, 1994) ermöglichen u.a. die Abschätzung von Ausführungs- und Lernzeiten einer Aufgabe, die in Form eines hierarchischen Aufgabenmodells organisiert wird. Die Grundidee von GOMS ist das Prinzip der Hierarchischen Aufgabenanalyse: die Interaktion zwischen Mensch und Maschine auf elementare Aktionen zu reduzieren und den Ablauf dieser in einem Modell darzustellen, um dann die Effizienz dieser Schritte zu ermitteln. Anzumerken ist hier, dass Dix, Finlay, Abowd und Beale (2004) die Aufgabenanalyse von zielorientierten kognitiven Ansätzen wie GOMS abgrenzen. Während das Ziel ersterer primär in der reinen Beschreibung von äußerlich beobachtbarem Nutzerverhalten bestehe, fokussiere z.B. GOMS stärker auf interne kognitive Prozesse des Nutzers. Laut Dix et al. (2004) ist das primäre Ziel der Aufgabenanalyse die Erstellung eines konzeptuellen Modells, welches z.B. zur Dialogstrukturierung verwendet werden kann. Mit Hilfe der Aufgabenanalyse können darüber hinaus Systemschwächen ermittelt werden, die insbesondere anhand des beobachtbaren Nutzerverhaltens erkennbar sind. Zielorientierte kognitive Ansätze nutzen die erstellte Zielhierarchie zur weiteren Analyse von z.B. Komplexität oder Erlernbarkeit, d.h., deren Ziel geht von vornherein über die Erstellung des Modells hinaus und liegt in der Ermittlung von Schwachstellen des Systems, die vor allem durch kognitive Prozesse aufgedeckt werden können.

So verhält es sich auch mit der in dieser Arbeit betrachteten, kognitiv verwurzelten Hierarchischen Aufgabenanalyse. Neben der Aufdeckung konkreter Usability-Schwächen sollen Zeitvorhersagen die Analyse unterstützen. Das Ziel der Aufgabenanalyse ist hier demnach mehr als die reine Modellerstellung. Einen Überblick über Werkzeuge zur Unterstützung und Automatisierung von Zeitanalysen gibt Kapitel 3.3.4.

Neben der Aufgabenanalyse gibt es noch viele weitere gängige Methoden, die zur

[4] Card, Moran, Newell

Erhebung von Usability-Maßen und allgemeinen Usability-Bewertung angewendet werden. Um den logischen Aufbau und Argumentationsfluss dieser Arbeit nicht zu unterbrechen wird hier jedoch nicht auf weitere Methoden eingegangen. Um dennoch die Einordnung der Aufgabenanalyse in das Gesamtfeld der Usability-Evaluation zu ermöglichen, sei an dieser Stelle auf einen umfangreichen Überblick über weitere gängige Methoden zur Usability-Evaluation in Anhang III hingewiesen.

3.3 Berücksichtigung von Zeitkonzepten bei der Aufgabenanalyse

Um die Aufgaben eines Aufgabenmodells in einem zeitlichen Ablauf anordnen zu können, ist die Definition von Zeitkonzepten erforderlich. Anschließend kann auf Basis dieser Zeitkonzepte die Berechnung von Ausführungszeiten einzelner Aufgaben oder der Gesamt-Aufgabe durchgeführt werden.

3.3.1 Definitionen verschiedener Arten von Zeitkonzepten

In diesem Abschnitt werdem einige Ansätze zur Definition von Zeitkonzepten vorgestellt.

Serielle, sequentielle und parallele Ausführung

Die zeitliche Abfolge, mit der ein Mensch Aufgaben durchführt, kann unterschiedlichen Konzepten folgen. Im Folgenden wird zwischen sequentieller, serieller und paralleler Zeitplanung unterschieden.

Sequentiell werden Aufgaben dann bearbeitet, wenn die Aktionen, die zur Bearbeitung einer Aufgabe notwendig sind, nacheinander ausgeführt werden. Die Reihenfolge der einzelnen Aktionen ist dabei strikt festgelegt. Ähnlich verhält es sich bei der seriellen Zeitplanung. Auch hier werden einzelne Aktionen nacheinander ausgeführt. Jedoch ist die Reihenfolge der einzelnen Aktionen untereinander nicht streng vorgegeben. Die Bedingung ist lediglich die nicht-simultane Ausführung einzelner Aktionen. Im Gegensatz dazu wird bei der parallelen Zeitplanung

versucht, möglichst viele Einzelaktionen gleichzeitig, d.h. nebenläufig auszuführen. Da der Mensch nur bedingt fähig ist, parallel Aktionen auszuführen und parallel Informationen zu verarbeiten, ist diese Art der Zeitplanung nicht trivial und bedarf genauerer Betrachtung.

Parallele Verarbeitung bedeutet die simultane Durchführung und Verarbeitung mehrerer Aktionen. Während Computer z.B. mit Hilfe mehrerer Prozessoren in der Lage sind, parallele Aktionen bzw. Berechnungen durchzuführen, gelingt dies dem Menschen nur bedingt (s. Abschnitt 3.3.2). Auf Grund der Einschränkungen in der menschlichen Kognitionsleistung können viele, insbesondere komplexe Aufgaben nicht nebenläufig bearbeitet werden. Gerade solche Aufgaben sind es i.d.R. jedoch, die im Rahmen von Aufgabenanalysen modelliert werden. Somit wird hier das menschliche Verhalten selten dem Begriff Parallelität gerecht, sondern eher durch den Begriff des Multitasking[5] beschrieben. Multitasking ist von streng paralleler Verarbeitung abzugrenzen, denn dabei findet meistens keine reale Parallelität statt. Der Begriff wurde ursprünglich in der Informatik verwendet. Er beschreibt das Verhalten von Prozessoren, die eine simultane Verarbeitung nur imitieren, tatsächlich aber zwischen einzelnen Aktionen wechseln oder diese ineinander verschachteln. Durch besonders kurze Abstände zwischen den einzelnen Prozessen verursachen sie beim Anwender die Illusion, gleichzeitig an mehreren Prozessen zu arbeiten. Der Grundgedanke des Multitasking war insbesondere, Wartezeiten auf Prozesse auszunutzen, um bereits andere Prozesse auszuführen und so insgesamt effizienter zu arbeiten. Die Abbildung des Begriffs auf menschliches Verhalten ist vergleichbar, jedoch wissenschaftlich noch nicht genau definiert. Es wird angenommen, dass das menschliche Gehirn ähnlich arbeitet und versucht, einzelne Aufgaben ineinander zu verschachteln (Lee und Taatgen, 2002). Bei dieser Verschachtelung[6] wird von unterschiedlichen Strategien ausgegangen, die sich an Kriterien wie z.B. Struktur, Dauer oder Prioritäten der Einzelaktionen (Adamczyk und Bailey, 2004; Brumby et al., 2009) oder der aktuellen mentalen Arbeitslast (Salvucci und Bogunovich, 2010) orientieren. Diese Strategien unterscheiden sich beispielsweise neben den Zeitpunkten der Wechsel auch in der Häufigkeit von Aufgabenwechseln und der Dauer zwischen Aufgabenwechseln, d.h. der Dauer der Unterbrechung einer Aufgabe (Monk et al., 2008; Brumby et al., 2007). Lange Unterbrechungen von Aufgaben können außerdem dazu führen, dass diese nicht

[5] dt.: Mehrprogrammbetrieb, Mehrprozessbetrieb
[6] engl.: Task Interleaving

direkt wieder aufgenommen werden können, sondern zunächst Rekonstruktionsprozesse nötig sind (Salvucci, 2010). Neben den Strategien grenzen die kognitiven Fähigkeiten die Möglichkeiten zum Multitasking ein. Außerdem beeinflussen individuelle Faktoren, wie erlerntes Wissen die Möglichkeit zum Multitasking (s. Abschnitt 3.3.2).

Es wird deutlich, dass bei Zeitanalysen von, insbesondere „parallel“ ausführbaren Aufgaben, viele, teils sehr komplexe Faktoren berücksichtigt werden können. Auf diese Faktoren wird nochmals abschließend an dieses Kapitel in Abschnitt 3.3.5 eingegangen.

13 temporale Intervall-Relationen nach James Allen

Innerhalb der groben Einteilung in seriell, sequentiell und nebenläufig können wiederum unterschiedliche Zeitbedingungen betrachtet werden. Allen (1983) stellt eine temporale Logik vor, die mit Hilfe von 13 verschiedenen temporalen Relationen alle möglichen zeitlichen Beziehungen zwischen zwei Zeitintervallen ausdrückt. Die Relationen setzen sich aus sieben verschiedenen Konzepten und der jeweils entsprechenden Inversen zusammen. Die Abbildung 3.2 veranschaulicht diese Relationen.

Die Bezeichner drücken die Bedeutung der einzelnen Relationen in englischer Sprache aus. Zu sehen sind Relationen, die sequentielle Beziehungen spezifizieren (before, after, meets, met by), sowie Relationen, die nebenläufige Abläufe beschreiben (equal, overlaps, overlapped by, during, contains, starts, started by, finishes, finished by). Die Skizzen in der o.g. Abbildung verdeutlichen die Definitionen der einzelnen Relationen.

Erweiterung der Intervall-Relationen um Punkt-Intervall- und Punkt-Punkt-Relationen

Vilain (1982) erweitert den Ansatz von Allen um Beziehungen zwischen Zeitintervallen und Zeitpunkten. Vilain leitet aus den Relationen von Allen 13 temporale Punkt-Intervall- und Punkt-Punkt-Beziehungen ab. Dabei handelt es sich um die Relationen

before, after, equals (zwischen zwei Zeitpunkten), sowie

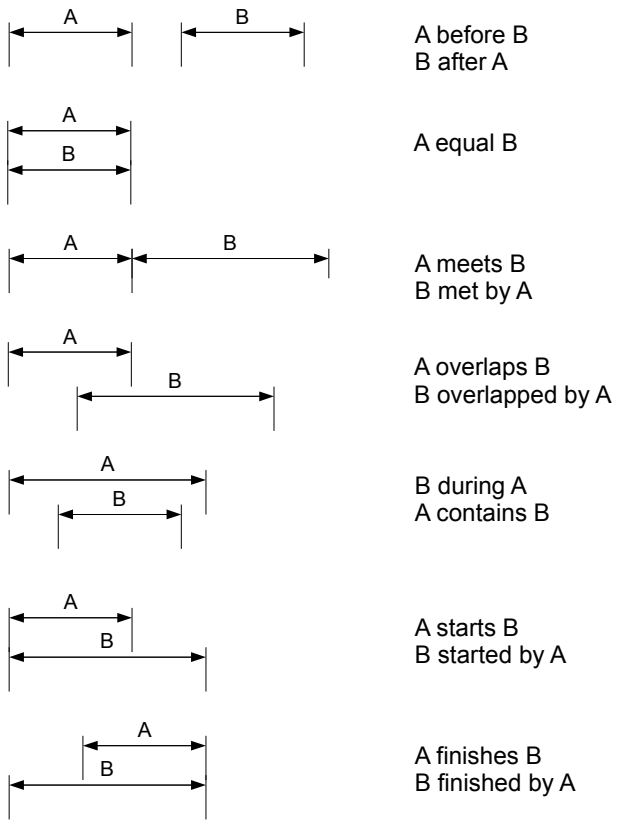


Abb. 3.2: 13 temporale Relationen nach Allen (1983)

before, after, begins, begun-by, during, contains, ends, ended-by (zwischen Zeitpunkt und -intervall).

Beide Ansätze beschreiben außerdem, welche Schlussfolgerungen aus Kombinationen mehrerer Bedingungen vorgenommen werden können. So lässt sich beispielsweise aus den Bedingungen *A before B* und *B before C* folgern, dass zudem gilt *A before C*. Darüber hinaus stellen beide Ansätze Algorithmen vor, die diese Folgerungen, d.h. die Weitergabe von Bedingungen berechnen.

Zusammenfassung einzelner Aktionen zu Task-Units

Salvucci und Bogunovich (2010) stellten in Experimenten einen Zusammenhang zwischen Aufgabenunterbrechungen und mentaler Arbeitslast fest. Nutzer tendierten stark dazu, die Unterbrechung einer Hauptaufgabe durch eine Nebenaufgabe, deren Durchführung verzögert werden kann, solange zu verzögern, bis die men-

tale Arbeitslast, die durch die Hauptaufgabe entstand, minimiert wurde. Diese Beobachtungen erscheinen plausibel, wenn folgende weitere Erkenntnisse früherer Untersuchungen hinzugezogen werden. Untersuchungen ergaben, dass erzwungene Unterbrechungen während Phasen hoher mentaler Arbeitslast störender, d.h. ineffizienter, waren, als in Phasen geringerer Arbeitslast (Adamczyk und Bailey, 2004; Bailey und Iqbal, 2008; McFarlane, 1999). Darüber hinaus wurde erforscht, dass Personen die Zeitpunkte zwischen des Abrufens einzelner Chunks dazu nutzen, zwischen zwei nebenläufigen Aufgaben zu wechseln. Eine Unterbrechung findet jedoch nicht während des Zugriffs auf einen Chunk statt (Salvucci, 2005).

Der Mensch tendiert also dazu, sofern ihm die Wahl gelassen wird, selbstständig zu Zeitpunkten zu unterbrechen bzw. zwischen Aufgaben zu wechseln, an denen seine Arbeitslast gering ist. Diese Zeitpunkte sind typischerweise an sogenannten „Subtask Boundaries“[7] zu finden, die die Grenzen zwischen zwei benachbarten Chunks innerhalb einer Aufgabe darstellen (Brumby et al., 2007). Da der Mensch die Ausführung eines einzelnen Chunks nicht unterbrechen würde, würde auch keine Verschachtelung innerhalb von Chunks im Rahmen von Zeitanalysen realistisch sein. Auf Grund dieser Erkenntnisse erscheint es sinnvoll, die Unterteilung in Chunks auch in Aufgabenmodellen, die zur Zeitanalyse herangezogen werden, zu berücksichtigen. In einem hierarchischen Aufgabenmodell können mehrere feingranulare Aktionen zu sogenannten „Task-Units“[8] zusammengefasst werden. Diese Task-Units fassen Aktionen zusammen, die aus Sicht menschlicher Kognition nur als Einheit ausgeführt bzw. repräsentiert sinnvoll sind. Wie bereits erwähnt, besteht eine solche Einheit häufig aus einem Gedächtnis-Chunk (Bailey und Iqbal, 2008; Janssen et al., 2010). Ein solcher Chunk kann z.B. eine Folge von Ziffern einer Telefonnummer sein, die als eine Einheit im Gedächtnis abgelegt wird (Brumby et al., 2007; Janssen et al., 2010). Ein anderes Beispiel für eine Task-Unit ist die kognitive Vorbereitung auf eine Motoraktion und die anschließende Ausführung der Motoraktion selbst.

Auch wenn feststeht, dass der Mensch einzelne Aktionen zu solchen Sinneinheiten zusammenfasst, so ist die Frage, wie er diese Zuteilung konkret vornimmt, nicht eindeutig geklärt. Die Festlegung, welche Aktionen eine Task-Unit oder einen Chunk beschreiben, bleibt eine nicht eindeutig lösbare Herausforderung für den Analysten (Janssen et al., 2010).

[7] dt.: Unteraufgaben-Grenzen
[8] dt.: Aufgabeneinheiten

Für eine Zeitanalyse bedeutet die Modellierung mit Task-Units letztlich, dass im Rahmen paralleler Ausführung Aktionen innerhalb einer Task-Unit nicht mit anderen Aktionen verschachtelt werden dürfen. Alle Aktionen einer Task-Unit müssen zunächst komplett ausgeführt sein, bevor zu einer anderen Aufgabe bzw. anderen Aktionen gewechselt wird oder eine neue Aufgabe begonnen wird. In dieser Arbeit wird das Konzept der Task-Units zwar berücksichtigt, deren explizite Modellierung bzw. Definition entfällt jedoch. Task-Units bündeln mehrere kleinste Aktionen, d.h. Operatoren zu einer untrennbaren Einheit. Diese Operatoren sind bereits einem in der Hierarchie eine Ebene höher platzierten Oberziel zugeordnet. In Anlehnung an das Task-Unit Konzept wird in dieser Arbeit die vereinfachte Annahme getroffen, dass zwischen einzelnen zu einem Oberziel zugeordneten Operatoren ohnehin keine Verschachtelung stattfindet. U.a. aus diesem Grund wird im Verlauf dieser Arbeit ausschließlich die explizite Definition von Zeitkonzepten zwischen Unterzielen verfolgt (s. Kapitel 5.2.3) und Operatoren werden stets sequentiell angeordnet.

In Kapitel 5.2.2 wird auf die Relevanz der vorgestellten Konzepte bzgl. dieser Arbeit eingegangen.

3.3.2 Psychologische Grundlagen der menschlichen Aufmerksamkeit und Leistung

Die Berücksichtigung von Zeitkonzepten bei der Durchführung von Aufgaben erfordert einen Einblick in die kognitive Leistungsfähigkeit des Menschens. Denn neben Bedingungen, die sich aus der Logik der modellierten Aufgabe ergeben, müssen auch Bedingungen berücksichtigt werden, die sich aus der Funktionsweise der menschlichen Kognition ergeben. Beide Arten von Bedingungen können direkten Einfluss auf die zeitliche Anordnung von Aktionen haben.

Während die serielle Durchführung von Aufgaben kein Problem für den Menschen darstellt, kann es bei der parallelen Ausführung Schwierigkeiten geben. Dies ist auf Einschränkungen in der menschlichen Kognitionsleistung zurückzuführen. Der Psychologe John Anderson (2001) spricht von einem „seriellen Flaschenhals“ in der menschlichen Informationsverarbeitung. Dieser tritt auf, wenn der Mensch nicht mehr fähig ist, alle Informationen parallel zu verarbeiten. Wann genau dies der

Fall ist, ist bis heute eine ungeklärte Frage der Kognitionspsychologie. Allerdings gibt es Erklärungsansätze, die gehäuft Zustimmung erhalten.

Card, Moran und Newell (1983) beschreiben die menschliche Informationsverarbeitung anhand der Multi-Prozessor Theorie *Model Human Processor* (MHP). Dieses stark vereinfachte Modell geht davon aus, dass ein Mensch mit Hilfe von Speichern und verschiedenen Prozessoren arbeitet, die jeweils entweder wahrnehmend, kognitiv oder motorisch sind (s. Abbildung 3.3).

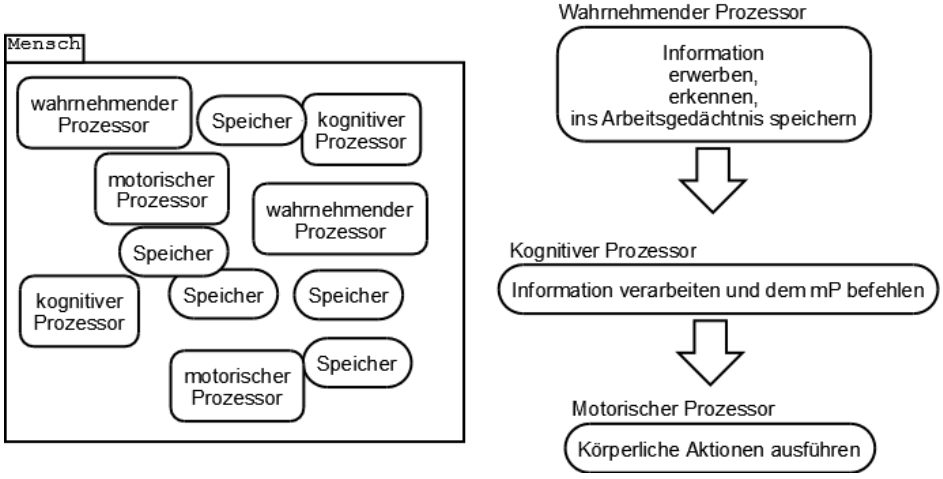


Abb. 3.3: Prinzip des *Model Human Processor*

Ein kompletter Informationsfluss funktioniert folgendermaßen: Ein wahrnehmender Prozessor nimmt Information auf, erkennt sie und speichert sie dann im Arbeitsgedächtnis. Ein kognitiver Prozessor verarbeitet diese Information anschließend und erteilt einem motorischen Prozessor ggfs. Befehle. Der motorische Prozessor sorgt letztlich dafür, dass entsprechend den Befehlen körperliche Aktionen ausgeführt werden. In sich arbeiten diese Prozessoren ausschließlich seriell. Da die einzelnen Prozessoren jedoch unabhängig voneinander agieren können, ist es möglich, dass sie zueinander parallel arbeiten. So kann ein Mensch beispielsweise eine Straße entlang gehen und dabei einer Unterhaltung zuhören. Motorische und auditive (= spezielle wahrnehmende) Prozessoren arbeiten dabei parallel zueinander. Auch verschiedene motorische Prozessoren können parallel zueinander arbeiten. Es ist z.B. kein Problem, zu gehen und dabei gleichzeitig Kaugummi zu kauen, da dabei unterschiedliche motorische Systeme angesprochen werden. Probleme treten jedoch auf, wenn mehrere verschiedene Aktionen durch dasselbe, z.B. motorische,

System gesteuert werden: z.B. fällt es schwer, sich mit der linken Hand an den Kopf zu tippen und gleichzeitig mit der rechten Hand den Bauch zu reiben.

Trotz dieser Einschränkungen ist es dem Menschen möglich, durch gezieltes Üben bestimmter Aufgaben die Auswirkung der Einschränkungen zu mindern. Gelingt es dem Menschen dadurch, eine komplexe Aufgabe zu automatisieren, d.h. dass kein oder kaum Denkaufwand mehr zur Durchführung nötig ist, so kann dies zu erhöhter paralleler Leistungsfähigkeit führen. Ziel des Übens, aus kognitiver Sicht, ist die Umwandlung von deklarativem zu prozeduralem Wissen. Prozedurales Wissen erfordert wesentlich weniger kognitive Leistung, da das gesamte Verhalten bereits in Form von Prozeduren gespeichert ist, die „einfach“ ausgeführt werden. Deklaratives Wissen hingegen ist noch nicht so stark verinnerlicht und muss erst ausgewertet werden, um ein Problem bearbeiten zu können. Folglich sind dabei verstärkt kognitive Prozessoren aktiv (Lee und Taatgen, 2002; Cutrell et al., 2001).

Autofahren ist ein häufig auftretendes Beispiel für eine komplexe Aufgabe, die während des Übens immer weiter automatisiert wird. So gelingt es vielen Menschen z.B. sich während des Autofahrens, bei dem wahrnehmende, kognitive und motorische Prozessoren beteiligt sind, mit dem Beifahrer zu unterhalten, was wiederum mehrere Prozessoren beansprucht. Die dabei entstehende hohe kognitive Last kann allerdings dazu führen, dass der Mensch mehr Fehler macht und die Aufgaben ineffizienter durchführt. Denn sobald die realen Bedingungen von denen in der automatisierten Situation abweichen, müssen kognitive Prozessoren zur Problemlösung in Anspruch genommen werden. Sind diese allerdings gerade durch eine Zweitaufgabe belegt, besteht zunächst ein Ressourcenkonflikt, der wiederum u.a. zu einer Reaktionsverzögerung und letztlich zum Unfall führen kann. Die zentrale Kognition ist, wie auch in diesem Beispiel, oft der Flaschenhals. Denn der Mensch ist zwar fähig mehrere motorische Aktionen gleichzeitig durchzuführen oder mehrere Informationen gleichzeitig wahrzunehmen, er kann jedoch nicht gleichzeitig über mehrere Sachverhalte nachdenken (Anderson, 2001).

In Abschnitt 3.3.1 wurde bereits das Prinzip der Verschachtelung angesprochen. Auf Grund der kognitiven Einschränkungen wird davon ausgegangen, dass Menschen zwei komplexe Aufgaben, die beide die zentrale Kognition beanspruchen, ineinander verschachteln, um sie parallel ausführen zu können (Anderson, 2001). Die Verschachtelung erfordert, dass die beiden Aufgaben mehrmals unterbrochen

werden. Cutrell, Czerwinski und Horvitz (2001) beschreiben, dass der Geübtheitsgrad einer Aufgabe Auswirkungen auf die Störkraft der Unterbrechungen hat. Je geübter, d.h. automatisierter ein Mensch in der Ausführung einer Aufgabe ist, desto weniger störend wirken sich Unterbrechungen dieser Aufgabe auf die Performanz der gesamten Aufgabe aus.

Die in diesem Abschnitt vorgestellten psychologischen Aspekte dienen als Hintergrundwissen. Die einzelnen Einschränkungen und Einflussfaktoren können in Aufgabenanalysen berücksichtigt werden, um die Analysen – aus psychologischer Sicht – möglichst realistisch durchzuführen. Der Fokus dieser Arbeit liegt jedoch nicht auf solchen, psychologischen Aspekten sondern widmet sich verstärkt Aspekten der Aufgabenlogik, wie sie in den Abschnitten 3.3.1 und 3.3.3 vorgestellt werden.

3.3.3 Ansätze zur Modellierung von Zeitkonzepten in Aufgabenmodellen

Es gibt bereits einige Ansätze, die die konkrete Modellierung von Zeitkonzepten in Aufgabenmodellen beschreiben. Im Folgenden werden einige wichtige dieser unterschiedlichen Ansätze erläutert.

Plans

Annett beschreibt im Rahmen der Hierarchischen Aufgabenanalyse sogenannte „Plans“ (Diaper und Stanton, 2004), die als grobe Ablaufpläne fungieren. Diese Pläne definieren u.a., in welcher Reihenfolge Unterziele auszuführen sind. Dazu werden die Konstrukte

sequentiell (>), *entweder/oder* (/), *parallel* (+) und *beliebig* (:)

verwendet. Diese Einteilung ist sehr grob und wurde in anderen Ansätzen in Anlehnung an die von Allen (1983) vorgestellten Zeitrelationen erweitert. Einige dieser Ansätze werden im Folgenden vorgestellt.

User Action Notation

Die User Action Notation (UAN) bietet, wie auch die Hierarchische Aufgabenanalyse die Möglichkeit, die Interaktion zwischen Mensch und Computer, d.h. deren Verhalten, dargestellt durch Aufgaben und Beziehungen zueinander, zu modellieren. Die Interaktion wird jedoch nicht graphisch in Form hierarchischer Netzwerke dargestellt, sondern geschieht textuell mit Hilfe von Tabellen. In den Tabellen drücken die verschiedenen Zeilen und Spalten die Hierarchie und den zeitlichen Ablauf aus. Für den Fokus dieser Arbeit sind die Relationen interessant, die das temporale Verhalten bei der Durchführung der Aufgaben beschreiben. Hartson und Gray (1992) stellen acht temporale Relationen vor:

sequence, waiting, repeating disjunction, order independence, interruptibility/one-way interleavability, mutual interleavability und *concurrency.*

Durch *waiting* kann z.B. eine Verzögerungszeit zwischen der sequentiellen Durchführung zweier Aufgaben definiert werden. Beispielsweise drückt $A(t > 4)B$ aus, dass Aufgabe B ausgeführt wird, nachdem mehr als vier Zeiteinheiten nach der Durchführung von Aufgabe A vergangen sind. *Interruptibility* und *mutual interleavability* beschreiben die Möglichkeit zweier Aufgaben, sich einseitig zu unterbrechen bzw. gegenseitig zu unterbrechen und ineinander zu verschachteln, während *concurrency* lediglich ausdrückt, dass zwei Aufgaben unabhängig voneinander sind und deshalb nebenläufig ausgeführt werden können.

UAN definiert die Ausführungsreihenfolge der einzelnen Aufgaben nur grob, d.h. die genaue Ordnung der Teilaufgaben untereinander wird weitestgehend nicht definiert. Kritisiert wird zudem die unzureichende Darstellung von Kontrollstrukturen in der Tabellenform (Gray et al., 1994).

eXtended User Action Notation

Um die Mächtigkeit der vorgestellten UAN-Notation hinsichtlich temporaler Relationen zu erweitern, wurde die erweiterte UAN-Notation XUAN[9] entwickelt. Neben der Erweiterung der tabellarischen Darstellung (s. Gray et al., 1994) wurden zusätzliche temporale Bedingungen eingeführt. Diese bieten u.a. die Möglichkeit, die Dauer einer Aufgabe festzulegen. Dazu werden Start- und Endzeitpunkt definiert

[9] eXtended UAN

und daraus die Dauer errechnet. Darüber hinaus erlaubt die Definition von Start- und Endpunkten in Verbindung mit Vergleichsrelationen die präzise Modellierung des Aufgabenablaufs. Durch

$start(A1) > start(A2)$

wird beispielsweise ausgedrückt, dass die Aufgabe *A1* startet, nachdem die Aufgabe *A2* gestartet ist (Lacaze und Palanque, 2004). So können nicht nur temporale Aussagen über Aufgaben als Ganzes getroffen werden, sondern auch separat über deren Start- und Endpunkte. Als weitere Bedingungen werden die Konzepte *before, after* und *during* eingeführt, die äquivalent zu den gleichnamigen Relationen von Allen (1983) definiert sind.

Die Notation XUAN stellt eine ausdrucksstarke Erweiterung von UAN dar. Sie ermöglicht die explizite Modellierung mengenbezogener, temporaler Aspekte in Aufgabenmodellen und geht damit deutlich über die Aussagekraft der bisher vorgestellten Ansätze hinaus.

Concur Task Trees

Paternò, Mancini und Meniconi (1997) stellen die Notation Concur Task Trees (CTT) vor, die es ermöglicht Aufgabenmodelle mit mehreren temporalen Relationen anzureichern. CTT wurde insbesondere hinsichtlich leichter Verständlichkeit und leichter Nutzbarkeit für die Erstellung von umfangreichen Aufgabenmodellen konzipiert. Hinter CTT verbirgt sich eine graphische Notation, die Aufgabenmodelle in Form Baum-ähnlicher hierarchischer Strukturen darstellt. Die Abbildung 3.4 zeigt ein CTT-Modell an einem Beispiel aus der Luftfahrt. Die Baumknoten repräsentieren die einzelnen Teilaufgaben einer Hauptaufgabe, die wiederum durch die Baumwurzel dargestellt wird. Temporale Relationen werden in Form von unterschiedlichen Zeichenkombinationen direkt zwischen Knoten notiert. Die unterschiedlichen Symbole in den einzelnen Knoten repräsentieren verschiedene Aufgabentypen: eine Wolke steht für eine abstrakte Aufgabe, während eine Person eine kognitive Aufgabe darstellt. Daneben gibt es noch System- (dargestellt durch einen Computer) und Interaktionsaufgaben (symbolisiert durch eine Person vor einem Computer). Um evtl. Mehrdeutigkeiten auszuschließen, die durch diese Notation auftreten können, wird ggfs. auf Hilfsknoten zurückgegriffen, die keine Aufgaben repräsentieren, sondern lediglich den Ablauf spezifizieren.

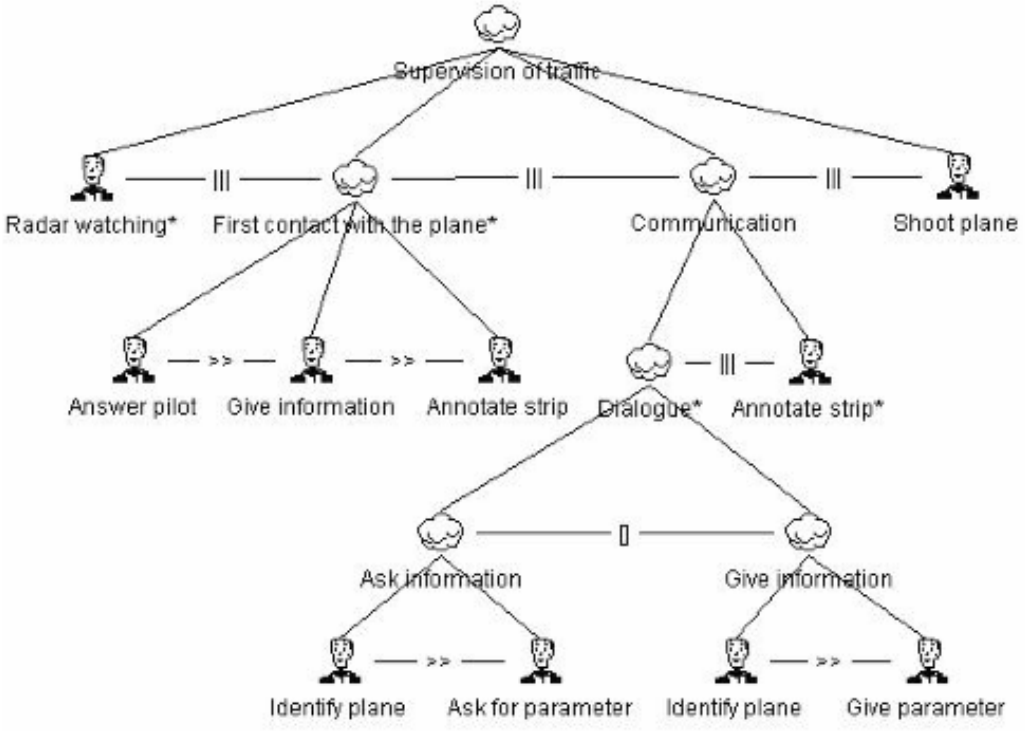


Abb. 3.4: Ausschnitt eines Aufgabenmodells in CTT-Notation (Quelle: Lacaze und Palanque, 2004)

Aufbauend auf den Operatoren der internationalen Standard-Notation LOTOS (Bolognesi und Brinksma, 1987) und der vorgestellten Notation UAN verwendet CTT elf Spezifikationen, die den Ablauf des Aufgabenmodells, d.h. die Ausführungsreihenfolge der einzelnen Knoten (bzw. Aufgaben) untereinander einschränken. Dabei ist es möglich, Nebenläufigkeit etwas genauer zu beschreiben, indem z.B. die Spezifikation *synchronization* (Symbol: $A1|[]|A2$) verwendet wird. Allerdings drückt diese Angabe lediglich aus, dass zwei Aufgaben während ihrer Durchführung einige Aktionen synchronisieren müssen um Informationen auszutauschen. An welchen Stellen genau diese Synchronisation stattfindet, wird nicht spezifiziert. Ähnlich undifferenziert ist die Definition weiterer verwandter Relationen wie die unabhängige Verschachtelung (Symbol: $A1|||A2$) oder die Erlaubnis zur Unterbrechung anderer Aufgaben (Symbol: $A1| > A2$). Neben der groben Beschreibung paralleler Durchführung bietet CTT weitere Relationen, die beispielsweise sequentielle Freigaben von Aufgaben (Symbol: $A1 >> A2$), dynamische Deaktivierung von Aufgaben (Symbol: $A1[> A2$), Wiederholung (Symbol: $A1*$), optionale Aufgaben (Symbol: $[A1]$) oder die Auswahl zwischen mehreren Aufgaben (Symbol:

$A1[]A2$) spezifizieren.

Zusammenfassend ist festzuhalten, dass CTT zwar die Modellierung mehrerer zeitlicher Relationen ermöglicht, diese teilweise jedoch nur auf hoher Hierarchieebene stattfindet. Konkret bedeutet dies, dass lediglich die Ordnungs-Beziehung zweier Aufgaben zueinander spezifiziert wird, nicht aber, wie die einzelnen Teilaktionen der Aufgaben zueinander durchgeführt werden. Dies betrifft insbesondere die Modellierung parallel durchführbarer Aufgaben. In dem späteren Ansatz ART[10] (Kahn et al., 2007) wird diese Schwäche aufgegriffen und ein Konzept vorgestellt, das CTT um einige der von Allen (1983) vorgestellten, Parallelität betreffenden Zeitkonzepte erweitert. Durch die Angabe zusätzlicher temporaler Abhängigkeiten zwischen parallel ausführbaren Aktivitäten ist eine präzisere Modellierung des zeitlichen Ablaufs möglich. Im Vergleich mit XUAN werden von Kahn, Klug und Flentge (2007) zwar noch weitere Relationen von Allen umgesetzt, die Modellierung ist jedoch in Zeitintervallen, d.h. vollständigen Aufgaben vorgesehen. XUAN setzt zwar weniger Relationen von Allen um, dafür bietet die Modellierung mit Anfangs- und Endzeitpunkten von Aufgaben die Spezifikation zusätzlicher, präziserer temporaler Bedingungen.

GOMS

GOMS wurde bereits in Kapitel 3.2 als Methode zur Modellierung hierarchischer Aufgaben vorgestellt. Das Aufgabenmodell wird typischerweise in textueller Form dargestellt, bei der unterschiedliche Hierarchieebenen in der Zielstruktur durch unterschiedliche Zeileneinrückungen repräsentiert werden. Die Modellierung des zeitlichen Ausführungsablaufs ist einfach gehalten. Die Reihenfolge, in der die einzelnen Aktionen eines Ziels untereinander aufgeführt sind, gibt auch die Ausführungsreihenfolge wieder. Diese ist somit sequentiell. Auch die einzelnen Unterziele, die zur Erreichung des entsprechenden Oberziels erreicht werden müssen, werden sequentiell nacheinander ausgeführt, sofern dies nicht durch eine Selektionsregel anders spezifiziert wird. Die Selektionsregeln können definieren, dass Ziele nur unter bestimmten Bedingungen ausgeführt werden sollen. Diese Bedingungen sind i.d.R. einfache Zustandsabfragen und ermöglichen so in Verbindung mit einer Selektionsregel die Modellierung einer Option. Zwischen den einzelnen Zielen und

[10] Activity Relation Trees

Operatoren werden keine zusätzlichen, zeitlichen Abhängigkeiten definiert.

CPM-GOMS

CPM[11]-GOMS ist ein Verfahren zur Hierarchischen Aufgabenanalyse, das mit den Prinzipien des Model Human Processor (s. Abschnitt 3.3.2) kombiniert wird. Modellierte Aufgaben werden sehr detailliert hierarchisch zerlegt, die feingranularen Operatoren mit Ausführungszeiten versehen und anschließend den einzelnen Prozessoren zugeordnet. Die Darstellung von CPM-GOMS-Modellen findet üblicherweise in sog. PERT[12]-Diagrammen (s. Abbildung 3.5) statt, die den zeitlichen Ablauf und die Zuteilung der Operatoren zu den einzelnen Ressourcen veranschaulichen.

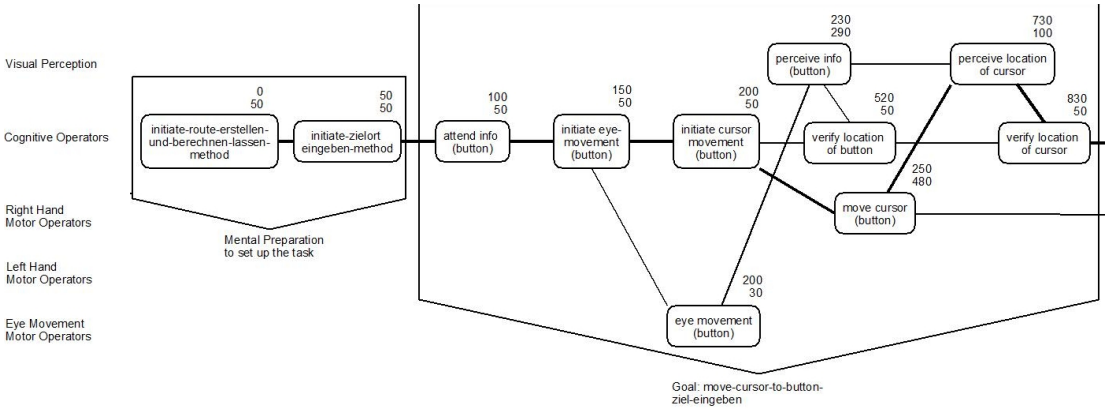


Abb. 3.5: Ausschnitt aus einem CPM-GOMS-Modell

Die durch Rechtecke dargestellten Operatoren sind teilweise über Linien verbunden, die sequentielle Abhängigkeiten repräsentieren. Operatoren, die unterschiedlichen Prozessoren zugeteilt sind, können unter Berücksichtigung sequentieller Abhängigkeiten parallel ausgeführt werden. Fünfeckige Umrandungen fassen eine Operatorsequenz zu möglichen Teilzielen zusammen. Die untere Zahl über den Rechtecken gibt die Ausführungsdauer des entsprechenden Operators, die obere Zahl die Gesamtausführungsdauer bis zu dem Punkt in Millisekunden wieder. Um die totale Bearbeitungszeit einer Aufgabe bzw. Sequenz von Operatoren zu berechnen, wird der kritische Pfad ermittelt. Auf dem kritischen Pfad eines solchen Diagramms liegen alle Operatoren, die für die Gesamt-Ausführungszeit der

[11] Cognitive-Perceptual-Motor oder Critical-Path-Method
[12] Program Evaluation and Review Technique

Aufgabe verantwortlich sind. Dies sind jene Operatoren mit den jeweils längsten Ausführungszeiten innerhalb einer Sequenz. Der kritische Pfad liefert folglich die Vorhersage der totalen Bearbeitungszeit der modellierten Aufgabe, welche am letzten Objekt des kritischen Pfades abgelesen wird (John und Kieras, 1994; John, 1990). Der kritische Pfad ist in der Abbildung mittels einer fett gedruckten Linie markiert.

In CPM-GOMS werden Sequenzen und parallele Ausführung explizit modelliert. Allerdings ist innerhalb dieser groben Zeitkonzepte keine genauere Spezifikation von zeitlichen Abhängigkeiten zwischen den einzelnen Operatoren vorgesehen. Auch Selektionsregeln, die in CMN-GOMS explizit modelliert werden, werden in CPM-GOMS nicht angegeben. Dies geschieht nur indirekt, indem der Analyst die zutreffende Methode selbst zuvor auswählt und anschließend in CPM-GOMS modelliert (John und Kieras, 1994). Die explizite Modellierung einer Option, wie in CMN-GOMS, ist somit nicht gegeben.

Inwiefern die hier vorgestellten Modellierungsansätze Einfluss auf das entwickelte Konzept dieser Arbeit nehmen, wird in Kapitel 5.2.2 erläutert.

3.3.4 Werkzeuge zur Zeitanalyse

Es gibt bereits einige Software-Werkzeuge, die die Erstellung von und zeitliche Analyse mittels Aufgabenmodellen unterstützen. In den folgenden Abschnitten werden einige dieser Werkzeuge beispielhaft vorgestellt.

Werkzeuge, die ausschließlich serielle Zeitanalysen unterstützen

GOMSED[13] (Wandmacher, 2002), CogTool[14] (Carnegie Mellon University, 2010) und GLEAN[15] (Kieras, 2006) eignen sich beispielsweise zum Erstellen und Auswerten sequentieller GOMS-Modelle. Während CogTool keine Hierarchien modelliert, sondern einzelne Operationen zur Durchführung einer Aufgabe sequentiell

[13] GOMS-Editor
[14] Cognitive Tool
[15] dt.: erkunden, auflesen

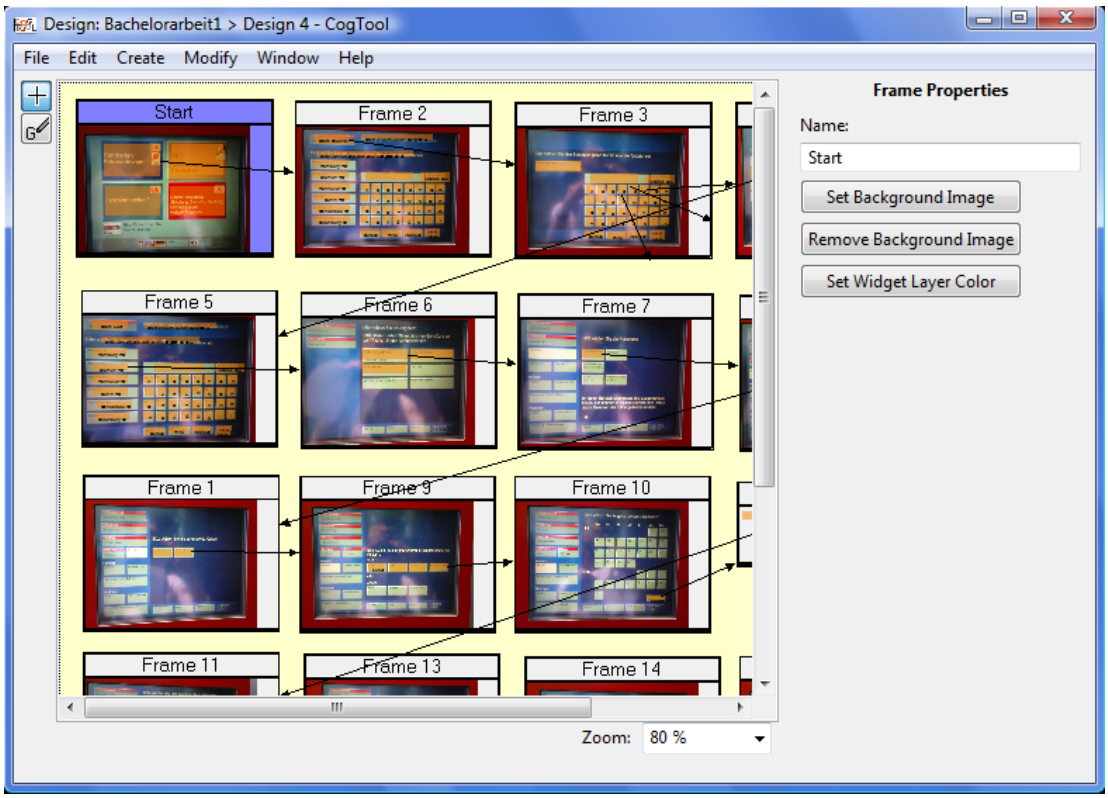


Abb. 3.6: Aufgabenmodellierung mit CogTool

auflistet, werden mit GOMSED hierarchische Aufgabenmodelle modelliert. Die Abbildung 3.6 zeigt den Ausschnitt eines Aufgabenmodells in CogTool. Mit CogTool wird die zu modellierende Benutzungsschnittstelle graphisch rekonstruiert. Die einzelnen Bildschirme (Frames) sind durch Rechtecke repräsentiert, die u.a. wiederum Rechtecke zur Modellierung von Schaltflächen beinhalten. Die einzelnen Frames werden nacheinander in einem sogenannten „Storyboard“ angeordnet. Linien zwischen modellierten Schaltflächen und Frames definieren die Abfolge zwischen einzelnen Aktionen, wie z.B. Tastendrücken. Zu sehen ist, dass Linien direkt von Schaltflächen ausgehen und zu einem neuen Frame führen. Auch mehrere Linien aus einem Frame können zu verschiedenen anderen Frames führen. Diese Modellierung beschreibt, was passiert, wenn ein Nutzer eine Taste drückt: Es erscheint entweder ein neuer Frame oder der aktuelle Frame bleibt sichtbar. Jede für das Aufgabenmodell wichtige Handlungsmöglichkeit wird in dieser Form modelliert, sodass eine geordnete Abfolge definiert ist, die sequentiell ausgeführt werden kann.

GLEAN ist im Gegensatz zu den beiden anderen Werkzeugen eine reine Konso-

lenanwendung, die zur Aufgabenanalyse auf zuvor definierte Zielhierarchien zurückgreift. Alle vorgestellten Werkzeuge können Ausführungszeiten zuvor modellierter Aufgaben berechnen, die auf sequentiellen Zeitkonzepten beruhen. GOMSED und GLEAN bieten zusätzlich die Möglichkeit, Lernzeiten zu ermitteln. Während CogTool aktiv weiterentwickelt wird und zukünftig auch explorierendes Verhalten von Nutzern vorhersagen können soll, wurde die Entwicklung von GOMSED bereits 1998 eingestellt. Die letzte Version von GLEAN wurde 2006 vorgestellt.

Werkzeuge, die neben seriellen auch parallele Zeitanalysen unterstützen

Bei der Analyse parallel durchführbarer Aufgaben, basierend auf der Methode CPM-GOMS, kann das ursprünglich zur kognitiven Modellierung verwendete Werkzeug APEX (NASA, 2006b) unterstützen. APEX ist eine kognitive Architektur, die von der NASA konstruiert wurde, um die Entwicklung intelligenter, autonomer Agenten einfacher und kostengünstiger zu gestalten. Wie auch MHP berücksichtigt sie das Prinzip der Informationsverarbeitung mit Hilfe unterschiedlicher Prozessoren. Dadurch können die CPM-GOMS-Konzepte direkt auf die von APEX abgebildet, d.h. CPM-GOMS-Modelle in APEX implementiert werden. Anhand der CPM-GOMS-Modelle kann APEX die Ausführungszeiten für kurze Routineaufgaben, unter Berücksichtigung verschiedener Zeitkonzepte wie seriell, parallel oder priorisiert berechnen. Beim parallelen Konzept verschachtelt APEX die einzelnen Operatoren automatisch, unter Berücksichtigung drei verschiedener Arten von Bedingungen (Freed et al., 2003):

Logical Erfordert der logische Aufgabenablauf, dass bestimmte Vorbedingungen, wie die Beendigung einer Aufgabe, erfüllt sein müssen, um eine andere Aufgabe durchführen zu können, wird eine logische Bedingung definiert. Diese Bedingung spezifiziert folglich explizit sequentielle Abläufe.

Unary resource Die menschliche Ressource (z.B. linke Hand), die ein Operator (s. Kapitel 3.2) beansprucht, wird explizit mit Hilfe der *unary resource* Bedingung definiert. Benötigen zwei Operatoren dieselbe Ressource, die weder simultan benutzbar noch aufteilbar ist, ist eine nebenläufige Ausführung der beiden Operatoren nicht möglich und wird durch die Angabe dieser Bedingung verhindert. Die einzelnen Operatoren werden dann sequentiell, ggfs.

anhand einer Rangfolgeregel ausgeführt.

Slack exclusion Soll verhindert werden, dass ein Operator eine „Schlupfzeit“ (Pufferzeit) zur Ausführung nutzt, obwohl eine Verschachtelung erlaubt und bzgl. der Operatordauer möglich wäre, wird eine *slack exclusion*-Bedingung angegeben. Diese Bedingung schließt aus, dass der Operator die Schlupfzeit nutzt und wird z.B. eingesetzt, um aufeinander aufbauende bzw. sinngemäß zusammengehörende Operatoren nie voneinander trennen zu können. Ein Beispiel ist die kognitive Vorbereitung auf eine Motoraktion, auf die unmittelbar die entsprechende Motoraktion folgen muss, damit die Modellierung realistisch ist.

APEX ist frei verfügbar und steht aktuell für verschiedene Plattformen in der Version 3.0 zum Download (NASA, 2006a) bereit.

AMBOSS[16] (Giese et al., 2008) ist eine in Java implementierte Modellierungsumgebung, die speziell auf die Modellierung und Simulation von Aufgabenmodellen aus dem sicherheitskritischen, soziotechnischen Bereich ausgelegt ist. Die Modellierung berücksichtigt deshalb Aspekte wie Zeitverhalten, räumliche Informationen und Kommunikation. Das Zeitverhalten kann durch fünf verschiedene temporale Relationen spezifiziert werden:

SEQ Diese Relation drückt eine sequentielle Beziehung aus, d.h., alle Unteraufgaben werden in einer festen Reihenfolge nacheinander ausgeführt. Diese Reihenfolge ist durch die Anordnung der Aufgaben in der Modellierung spezifiziert und wird von links nach rechts interpretiert.

SER Diese serielle Relation drückt aus, dass alle Unteraufgaben in einer willkürlichen Reihenfolge nacheinander ausgeführt werden.

PAR Die unabhängige, nebenläufige Ausführung von Unteraufgaben wird durch diese Relation definiert.

SIM Durch diese Relation wird eine bedingte nebenläufige Ausführung in der Form spezifiziert, dass alle Unteraufgaben gestartet sein müssen, bevor eine Unteraufgabe enden kann.

[16] Aufgabenmodellierung zur Bedienung von sicherheitskritischen Systemen

ALT Diese Relation beschreibt eine exklusive Alternative, d.h., dass genau eine Unteraufgabe ausgeführt wird.

Diese Relationen beziehen sich ausschließlich allgemein auf alle Unteraufgaben einer bestimmten Aufgabe, d.h., es können, außer durch die Relation SEQ, keine präzisen Relationen zwischen Operatoren festgelegt werden. Wie genau die Aufgaben auf der untersten Modellierungsebene, d.h. die Operatoren zueinander ausgeführt werden sollen, wird nicht näher modelliert.

Zusätzlich zu den o.g. Relationen bietet AMBOSS die Möglichkeit, Vorbedingungen zu definieren und bei der Simulation, d.h. auch bei der zeitlichen Anordnung der Unteraufgaben, zu berücksichtigen. Es können u.a. *Message preconditions* verwendet werden, die ausdrücken, dass eine Aufgabe erst dann ausgeführt werden kann, wenn eine Nachricht mit bestimmten Informationen vorliegt. Die Nachricht agiert in dem Fall sozusagen als „Trigger“ für die entsprechende Aufgabe.

AMBOSS bietet darüber hinaus die Möglichkeit, einzelnen Aufgaben Werte für deren minimale, maximale und durchschnittliche Ausführungszeit zuzuweisen. Diese zeitlichen Informationen werden in der Simulationsfunktion verwendet, sodass z.B. während einer Simulation die Gesamt-Ausführungszeiten pro Hierarchieebene des Aufgabenmodells berechnet und angezeigt werden. AMBOSS wurde an der Universität Paderborn entwickelt und steht für MS Windows XP und Linux auf der Internetseite (University of Paderborn, 2009) zum kostenlosen Download zur Verfügung.

CTTE[17] ist ein Werkzeug zur Erstellung und Simulation von graphischen und textuellen Aufgabenmodellen in CTT-Notation (s. Abschnitt 3.3.3). Es dient zur Modellierung einer Aufgabenspezifikation mit dem Ziel, diese auf eine Software-Architektur abzubilden, die auch die modellierten temporalen Relationen berücksichtigt. Darüber hinaus kann anhand eines erstellten Modells eine Simulation durchgeführt werden, die insbesondere zur Validierung des Modells dient. In diese Simulation kann der Analyst zur Laufzeit eingreifen und den zu betrachtenden Aufgabenweg durch das Modell bestimmen. Die zeitliche Analyse steht allerdings nicht im Vordergrund dieses Werkzeugs, sodass damit z.B. auch keine automatische Berechnung von Ausführungszeiten möglich ist. CTTE ist in Java imple-

[17] ConcurTaskTree Environment

mentiert und liegt aktuell in der Version 2.4.4 vor. Die Software kann auf der Internetseite (Mori et al., 2010) kostenlos heruntergeladen werden.

Unter den hier vorgestellten Werkzeugen sind interessante und leistungsstarke Anwendungen zu finden, die teilweise auch Zeitanalysen von Aufgabenmodellen ermöglichen. Das in dieser Arbeit betrachtete Werkzeug PED verfolgt die Aufgabenanalyse aus einer zu allen vorgestellten Werkzeugen unterschiedlichen Sichtweise. In PED wird aufbauend auf dem in Kapitel 2.3 vorgestellten Regelsystem ein Aufgabenmodell spezifiziert. Die Zeitanalyse muss unter Berücksichtigung des Regelsystems durchgeführt werden und ist deshalb anderen, neuen Herausforderungen ausgesetzt (s. Kapitel 6), was die Nutzung bereits bestehender Werkzeuge zur Zeitanalyse unmöglich macht. Darüber hinaus berücksichtigen Werkzeuge wie CogTool, APEX oder AMBOSS zwar einige der hier gewünschten Zeitkonzepte (vgl. Kapitel 5.2.2), aber keines dieser Werkzeuge vereint alle, in dieser Arbeit bzgl. des Szenarios für sinnvoll erachteten Konzepte. AMBOSS liegt zwar von der Funktionalität her sehr nah, berücksichtigt jedoch nicht die speziellen, mengenbezogenen zeitlichen Relationen, die insbesondere für das sicherheitskritische Szenario (s. Kapitel 4) als wichtig identifiziert wurden (vgl. Kapitel 5.2.2).

3.3.5 Zusammenfassung von und Ausblick auf weitere Einflussfaktoren bei Zeitanalysen

In Kapitel 3.3 wurden einige Faktoren aufgezeigt, die maßgeblichen Einfluss auf Zeitanalysen haben können. Neben dem seriellen Flaschenhals, dem Geübtheitsgrad einer Aufgabe (s. Abschnitt 3.3.2) oder der Untrennbarkeit von Aktionen durch die Definition von Task-Units wurde auf viele einschränkende Faktoren eingegangen, die sich aus der Aufgabenlogik ergeben (s. Abschnitt 3.3.1).

Die Berücksichtigung paralleler Ausführung bzw. des Multitasking-Prinzips bringt eine Reihe weiterer Faktoren mit sich. So stellten Wissenschaftler fest, dass der Aufgabe, der ein Nutzer die höchste Priorität zuweist, deutlich effizienter und aufmerksamer nachgegangen wird, als den Nebenaufgaben. Dieses Verhalten zeigt sich z.B. in den Verschachtelungsstrategien von Nutzern (Brumby et al., 2007). Wie bereits in obigem Abschnitt zu Task-Units erläutert, hat zudem die Kontrolle des Nutzers über den Beginn einer Unterbrechung Auswirkungen auf die Ausführungszeit der Gesamtaufgabe (McFarlane, 1999). Darüber hinaus entstehen bei

der Verschachtelung von Aufgaben Zusatzkosten in Form von Zeit, sogenannte „Switching Costs“ oder „Resumption Lags“, die durch Unterbrechungs- und Wiederaufnahmeprozesse bedingt sind. Deren Ausprägung kann wiederum von der Komplexität (Jin und Dabbish, 2009; Wickens, 2002) oder Verwandtschaft der Aufgaben zueinander (Gillie und Broadbent, 1989) oder der unterschiedlichen Verschachtelungsstrategien (Brumby et al., 2007) bzw. einzelnen Unterbrechungsdauern beeinflusst werden, da durch diese ggfs. zusätzliche Rekonstruktionsprozesse erforderlich sind (Salvucci, 2010; Monk et al., 2008). Natürlich tragen auch die Ausführungszeiten der kleinsten Aktionen bzw. Operatoren selbst maßgeblich zur Ausführungszeit der gesamten Aufgabe bei. Diese beruhen i.d.R. auf durchschnittlichen Erfahrungswerten oder tatsächlich zuvor gemessenen Zeiten (Card et al., 1983).

Ein weiterer wichtiger Einflussfaktor auf die Ausführungszeit können Fehler sein, die der Nutzer macht. Es ist anzunehmen, dass Fehler die Ausführungszeit verlängern oder zum Scheitern der Aufgabendurchführung führen können (Card et al., 1983, S. 176f). Ein Konzept zur Einbindung von Fehlern in Aufgabenanalysen als Grundlage für die Berechnung von Ausführungszeiten wurde bereits in einer früheren Arbeit vorgestellt (Fortmann, 2008). Darüber hinaus können emotionale Faktoren wie die Zufriedenheit oder das Stressempfinden bzw. die Arbeitslast des Nutzers betrachtet werden und hinsichtlich möglicher Auswirkungen auf die Aufgabendurchführung und damit die Zeitanalyse analysiert werden (Kieras und Polson, 1985; Gil et al., 2009).

Die Vielzahl an möglichen Einflussfaktoren zeigt die Komplexität und den Detailgrad, mit denen Zeitanalysen durchgeführt werden können. Im Rahmen dieser Arbeit werden jene Faktoren berücksichtigt, die sich aus der Aufgabenlogik ergeben.

4 Anwendungsszenario für die Zeitanalyse

Dieses Kapitel stellt das Anwendungsszenario dieser Arbeit vor. Als Szenario wird hier eine komplette Aufgabensituation verstanden, d.h., es handelt sich dabei nicht um eine konkrete, simulierte Aufgabenabfolge. Alle möglichen Abläufe innerhalb der Aufgabensituation sind hier aufgeführt, d.h., die Situation wurde nicht durch spezielle Ereignisse oder geltende Bedingungen eingeschränkt.

4.1 Eine Alltagssituation im Flugzeug-Cockpit

Als Anwendungsbeispiel für die Zeitberechnung im Rahmen einer Aufgabenanalyse wird die Modellierung von zwei nebenläufig durchzuführenden Aufgaben bzw. Zielen im Flugzeug-Cockpit während des Fluges betrachtet. Der Pilot soll einerseits die Bordinstrumente kontrollieren und andererseits eine neue Flugroute mit Hilfe einer interaktiven Benutzungsschnittstelle generieren und diese anschließend von der Flugkontrolle bestätigen lassen. Diese beiden Hauptaufgaben setzen sich aus verschiedenen Teilaufgaben zusammen. Die Kontrolle der Bordinstrumente beinhaltet die einzelne Kontrolle von fünf verschiedenen Instrumenten des Cockpits, die z.B. Geschwindigkeit und Raumlage des Flugzeugs während des Fluges anzeigen. Die Erzeugung einer neuen Flugroute wird anhand eines automatisch generierten Vorschlags vorgenommen und erfordert u.a. die Überprüfung von Wegpunkten und Flughöhen zu verschiedenen Streckenpunkten der Route, bevor nach einer Bestätigung der Flugsicherung der Autopilot entsprechend aktualisiert wird. Bemerkt wird an dieser Stelle, dass es sich bei diesem Szenario um einen von der Realität abstrahierten Aufgabenablauf handelt.

Nachfolgend werden die einzelnen Teilaufgaben bzw. Unterziele und deren Ablauf im Rahmen des vorgestellten Szenarios aufgelistet.

4.1.1 Aufgabe: Monitoring der Borddisplays im Cockpit

Monitor[1]

- Führe Aufgabe „Monitor Display A“, „Monitor Display B“, „Monitor Display C“, „Monitor Display D“ und „Monitor Display E“ aus.

Monitor Display A „Fahrtmesser“

- Wahrnehmen des Zustandes von Display A
- Bewerten des Zustandes von Display A: Wenn Zustand angemessen ist, dann beende Aufgabe „Monitor Display A“, sonst starte Aufgabe „Ausnahme-Behandlung“.

Monitor Display B „Fluglageanzeiger“

- Wahrnehmen des Zustandes von Display B
- Bewerten des Zustandes von Display B: Wenn Zustand angemessen ist, dann beende Aufgabe „Monitor Display B“, sonst starte Aufgabe „Ausnahme-Behandlung“.

Monitor Display C „Höhenmesser“

- Wahrnehmen des Zustandes von Display C
- Bewerten des Zustandes von Display C: Wenn Zustand angemessen ist, dann beende Aufgabe „Monitor Display C“, sonst starte Aufgabe „Ausnahme-Behandlung“.

Monitor Display D „Variometer“

- Wahrnehmen des Zustandes von Display D
- Bewerten des Zustandes von Display D: Wenn Zustand angemessen ist, dann beende Aufgabe „Monitor Display D“, sonst starte Aufgabe „Ausnahme-Behandlung“.

[1] engl.: überwachen, kontrollieren

Monitor Display E „Wendezeiger“

- Wahrnehmen des Zustandes von Display E
- Bewerten des Zustandes von Display E: Wenn Zustand angemessen ist, dann starte Aufgabe „Monitor“ erneut, sonst starte Aufgabe „Ausnahme-Behandlung“.

4.1.2 Aufgabe: Ausnahme-Behandlung

Ausnahme-Behandlung

- Beende Aufgabe „Ausnahme-Behandlung“.

Diese Aufgabe ist abstrakt, d.h. sie wird nicht genauer spezifiziert, weil dies für den weiteren Verlauf dieser Arbeit nicht relevant ist. Sie ist an dieser Stelle lediglich der Vollständigkeit halber aufgeführt. Da die Modellierung dieser Aufgabe in PED jedoch eindeutig sein muss, wird sie dort wie eine „leere Aufgabe“ interpretiert, d.h., dass für die Durchführung dieser Aufgabe „nichts“ getan werden muss und der Weg durch das Aufgabenmodell an der entsprechenden Stelle dann – im Sinne des Aufgabenmodells – erfolgreich beendet ist.

4.1.3 Aufgabe: Verhandlung einer Trajektorie mit der Flugsicherung (ATC)

Annahme: Uplink-Meldung[2] über eingehende Flugroute ist bereits auf dem Bildschirm sichtbar.

Verhandle Trajektorie

- Führe die Aufgaben „Generiere Trajektorie“ und „Aktualisiere Autopilot“ aus.

Generiere Trajektorie

- Wahrnehmen der Wegpunkte

[2]Meldung von der Bodenstation zum Flugzeug

- Auswerten der Wegpunkte: Wenn Wegpunkte akzeptabel sind, dann drücke GENERATE-Button und führe die Aufgabe „Horizontale Kontrolle“ und „Vertikale Kontrolle“ aus, sonst drücke REJECT-Button und starte Aufgabe „Generiere Trajektorie“ erneut.

Horizontale Kontrolle

- Drücke HOR-Button
- Wahrnehmen der Flughöhen der generierten Trajektorie
- Auswerten der Flughöhen anhand von Bedingungen: Wenn Flughöhen akzeptabel sind, dann beende Aufgabe „Horizontale Kontrolle“, sonst drücke REJECT-Button und starte Aufgabe „Generiere Trajektorie“ erneut.

Vertikale Kontrolle

- Drücke VERT-Button
- Wahrnehmen der Wegpunkte der generierten Trajektorie
- Auswerten der Wegpunkte: Wenn Wegpunkte akzeptabel sind, dann drücke SEND-TO-ATC-Button und beende Aufgabe „Vertikale Kontrolle“, sonst drücke REJECT-Button und starte Aufgabe „Generiere Trajektorie“ erneut.

Aktualisiere Autopilot

- Wahrnehmen des Zustandes der Bestätigung von ATC
- Auswerten: Wenn Bestätigung von ATC erhalten, dann drücke ENGAGE!-Button, sonst starte Aufgabe „Generiere Trajektorie“ erneut.

Das Szenario wird in Form eines gerichteten Graphs im Sinne eines Aufgabenmodells in den Abbildungen 4.1 und 4.2 veranschaulicht. Dort sind alle Aufgaben und Teilaufgaben aufgeführt. Das Modell ist wie in Abschnitt 2.2 beschrieben zu interpretieren. Abgerundete Rechtecke stellen Aufgaben bzw. Ziele (Goals) dar, während die anderen Rechtecke Operatoren, d.h. Aktionen auf unterster Hierarchieebene, repräsentieren. Bedingungen sind direkt an Kanten notiert und steuern den Ablauf. Schwarze waagerechte Balken dienen der besseren Veranschaulichung von zusammenhängenden Knoten, d.h. jenen Knoten, die alle an eine gemeinsame Bedingung geknüpft sind. Symbole für leere Mengen symbolisieren das Ende einer Aufgabe bzw. eine leere rechte Seite einer Regel (s. Kapitel 5.1 dieser Arbeit). Die

in PED modellierte Prozedur zu diesem Szenario ist in XML-Notation im Anhang I zu finden.

4.2 Zeitliche Bedingungen zwischen den einzelnen Aufgaben

Die Tabelle 4.1 listet alle festgelegten zeitlichen Bedingungen zwischen den Aufgaben des Szenarios auf. Die Bedeutung der einzelnen zeitlichen Bedingungen („Constraints“) wird im nächsten Kapitel 5.2 erläutert.

1. Aufgabe	Constraint	2. Aufgabe
InitializeTask	max 180sek	
Monitor	max 40sek	
EditRoute	max 150sek	
Monitor	interleaves	EditRoute
MonitorD	max 5sek nach	MonitorC
CheckHorizontal	min 1sek nach	PressGenerate
UpdateAutoPilot	min 8sek nach	PressSendTo
UpdateAutoPilot	max 120sek nach	PressSendTo
MonitorB	nach	MonitorA
MonitorC	nach	MonitorB
MonitorE	nach	MonitorD
MonitorA0	strikt nach	PerceptGoalA
MonitorB0	strikt nach	PerceptGoalB
MonitorC0	strikt nach	PerceptGoalC
MonitorD0	strikt nach	PerceptGoalD
MonitorE0	strikt nach	PerceptGoalE
GenerateTrajectory0	strikt nach	PerceptGoalWp
GenerateTrajectory	nach	PressReject
CheckVertical	nach	CheckHorizontal
CheckHorizontal0	strikt nach	PerceptGoalAlt

Fortsetzung auf nächster Seite

Fortsetzung von vorhergehender Seite

1. Aufgabe	Constraint	2. Aufgabe
CheckVertical0	strikt nach	PerceptGoalWp
UpdateAutoPilot0	strikt nach	PerceptGoalCon

Tab. 4.1: Zeitliche Bedingungen zwischen den Aufgaben des Szenarios

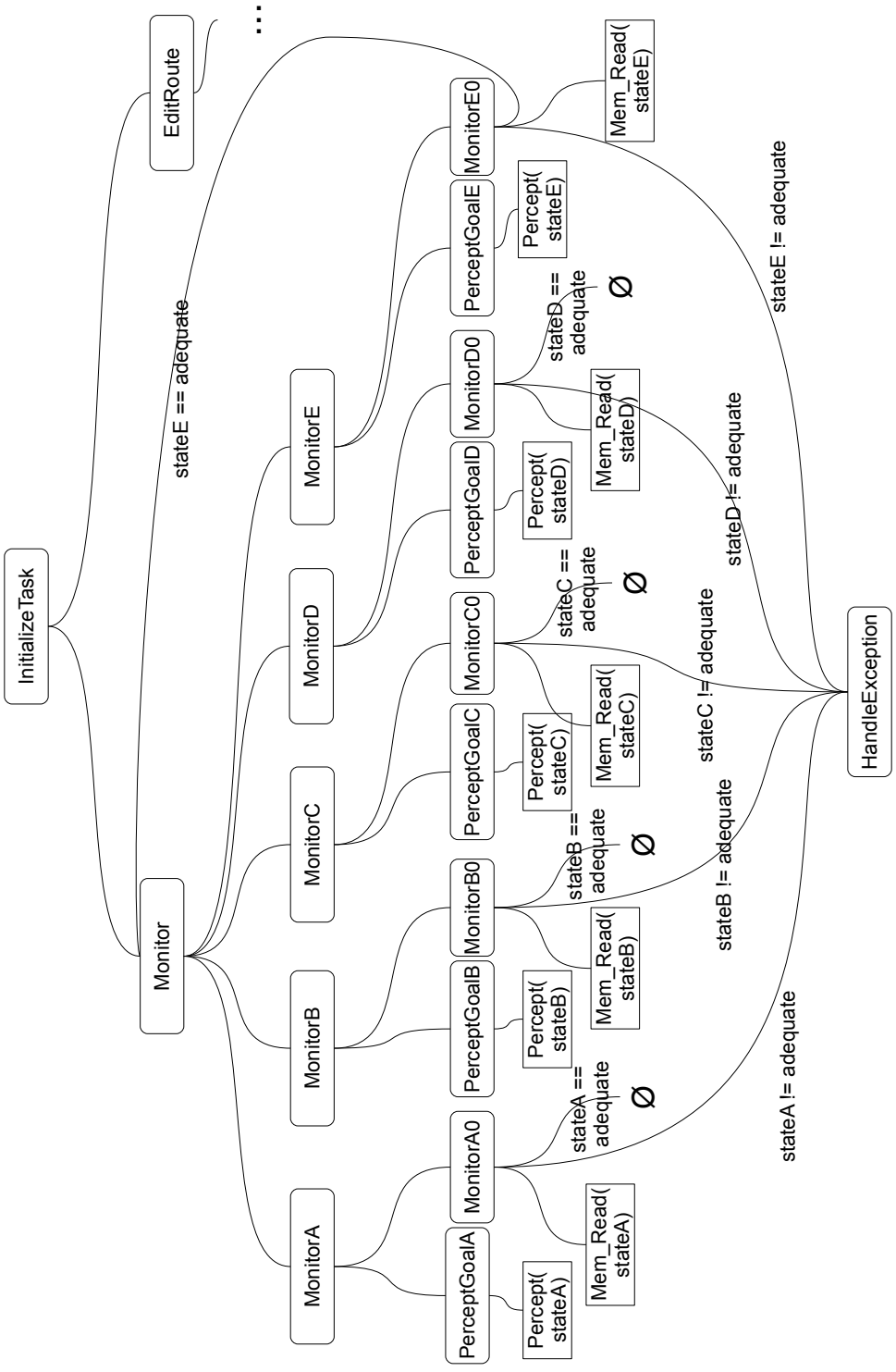


Abb. 4.1: Aufgabenmodell des Szenarios, Teil 1

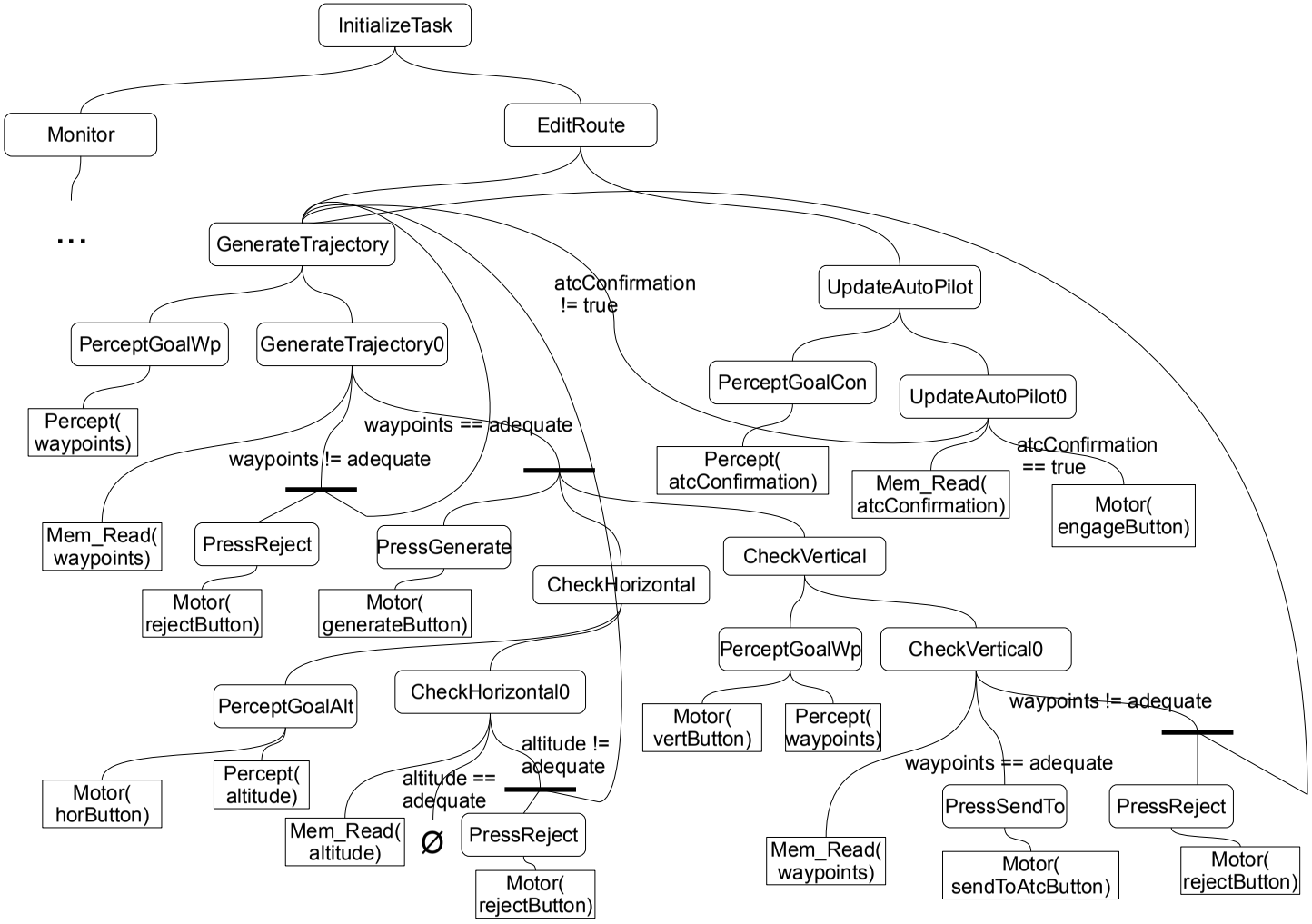


Abb. 4.2: Aufgabenmodell des Szenarios, Teil 2

5 Erweiterung des Regelsystems um zeitliche Abfolgen

Dieses Kapitel stellt die Erweiterung des Regelsystems vor, wodurch die Berücksichtigung von zeitlichen Abfolgen innerhalb des Regelsystems ermöglicht wird. Zunächst wird die grundlegende Funktionsweise des Regelsystems erklärt und anschließend die formale Syntax einer Prozedur dieses Regelsystems beschrieben. Im zweiten Teil des Kapitels wird die Erweiterung vorgestellt, indem zunächst auf die ausgewählten Zeitkonzepte eingegangen und dann die formale Beschreibung der veränderten bzw. ergänzten Prozeduren-Syntax vorgenommen wird.

5.1 Analyse des Regelsystems von PED

In Kapitel 2.3 wurden die Grundlagen des in dieser Arbeit betrachteten Regelsystems vorgestellt. In diesem Abschnitt werden bestimmte Teile des Regelsystems genauer betrachtet, da sie für das Hauptziel dieser Arbeit, d.h. die Berechnung von Ausführungszeiten für Aufgabensequenzen erforderlich sind.

Bevor konkret auf die formale Syntax der Regelsprache eingegangen wird, wird im Folgenden ihr Grundprinzip mit den wichtigsten Regelelementen anhand von Abbildung 5.1 erläutert.

Zu sehen sind die beiden Regeln *Rule1* und *Rule2*. Auf der LHS der Regeln ist jeweils das Ziel (*Goal*) angegeben, für das die Regel definiert ist. Sowohl *Rule1* als auch *Rule2* sind für *Goal1* definiert. Beide Regeln weisen eine optionale Bedingung (*Condition*) auf, die ebenfalls auf der LHS definiert ist. *Rule1* darf nur gefeuert werden, wenn die Variable a den Wert 1 hat, wohingegen *Rule2* gefeuert wird, wenn a ungleich 1 ist. Eine Condition besteht folglich aus zwei Vergleichsoperatoren, in diesem Fall „a“ und dem Wert 1, sowie einem booleschen Operator,

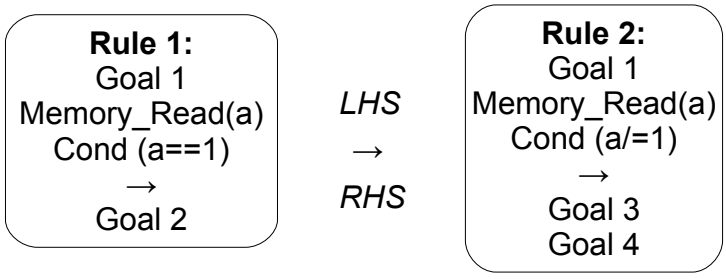


Abb. 5.1: Beispiel für die Definition zweier Regeln im PED-Regelsystem

der hier durch „==“ bzw. „/ =“ spezifiziert ist. Die Auswertung einer Condition erfordert, dass die aktuelle Belegung der auszuwertenden Variable bekannt ist. Um diesen Gedächtnisabruf explizit anzufordern, wird eine *memory_ read*-Aktion auf der LHS der Regel angegeben.

Ein rechtsgerichteter Pfeil trennt LHS und RHS voneinander. Die RHS kann neben Zielen auch Operatoren, d.h. konkrete Aktionen (*memory_ store, percept, motor*) beinhalten. *Rule1* definiert *Goal2* als einziges RHS-Goal, während *Rule2* die beiden Ziele *Goal3* und *Goal4* als RHS-Goals angibt. Diese Angabe bedeutet, dass beim Feuern von *Rule1* das Ziel *Goal2* verfolgt wird, wohingegen beim Feuern von *Rule2* die Ziele *Goal3* und *Goal4* verfolgt werden. Das Ziel *Goal1* kann demnach auf zwei unterschiedlichen Wegen erreicht werden, die durch die beiden Regeln beschrieben werden. Auch eine leere RHS ist möglich und drückt aus, dass „nichts“ getan werden soll, um das auf der LHS definierte Ziel zu erreichen. In welcher Reihenfolge z.B. *Goal3* und *Goal4* erreicht werden, wird durch das Regelsystem bisher nicht definiert. Diesen Mangel soll das im folgenden Abschnitt 5.2 vorgestellte Konzept zur Erweiterung des Regelsystems beheben. Zuvor wird jedoch noch konkreter auf die Syntax des bestehenden Regelsystems eingegangen.

Die folgende Tabelle 5.1 beschreibt die Syntax einer Prozedur des Regelsystems von PED in EBNF[1].

procedure	=	{rule}
rule	=	standard_rule | reactive_rule
standard_rule	=	id, type, lhs '⇒' rhs
reactive_rule	=	id, type, lhs_without_goal '⇒' rhs

[1]Die Erweiterte Backus-Naur-Form (EBNF) ist eine Erweiterung der Backus-Naur-Form (BNF). Dabei handelt es sich um eine formale Sprache, die zur Beschreibung der Syntax von kontextfreien Grammatiken dient. In der Norm ISO/IEC 14977:1996(E) ist die EBNF standardisiert.

lhs	=	goal, {memory_read}, [condition]
rhs	=	{memory_store}, {percept}, {motor}, {goal}
lhs_without_goal	=	{memory_read}, [condition]
goal	=	'Goal(' goal_name ')'
condition	=	'Condition(' bool_expr ')'
memory_read	=	'Memory-Read(' object ')'
memory_store	=	'Memory-Store(' object ')'
percept	=	'Percept(' object ')'
motor	=	'Motor(' object ')'
id	=	digit, {digit}
type	=	'standard' | 'percept' | 'reactive'
goal_name	=	identifier
bool_expr	=	item, bool_op, item
object	=	string '.' string
identifier	=	alpha_upper_case, {cyms_upper_case}
item	=	alpha_lower_case, {cyms_lower_case}
bool_op	=	'>=' | '<=' | '>' | '<' | '/ =' | '=='
alpha_upper_case	=	'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
cyms_upper_case	=	alpha_upper_case | digit | '_'
alpha_lower_case	=	'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
cyms_lower_case	=	alpha_lower_case | digit | '_'
digit	=	digit_without_zero | '0'
digit_without_zero	=	'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
character	=	alpha_upper_case | alpha_lower_case | digit | '_' | '-'
string	=	character, {character}

Tab. 5.1: Syntax einer Prozedur des Regelsystems von PED

Werden die Bestandteile einer Standardregel (*standard_rule*) hinsichtlich ihrer Bedeutung für Zeitberechnungen betrachtet, so weist die linke Seite alle Ziele der rechten Seite einem gemeinsamen Oberziel zu, d.h., alle Ziele der RHS tra-

gen zur Erreichung des Oberziels auf der LHS bei. Zusätzlich können in der LHS Gedächtnisabrufe (*memory_read*), sowie die Auswertung einer Bedingung (*condition*) modelliert sein. Auch diese Konstrukte tragen zur Ausführungszeit bei und müssen bei der Zeitberechnung berücksichtigt werden. Jedoch ist durch die Arbeitsweise des zu Grunde liegenden Regelsystems deren Ausführungsreihenfolge bereits festgelegt. Die Ausführungsreihenfolge ist wie folgt definiert: Alle zu einem Ziel zugehörigen Regeln werden identifiziert. In diesen Regeln werden alle *memory_read*-Operatoren auf der jeweiligen LHS gesammelt. Alle erforderlichen Variablenwerte werden abgerufen bzw. ggfs. entsprechende Perzeptregeln gefeuert, die die unmittelbare Wahrnehmung der Variablen veranlassen. Anschließend werden die Bedingungen (*conditions*) auf der LHS aller Regeln ausgewertet. Unter allen Regeln, deren Bedingung erfüllt ist, wird anschließend eine Regel ausgewählt, wodurch die Ausführung der Operatoren bzw. Verfolgung der Ziele auf der RHS dieser Regel veranlasst wird. Die Definition weiterer Zeitkonzepte auf der LHS einer Regel erübrigt sich dadurch. Die Reihenfolge der Operatoren auf der RHS wird sequentiell nach Modellierungsreihenfolge interpretiert. Allerdings muss zur Zeitberechnung insbesondere die konkrete Abfolge der Ziele auf der RHS definiert sein. Durch die oben beschriebene aktuelle Syntax werden jedoch keine konkreten zeitlichen Abfolgen oder zeitlichen Abhängigkeiten zwischen Unterzielen definiert, stattdessen wird angenommen, dass alle Ziele der RHS ausnahmslos sequentiell, d.h. wie modelliert nacheinander verfolgt werden. Um die in Abschnitt 5.2.2 vorgestellten Zeitkonzepte in die Syntax integrieren zu können, muss demnach die RHS einer Regel entsprechend erweitert werden. Um ebenfalls *memory_read*-Operatoren und *conditions* der LHS in die Zeitberechnung einfließen lassen zu können, ist darüber hinaus eine geringfügige Erweiterung der LHS nötig.

Reaktive Regeln (*reactive_rule*) werden in dieser Arbeit nicht betrachtet, da sie im Rahmen dieser Arbeit wenig relevant erscheinen. Diese Regeln sind keinem Ziel zugeordnet und beschreiben vorwiegend unvorhersehbare, überraschende Situationen mit entsprechend reflexartigem Verhalten, d.h., sie können jederzeit auftreten und müssen in keinem direkten Kontext zu dem aktuellen Ziel stehen. Sollen Ausführungszeiten von bestimmten Aufgaben, d.h. Zielen berechnet werden, würde die Berücksichtigung solcher Regeln die Berechnungen u.U. erheblich verfälschen und deren Aussagekraft stark mindern, da ein unberechenbarer Faktor einbezogen wird.

5.2 Erweitertes Regelsystem in PEDTime

Die in dieser Arbeit vorgestellte Erweiterung des Procedure Editors um die Funktion der Zeitanalyse wird PEDTime genannt. Im Folgenden wird die dafür notwendige Erweiterung des Regelsystems um konkrete zeitliche Bedingungen beschrieben.

5.2.1 Anforderungsdefinition

Das Regelsystem soll dahingehend erweitert werden, dass eine zeitliche Ordnung zur Ausführung der einzelnen Ziele spezifiziert werden kann. Dies setzt voraus, dass für das Szenario geeignete Zeitkonzepte aus den in Kapitel 3.3 vorgestellten Konzepten ausgewählt bzw. von diesen abgeleitet werden. Anhand der zeitlichen Ordnung soll anschließend die Berechnung von Ausführungszeiten vorgenommen werden, auf die explizit in Kapitel 6 eingegangen wird.

5.2.2 Ausgewählte Zeitkonzepte

Da es sich bei dem Szenario um ein Beispiel aus dem sicherheitskritischen Bereich handelt, erscheint insbesondere die Verwendung von Zeitkonzepten, die konkrete Zeitdauern definieren sinnvoll. In diesem Bereich können minimale zeitliche Abweichungen bei der Durchführung einer Aufgabe schwerwiegende Folgen haben. Dies wird an folgendem Beispiel verdeutlicht: Im Verlauf eines Fluges werden sehr hohe Geschwindigkeiten erreicht, d.h., dass sich ein Flugzeug während kurzer Reaktionsverzögerungen seines Pilotens bereits in einem großen Raum fortbewegt hat. Fliegt ein Flugzeug z.B. mit einer Geschwindigkeit von 900km/h, so bewegt es sich in 1 Sekunde um 250 Meter in horizontale Richtung. Hinzu kommen Bewegungen in vertikale und seitliche Richtung, wodurch sich der betroffene Raum zusätzlich vergrößert. Aus diesem Grund wurde der Schwerpunkt der ausgewählten Zeitkonzepte auf sequentielle, mit konkreten Zeitwerten definierte Bedingungen gelegt. Die Definition von Serialität ist unerlässlich, da die meisten von einem Menschen durchgeführten Aufgaben nacheinander abgearbeitet werden. Gerade Dialogsysteme, wie die im Szenario betrachtete Schnittstelle zur Eingabe einer Flugroute, erfordern serielle Abläufe, die ggfs. auch strikt seriell, d.h. sequentiell

sein müssen. Sind serielle und sequentielle Zeitkonzepte berücksichtigt, kann zusätzlich die Verwendung einer Bedingung, die nebenläufige Ausführung erlaubt, sinnvoll sein. Im Szenario hat der Pilot neben der Eingabe einer Flugroute die Aufgabe, die Bordinstrumente zu kontrollieren. Die Verschachtelung dieser beiden Aufgaben bietet sich an und ist ggfs. sogar erforderlich, um allen zeitlichen Vorgaben gerecht werden zu können. Zudem ist hinsichtlich der Ausführungszeit beider Aufgaben interessant, wie sich eine Verschachtelung auswirkt.

In Kapitel 3.3 wurden verschiedene Ansätze für zeitliche Relationen vorgestellt. Die hier ausgewählten Konzepte wurden aus diesen Ansätzen aufgegriffen bzw. von diesen abgeleitet. Wie auch bei Allen (1983) werden hier Zeitintervalle zueinander bzw. zu sich selbst in zeitliche Beziehungen gesetzt. Da Aufgaben als Zeitintervalle betrachtet werden können, ist die Verwendung dieses Ansatzes naheliegend. Das serielle Konzept, das hier verwendet wird, findet sich ebenfalls bei Allen (1983) in Form der Relationen *before* und *after* wieder. Da die eine Relation lediglich das Inverse der anderen Relation darstellt, wird hier nur die Relation *after* verwendet. Sequenz, bei Allen (1983) durch *meets* und *met by* ausgedrückt, wird hier in Form des *strikt nach*-Konzepts definiert. Allen's weitere Konzepte *overlaps, overlapped by, during, contains, starts, started by, finishes, finished by* drücken Nebenläufigkeit aus und werden in dieser Arbeit unter dem Zeitkonzept *verschachtelt* zusammengefasst, da zusätzliche, spezielle serielle Konzepte als – auf das Szenario bezogen – wichtiger eingestuft werden, als die separate Betrachtung der Verschachtelungskonzepte. Neben Allen verwenden weitere Ansätze wie z.B. UAN (Hartson und Gray, 1992) und CTT (Paternò et al., 1997), sowie die Werkzeuge AMBOSS (Giese et al., 2008) und APEX (NASA, 2006b) die Konzepte Sequenz und Verschachtelung. Zusätzlich zu den o.g. zeitlichen Relationen wird in UAN das *waiting*-Konzept definiert, welches sich in dieser Arbeit in der Relation *min T nach* wiederfindet und eine minimale Pufferzeit zwischen zwei Aufgaben spezifiziert. Daraus abgeleitet wurden in dieser Arbeit die Konzepte *max T nach* und *genau T nach*, die die Berücksichtigung weiterer mengenbezogener, temporaler Aspekte vorsehen. Dazu zählt auch das in XUAN (Lacaze und Palanque, 2004) vorgestellte Konzept der Dauer eines Prozesses, welches ebenfalls in dieser Arbeit berücksichtigt wird. Die in den *Plans* von Annett (Diaper und Stanton, 2004) und in CTT (Paternò et al., 1997) zusätzlich aufgeführten Konstrukte *entweder/oder* bzw. *Auswahl* und *Wiederholung* sind bereits durch das derzeitige Regelsystem von PED modellierbar und werden hier nicht als neue Zeitkonzepte aufgefasst.

Im Folgenden werden die o.g. und in dieser Arbeit verwendeten Zeitkonzepte, die zur Ablaufplanung des in Kapitel 4 vorgestellten Anwendungsszenarios dienen, definiert und erläutert.

(1) Prozess B nach Prozess A seriell: Ein Prozess B wird irgendwann nach der Durchführung von Prozess A gestartet. Dazwischen können auch andere Prozesse stattfinden. (A $\rightarrow [*] \rightarrow$ B)

(2) Prozess B strikt nach Prozess A sequentiell: Ein Prozess B wird in direkter Reihenfolge nach der Durchführung von Prozess A gestartet. Zwischen A und B dürfen keine anderen Prozesse stattfinden. (A $\rightarrow$ B)

(3a) Prozess B genau T nach Prozess A seriell: Ein Prozess B wird gestartet, nachdem genau eine bestimmte Zeitdauer T nach Beendigung von Prozess A vergangen ist. Dabei gilt stets: $T > 0$. Zwischen A und B dürfen dabei auch andere Prozesse stattfinden. ($A \underbrace{\rightarrow [*] \rightarrow}_{T} B$)

(3b) Prozess B max T nach Prozess A seriell: Ein Prozess B wird gestartet, nachdem höchstens (maximal) eine bestimmte Zeitdauer T nach Beendigung von Prozess A vergangen ist. Dabei gilt stets: $T > 0$. Zwischen A und B dürfen dabei auch andere Prozesse stattfinden. ($A \underbrace{\rightarrow [*] \rightarrow}_{max\ T} B$)

(3c) Prozess B min T nach Prozess A seriell: Ein Prozess B wird gestartet, nachdem mindestens (minimal) eine bestimmte Zeitdauer T nach Beendigung von Prozess A vergangen ist. Dabei gilt stets: $T > 0$. Zwischen A und B dürfen dabei auch andere Prozesse stattfinden. ($A \underbrace{\rightarrow [*] \rightarrow}_{min\ T} B$)

(4) Prozess A und Prozess B verschachtelt nebenläufig: Die Prozesse A und B werden nebenläufig ausgeführt, d.h. alle Unterprozesse von A und B können – sofern dabei keine weiteren Zeitkonzepte verletzt werden – in beliebiger Reihenfolge nacheinander ausgeführt werden. ($A = \{a0, a1, a2\}$, $B = \{b0, b1, b2\}$; dann z.B. $A\_verschachtelt\_B = a0 \rightarrow b1 \rightarrow b0 \rightarrow a2 \rightarrow b2 \rightarrow a1$)

(5) Prozess A max T Prozessdauer: Ein Prozess A darf maximal eine bestimmte Zeitdauer zur Ausführung benötigen. ($\underbrace{A}_{T}$)

Der Begriff Prozess wurde auf Grund des besseren Verständnisses gewählt, da es sich um zeitliche Prozesse handelt, über die Aussagen getroffen werden. In Aufgabenmodellen werden Prozesse durch Aufgaben bzw. Ziele, genauer durch das Erreichen von Zielen, repräsentiert. Angelehnt an das Regelsystem werden die o.g. zeitlichen Bedingungen nicht nur auf das Ziel, für das sie definiert sind bezogen, sondern auch auf alle indirekt mit der Ausführung des Ziels verbundenen Vorgänge. Denn diese Vorgänge kosten ebenfalls Zeit, die später bei der *Constraint*-Bewertung berücksichtigt werden muss. Solch ein Vorgang ist z.B. die Auswahl eines Ziels bzw. das Feuern einer Regel (s. Kapitel 6.2.2). Alle diese zu einem Ziel gehörenden Vorgänge werden intern durch eine Regel repräsentiert, die diesem Ziel zugeordnet ist (s. Kapitel 2.3).

Im Anwendungsszenario besteht eine Hauptaufgabe des Pilotens darin, die Bordinstrumente zu kontrollieren. Die Kontrolle der einzelnen Instrumente geschieht seriell, d.h., es ist festgelegt in welcher Reihenfolge die einzelnen Instrumente nacheinander kontrolliert werden. Diese zeitliche Spezifikation kann durch das erste Zeitkonzept ausgedrückt werden, da zwar die Reihenfolge festgelegt, eine strikte Sequenz jedoch nicht erfordert ist.

Um die Bordinstrumente kontrollieren zu können, muss deren gegenwärtiger Zustand zunächst wahrgenommen werden, bevor er ausgewertet werden kann und ggfs. entsprechende Folgehandlungen eingeleitet werden können. Die Sequenz Wahrnehmen-Auswerten soll strikt aufeinander folgen, da eine Unterbrechung dieser durch andere Prozesse später unter Umständen einen Mehraufwand auf Grund der Rekonstruktion der wahrgenommenen Werte verursachen würde. Durch die Verwendung des zweiten Zeitkonzepts, d.h. durch strikte Sequenz, kann dieser Mehraufwand, der insbesondere in verlängerter Gesamt-Ausführungszeit resultieren würde, verhindert werden.

Es kann jedoch auch sein, dass eine einfache Sequenz als Zeitbedingung nicht ausreicht. Im Anwendungsszenario wird angenommen, dass nur die zeitlich unmittelbar aufeinander folgende Zustandsauswertung zweier verschiedener Instrumente, wie Höhenmesser und Variometer (Vertikalgeschwindigkeit), ausreichend Sicherheit gewährleistet. Um diese Bedingung zu definieren, kann das dritte Zeitkonzept mit dem Parameter „max“ (3b) verwendet werden. Im Gegensatz dazu kann explizit gewünscht sein, dass ein Ziel frühestens aktiviert wird, nachdem eine bestimmte Zeitdauer nach dem vorherigen Ziel vergangen ist. Dies ist z.B. sinnvoll,

wenn zur Zieldurchführung bestimmte Informationen vorliegen müssen, die erst nach einer bestimmten Wartezeit erwartet werden. Dabei kann es sich z.B. um die Antwortzeit des Systems oder die Zeit bis zur Reaktion einer anderen Person handeln. Im Anwendungsszenario sind die Zeit, die das System zur Generierung einer Trajektorie benötigt, oder die Zeit, die die Flugsicherheit zur Kontrolle der vom Piloten erstellten Trajektorie benötigt, solche Fälle. Zur Spezifikation kann dann das dritte Zeitkonzept mit dem Parameter „min“ (3c) verwendet werden.

Der Pilot soll einerseits die Bordinstrumente kontrollieren und andererseits eine neue Flugroute erstellen und diese mit der Flugsicherheit verhandeln. Im Szenario wird angenommen, dass der Pilot die Bordinstrumente regelmäßig und häufig kontrollieren muss, d.h., er muss während er die Flugroute generiert gelegentlich die Bordinstrumente kontrollieren. Diese beiden Hauptaufgaben werden folglich verschachtelt, was durch das vierte Zeitkonzept definiert wird.

In sicherheitskritischen Bereichen ist es häufig erforderlich, dass bestimmte Aufgaben in einer bestimmten Zeit ausgeführt sein müssen. In der Cockpit-Situation wird festgelegt, dass beide Hauptaufgaben jeweils nur eine bestimmte maximale Dauer zur Durchführung beanspruchen dürfen. Dadurch wird gewährleistet, dass ein bestimmtes Maß an Sicherheit eingehalten wird, weil der Pilot tatsächlich auch zwischen den beiden Hauptaufgaben hin- und her wechseln muss und sie nicht einfach sequentialisieren kann. In diesem Szenario wird das fünfte Konzept z.B. verwendet, um eine regelmäßige Kontrolle der Bordinstrumente als entscheidenden Sicherheitsfaktor während des Fluges zu gewährleisten.

Sofern keine anderen zeitlichen Bedingungen definiert sind, die Serialität ausschließen, wird stets angenommen, dass Aufgaben seriell durchgeführt werden. Auf dieser Grundlage basiert auch das algorithmische Konzept, welches in Kapitel 6 vorgestellt wird.

5.2.3 Erweiterung der Prozeduren-Syntax hinsichtlich der Zeitkonzepte

Um die in Abschnitt 5.2.2 vorgestellten Zeitkonzepte in PED berücksichtigen zu können, muss zunächst die Prozeduren-Syntax erweitert werden. Im Folgenden

werden in der Tabelle 5.2 alle nötigen Änderungen und Ergänzungen der in Abschnitt 5.1 präsentierten Syntax aufgeführt.

lhs	=	goal, { '(' memory_read, id, duration ')' }, [condition]
rhs	=	{ '(' memory_store, id, duration ')' }, { '(' percept, id, duration ')' }, { '(' motor, id, duration ')' }, { '(' goal, id ')' }, {time_constraint}
time_constraint	=	'Time-Constraint(' (id, constraint_op1, id) | (id, constraint_op2) ')'
constraint_op1	=	'max_after', duration | 'min_after', duration | 'exactly_after', duration | 'after' | 'strictly_after' | 'interleave'
constraint_op2	=	'max', duration
duration	=	digit_without_zero, {digit}

Tab. 5.2: Erweiterung der Prozeduren-Syntax des PED-Regelsystems um Zeitkonzepte

Um eine bestimmte Reihenfolge bei der Ausführung von Zielen und Operatoren festzulegen und Bedingungen an unterschiedliche Ziele knüpfen zu können, ist eine eindeutige Kennung von Operatoren und Zielen erforderlich. Diese wird durch eine *id* angegeben, die zusammen mit dem entsprechenden Operator bzw. Ziel in einem Tupel in der RHS bzw. für *memory_ read* in der LHS definiert wird.

Für die Definition der Ausführungszeiten der Operatoren *memory_ read*, *memory_ store*, *percept* und *motor* wird zunächst auf Standardzeiten zurückgegriffen. Verfügt der Modellierer über präzisere Angaben zur Ausführungsdauer eines Operators, so soll es ihm möglich sein, diese speziell während der Aufgabenmodellierung festzulegen. Wird eine spezielle Ausführungszeit angegeben, so basiert darauf auch die Berechnung der Gesamt-Ausführungszeiten. Ist dies nicht der Fall, wird auf entsprechende Standardwerte zurückgegriffen. Das Feuern einer Regel, die Zielauswahl sowie die Auswertung einer *Condition* beanspruchen zusätzliche Zeit. Für diese Zeitwerte werden ebenfalls Standardzeiten angenommen, sofern der Modellierer diese Parameterwerte nicht explizit angibt (s. Kapitel 6).

(Sub-)Ziele aller Hierarchien können mit zeitlichen Bedingungen und Relationen spezifiziert werden. Für atomare Aktionen, d.h. Operatoren werden keine zeit-

lichen Bedingungen definiert. Diese Modellierungsentscheidung liegt einerseits der Funktionsweise des Regelsystems zu Grunde, wodurch bestimmte Operator-Reihenfolgen bereits festgelegt sind (s. Abschnitt 5.1). Andererseits trägt auch das berücksichtigte Konzept von Task-Units dazu bei (s. Abschnitt 3.3.1). Ist vom Modellierer dennoch explizit eine zeitliche Beziehung zwischen Operatoren gewünscht, so kann pro betroffenem Operator ein Oberziel definiert werden, welches ausschließlich dem jeweiligen Operator zugeordnet ist. Dadurch können die Operatoren indirekt über deren Oberziele in zeitliche Beziehungen zueinander gesetzt werden. Dabei ist zu berücksichtigen, dass durch die Einführung zusätzlicher Ziele die berechnete Ausführungszeit minimal verlängert wird.

Um zeitliche Relationen und Bedingungen zwischen bzw. für die auszuführenden Ziele einer Regel zu definieren, muss die RHS um die Angabe solcher Zusatzbedingungen erweitert werden. Eine solche Bedingung wird durch das Nichtterminal *time_ constraint* repräsentiert. Ein *time_ constraint* kann entweder eine zeitliche Relation zwischen zwei Zielen festlegen oder eine zeitliche Bedingung an ein Ziel knüpfen. Die Ziele werden dabei über ihre *id* angesprochen. Durch *constraint_ op1* und *constraint_ op2* werden alle in Abschnitt 5.2.2 aufgeführten Zeitkonzepte in der Prozeduren-Syntax, dargestellt durch die jeweils englische Übersetzung umgesetzt. Die Anordnung von Ziel-IDs und Constraint-Typ in der Syntax ist entsprechend der Leserichtung von links nach rechts zu interpretieren, d.h. „Time-Constraint((id_goal1, after, id_goal2))“ wird z.B. als „Ziel1 nach Ziel2“ interpretiert.

6 Konzept zur Zeitanalyse mit PEDTime

In diesem Kapitel wird zunächst ausführlich die Problemstellung der PEDTime-Zeitanalyse erläutert. Anschließend wird erläutert, inwiefern der entscheidende Teil des vorgestellten Problems als ein Ablaufplanungsproblem interpretiert werden kann. Daraufhin folgt die vollständige Konzeptbeschreibung von PEDTime sowie die kritische Betrachtung bestimmter Konzeptprobleme.

6.1 Problemstellung

Die Erweiterung PEDTime soll in PED modellierte Aufgabenmodelle hinsichtlich ihrer Ausführungszeit analysieren. Es sollen Vorhersagen zu Best-/Schlechtest- und Durchschnittszeiten aller gültigen Zielsequenzen getroffen werden. Dabei sollen zeitliche Bedingungen (Constraints) berücksichtigt werden, die zwischen einzelnen Aufgaben definiert sein können.

6.1.1 Anforderungen an PEDTime

Aus der Problemstellung ergeben sich die im Folgenden aufgeführten Anforderungen an PEDTime.

Erweiterungsmodul von PED

PEDTime soll kein eigenständiges Werkzeug, sondern eine Erweiterung von PED sein, d.h. es muss sich an speziellen Systemstrukturen von PED, wie z.B. dem Regelsystem und der darauf aufbauenden Datenstruktur orientieren.

Zeitanalyse

PEDTime soll anhand eines PED-Aufgabenmodells eine Zeitanalyse durchführen. Das Ergebnis der Zeitanalyse soll die beste (minimale), schlechteste (maximale) und durchschnittliche Ausführungszeit der möglichen Zielsequenzen beinhalten. Außerdem sollen die dazugehörigen Zielsequenzen ermittelt werden, sodass der Weg[1] durch das Aufgabenmodell rekonstruierbar ist.

Constraints

Eine Besonderheit der Zeitanalyse ist die Berücksichtigung von zeitlichen Bedingungen (Constraints), die für ein bestimmtes Ziel oder zwischen zwei bestimmten Zielen definiert sein können. Diese Constraints (s. Kapitel 5.2.2) schränken die möglichen Zielsequenzen ein und müssen überprüft werden.

Abgrenzung

In dieser Arbeit wird das algorithmische Konzept von PEDTime vorgestellt. Die Einbindung von PEDTime in PED, die z.B. die Konfiguration und Modellierung der von PEDTime benötigten Parameter (s. Kapitel 5), sowie die graphische Darstellung der PEDTime-Funktionalität und -Ergebnisse innerhalb von PED beinhaltet, ist nicht Teil dieser Arbeit.

6.1.2 Konkretisierung der Anforderungen

Bevor ein Konzept zur Zeitanalyse entwickelt werden kann, müssen weitere Konkretisierungen der o.g. Anforderungen vorgenommen werden, auf die nachfolgend eingegangen wird.

[1] Unter „Weg“ ist hier eine Instanz des Aufgabenmodells zu verstehen, d.h. ein vollständiger korrekter Weg zur Durchführung der Wurzelaufgabe des Modells ohne Berücksichtigung von Constraints.

Abbildung der Prozeduren-Syntax von PED auf PEDTime-Konzept

Die in Kapitel 5.1 vorgestellte Prozeduren-Syntax von PED ist sehr mächtig. Mit ihr lassen sich z.B. beliebig breite und tiefe Aufgabenhierarchien mit Bedingungen, Zyklen und „leeren“ Zielen, d.h. einer leeren RHS modellieren. Es gilt zu klären, inwiefern diese Modellierungsvielfalt für die Berechnung der Ausführungszeiten von Zielsequenzen erforderlich ist bzw. sinnvoll berücksichtigt werden kann.

Differenzierung von ODER- und UND-Verknüpfungen

In den mit PED modellierten Aufgabenmodellen können sowohl ODER- als auch UND-Verknüpfungen von Zielen definiert sein. Während per ODER verknüpfte Ziele Alternativen zur Erreichung des entsprechenden Oberziels darstellen, müssen alle per UND verknüpften Ziele erreicht sein, damit auch das entsprechende Oberziel erreicht wird. Die unterschiedlichen Verknüpfungen sind zu identifizieren und für die Erstellung gültiger Zielsequenzen zu berücksichtigen, denn per ODER verknüpfte Ziele dürfen nicht in derselben Zielsequenz vorkommen, während, sobald ein per UND verknüpftes Ziel in einer Zielsequenz enthalten ist, auch alle weiteren, mit ihr per UND verknüpften Ziele zu derselben Zielsequenz gehören müssen.

Rekonstruktionsmöglichkeit von Variablenbelegungen

Das Ergebnis einer mit PEDTime durchgeführten Zeitanalyse muss neben den gültigen Zielsequenzen auch die Werte der ggfs. zu belegenden Variablen beinhalten, die zu der jeweiligen Zielsequenz führen. Andernfalls wäre der exakte Weg durch das Aufgabenmodell nicht nachvollziehbar. Diese Variablen werden in der Condition einer Regel abgefragt. Somit müssen alle dieser Variablenbelegungen zu der entsprechenden Zielsequenz vorgehalten werden bzw. nach Erstellung der Zielsequenzen rekonstruierbar sein.

Ermittlung von Instanzen des Aufgabenmodells (Zielsequenzen)

Ausführungszeiten werden für einen bestimmten Weg durch das Aufgabenmodell, d.h. eine bestimmte Sequenz von Zielen berechnet. Um diese Berechnung vorneh-

men zu können, müssen folglich zunächst alle gültigen Zielsequenzen ermittelt werden, die sich aus dem Aufgabenmodell ergeben.

Berücksichtigung der in Kapitel 5.2.2 erläuterten Constraints

Bei der Erstellung der Zielsequenzen sind ggfs. definierte Constraints zwischen Zielen oder bzgl. eines Ziels zu beachten. Diese Constraints können bestimmte Zielsequenzen verbieten und dadurch den Lösungsraum mit gültigen Zielsequenzen einschränken. Da einige der definierten Arten von Constraints (s. Kapitel 5.2.2) bestimmte Zeitangaben beinhalten, für die einzelne Ziel-Ausführungszeiten bereits bekannt sein müssen, ist eine sorgfältige Planung der Reihenfolge, mit der die Constraints innerhalb des Zeitanalyseprozesses überprüft werden notwendig. Da das Constraint *verschachtelt* einen sehr komplexen und völlig anderen Typ als alle anderen hier betrachteten Constraints darstellt, der starke Auswirkungen auf die Konzeption der gesamten Constraint-Überprüfung hat, erfordert dessen Berücksichtigung eine umfangreiche separate Planung, die im Rahmen dieser Arbeit nicht durchführbar ist. Deshalb wird die Betrachtung dieses Constraint aus dem im weiteren Teil dieser Arbeit vorgestellten konkreten Konzept ausgenommen.

Berechnung der Ausführungszeiten von Zielsequenzen

Pro einzelner Zielsequenz muss deren voraussichtliche Ausführungszeit berechnet werden. Bevor diese Berechnung möglich ist, muss festgelegt werden, welche Prozesse einer Zielsequenz Zeit beanspruchen und wieviel Zeit dies jeweils genau ist. Dies wurde bereits in Kapitel 5.2.3 vorgenommen.

Ermittlung von maximalen, minimalen und durchschnittlichen Ausführungszeiten mit entsprechenden Zielsequenzen

Nachdem die Ausführungszeiten der Zielsequenzen berechnet wurden, sollen darunter die minimale, maximale und durchschnittliche Ausführungszeit ermittelt werden. Da es möglich und wahrscheinlich ist, dass der Durchschnittswert durch keine Zielsequenz erreicht werden kann, wird stattdessen der Median der Ausführungszeiten berechnet. Zusätzlich sind pro Minimum, Maximum und Median

jene Zielsequenzen festzustellen, für die die entsprechende Ausführungszeit berechnet wurde. Minimale, Maximale und mediane Ausführungszeit in Verbindung mit den entsprechenden Zielsequenzen bilden das Hauptergebnis einer Zeitanalyse mit PEDTime.

Um PEDTime erfolgreich in den Procedure Editor integrieren zu können, ist dieser um einige Funktionen zu erweitern. Der Vollständigkeit halber werden im Folgenden die wichtigsten Anforderungen an PED aufgeführt, die zur Integration von PEDTime zu beachten sind, auch wenn diese Integration nicht Teil dieser Arbeit ist und deshalb im Rahmen dieser Arbeit nicht weiter verfolgt wird.

Einbindung der Regelsystem-Erweiterung in PED

Die Prozeduren-Syntax des Regelsystems von PED muss um die in Kapitel 5.2.3 vorgestellte Erweiterung ergänzt werden. Dadurch werden die Definition von Constraints, sowie die Festlegung der Ausführungszeiten einzelner Operatoren ermöglicht, die nötig sind, um alle Funktionen einer Zeitanalyse mit PEDTime nutzen zu können.

Aufruf der Zeitanalyse-Funktion aus der GUI

Die Zeitanalyse soll aus der GUI von PED aufgerufen werden können. Aus diesem Grund muss eine Funktion zur Aktivierung der Zeitanalyse in die aktuelle GUI des Procedure-Editors integriert werden.

Anzeige des Ergebnisses in der GUI

PEDTime gibt als Ergebnis maximale, minimale und mediane Ausführungszeiten mit entsprechenden Zielsequenzen zurück. Außerdem werden ggfs. ausgewählte Variablenbelegungen, die zu der jeweiligen Zielsequenz geführt haben, zurückgegeben. In der GUI von PED soll dieses Ergebnis sinnvoll und übersichtlich präsentiert werden.

Konfiguration von Zeitparameterwerten

Sind alle Prozesse identifiziert, die bei der Abarbeitung eines Weges durch das Aufgabenmodell, d.h. einer Zielsequenz Zeit kosten, so sollen deren Zeitwerte in der GUI von PED konfigurierbar sein. Je nach Art des Kostenverursachers (z.B. Operator) ist zu entscheiden, ob der Nutzer diese Zeitwerte z.B. global für eine ganze Prozedur oder individuell pro Ziel bzw. Regel festlegen können soll.

6.1.3 Abbildung auf ein Ablaufplanungsproblem

Die in den vorherigen Abschnitten erläuterten Anforderungen skizzieren die einzelnen Probleme, die bei der Konzipierung einer PEDTime-Zeitanalyse zu lösen sind. Es ist möglich einige dieser Probleme zusammengefasst als ein Ablaufplanungsproblem zu interpretieren, worauf im Folgenden genauer eingegangen wird.

Zur Berechnung von Ausführungszeiten müssen zunächst Ablaufpläne erstellt werden, die die Aufgaben bzw. Ziele des betrachteten Aufgabenmodells in eine gültige Anordnung bringen. Ob eine Anordnung gültig ist, hängt einerseits von der Struktur der Aufgabenhierarchie und andererseits von ggfs. definierten zeitlichen Bedingungen zwischen den Zielen (Constraints) ab. Für jede dieser gültigen Anordnungen kann anschließend die erwartete Ausführungszeit berechnet werden. Schließlich sollen unter allen gültigen Anordnungen jeweils jene ermittelt werden, für die die kürzeste, längste und mediane Ausführungszeit berechnet wurde.

Dieses Problem kann als ein Ablaufplanungsproblem angesehen werden. Die Ablaufplanung, genannt *Scheduling*, befasst sich mit der Erstellung von Ablaufplänen, bei denen Aktivitäten zeitlich zu limitierten Ressourcen zugeordnet werden. Dabei können Nebenbedingungen definiert sein, die berücksichtigt werden müssen und es kann ein bestimmtes Ziel festgelegt sein, welches erreicht oder optimiert werden soll. Ein Ablaufplanungsproblem wird gelöst, indem ein Ablaufplan gefunden wird, der alle definierten Nebenbedingungen und Zielsetzungen erfüllt.

Ein allgemeines Ablaufplanungsproblem kann wie folgt auf das oben erläuterte Problem in dieser Arbeit abgebildet werden:

- Ziele stellen Aktivitäten dar, die zeitlich angeordnet werden.
- Neben der Zeit sind hier keine limitierten Ressourcen zu berücksichtigen.

- Die durch die Aufgabenstruktur vorgegebene Hierarchie (UND-/ODER-Verknüpfungen), und Constraints, die zwischen Zielen definiert sein können stellen Nebenbedingungen dar.
- Ein einziges Ziel kann nicht festgelegt werden, denn es gibt drei Ziele, die erreicht werden sollen: Ermittlung der schnellsten, längsten und medianen Ablaufpläne.

Der Lösung von Ablaufplanungsproblemen wurde insbesondere in den Bereichen des Operations Research[2] (OR) und der Künstlichen Intelligenz (KI) nachgegangen. Beim OR steht die Optimierung von definierten Zielsetzungen, d.h. Zielfunktionen im Vordergrund. Problematisch ist, dass schon einfache Problemstellungen NP-hart sind, d.h. dass ein Algorithmus zur exakten Lösung des Problems, sofern er existiert, mindestens exponentielle Laufzeit hat. Deshalb werden hier i.d.R. reale Probleme zu idealisierten Problemen abstrahiert (z.B. *Job-Shop-* oder *Flow-Job-Scheduling*). Dies führt allerdings dazu, dass die Lösungen dieser Ergebnisse für reale Probleme nicht ausreichend genau bzw. nur schwer nutzbar sind. Zur Lösung von Ablaufplanungsproblemen werden hier am häufigsten einfache Verfahren, wie z.B. Netzplantechnik oder Prioritätsregeln angewendet. Aber auch z.B. lineare Optimierung, sowie einige heuristische Verfahren, auf die im folgenden Absatz kurz eingegangen wird, sind im OR anzusiedeln (Sauer, 2004).

Der entsprechende Bereich in der KI ist jünger und versucht, Probleme mit relevanter praktischer Anwendung mit Hilfe neuer Lösungsverfahren zu lösen. Diese neuen Verfahren können z.B. spezielles Wissen über die Anwendungsdomäne des Problems für den Lösungsprozess berücksichtigen und diesen dadurch ggfs. beschleunigen. Zu diesen Verfahren gehören beispielsweise heuristische Verfahren, Constraint-basierte Verfahren und Verfahren des Soft Computing. Unter die heuristischen Verfahren fallen die heuristische Suche und allgemeine Heuristiken, wie z.B. Problemzerlegung. Bekannte heuristische Suchalgorithmen sind z.B. A^*, sowie IDA* als Kombination von iterativer Tiefensuche und A^*. Zur Unterstützung einer heuristischen Suche werden auch Constraint-basierte Verfahren genutzt, die zu den am häufigsten angewendeten Lösungsverfahren bei Ablaufplanungsproblemen gehören (Sauer, 2004). Ein Ablaufplanungsproblem wird bei diesen Verfahren als ein „Constraint-Satisfaction Problem“ (CSP) interpretiert, bei dem es das Ziel ist, eine Lösung für ein durch Bedingungen (Constraints) eingeschränktes Problem

[2]dt.: Unternehmensforschung, Ablauf- und Planungsforschung

zu finden, die alle definierten Bedingungen erfüllt. Die Lösung wird dabei durch die bestimmte Belegung von Variablen erreicht. Ein CSP hat eine Lösung, wenn jeder Variablen ein Wert zugewiesen werden kann, sodass alle Constraints gleichzeitig erfüllt sind. Zur Lösung eines CSPs werden u.a. simple Suchalgorithmen wie *Backtracking* oder intelligentere Methoden wie z.B. *Forward Checking* verwendet, die die Propagierung von Constraints[3] zur Einschränkung des Suchraums nutzen (Kumar, 1992; Tsang, 1993). Zur Lösung von Ablaufplanungsproblemen mit Hilfe von Verfahren des Soft Computing wird u.a. auf Fuzzy-Techniken, künstliche Neuronale Netze, iterative Verbesserungstechniken und genetische Algorithmen zurückgegriffen (Sauer, 2004).

In diesem Abschnitt wurde ein kurzer Überblick über Ablaufplanungsprobleme und einige dafür existierende Lösungsansätze gegeben. Darüber hinaus gibt es weitere Verfahren, die zur Lösung dieser Probleme eingesetzt werden. Die Einarbeitung in die große Anzahl und Vielfalt der teilweise vorgestellten bestehenden Lösungsansätze, sowie die anschließende Identifikation eines geeigneten Lösungsverfahrens, sofern es für das spezielle Problem dieser Arbeit existiert, erfordern eine zusätzliche umfangreiche Recherche, die im Rahmen dieser Arbeit nicht vorgenommen werden kann. Deshalb wird an dieser Stelle nicht konkreter auf Methoden zur Lösung von Ablaufplanungsproblemen eingegangen und im weiteren Verlauf dieser Arbeit ein eigener Ansatz zur Lösung des vorgestellten Problems verfolgt.

6.2 Konzeptbeschreibung von PEDTime

Basierend auf den vorgestellten Anforderungen wird in diesem Kapitel das konkrete Vorgehen bei der Zeitanalyse mit PEDTime dargelegt.

6.2.1 Vorüberlegungen

Bevor das Konzept detailliert vorgestellt wird, sind einige Vorüberlegungen festzuhalten, auf die im Folgenden eingegangen wird.

[3] engl.: Constraint Propagation

Überprüfung der Constraints während der Zeitanalyse

Wie in Kapitel 5 erläutert, werden Constraints für Ziele der RHS einer Regel (*rhsGoals*) definiert, d.h. zwischen den *rhsGoals* unterschiedlicher Regeln werden keine Constraints festgelegt. Somit müssen Constraints für alle *rhsGoals* einer Regel überprüft werden. Für jene Constraints, die Reihenfolgebeziehungen ausdrücken reicht damit deren einmalige Überprüfung zu dem Zeitpunkt, an dem alle möglichen Reihenfolgen von Zielen einer Menge von *rhsGoals* erstellt werden aus. Mengenbezogene Constraints können erst überprüft werden, wenn die entsprechenden Ausführungszeiten der betroffenen Ziele vorliegen.

Darüber hinaus wird festgestellt, dass ein Weg durch das Aufgabenmodell, d.h. eine Zielsequenz immer gültig ist, wenn für kein Ziel dieser Zielsequenz ein Constraint definiert ist.

Zeitbenötigende Prozesse innerhalb des Aufgabenmodells

Die Berechnung der Ausführungszeiten erfordert die Identifikation aller zeitbenötigenden Prozesse innerhalb des Aufgabenmodells. Wie bereits in Kapitel 5.2.3 erwähnt, gehören dazu alle unterschiedlichen Arten von Operatoren, d.h. *memory_ store*, *percept*, *motor* und *memory_ read*. Darüber hinaus beanspruchen das Feuern einer Regel und das Auswählen eines Ziels, sowie das Auswerten einer Condition Zeit (Lüdtke et al., 2009; Anderson et al., 2004).

6.2.2 Konzept

Das in Kapitel 6.1 erläuterte Problem wird konzeptuell in zehn Schritten gelöst, die im Folgenden beschrieben werden.

1. Abbilden des PED-Aufgabenmodells in internes PEDTime-Format
2. Aufstellen der Regelhierarchie
3. Initiale Regelsequenzen erstellen und reihenfolgebezogene Constraints prüfen
4. Komplette Regelsequenzen erstellen und reihenfolgebezogene Constraints prüfen

5. Ausführungszeiten für alle Regeln berechnen
6. Überprüfung aller mengenbezogenen Constraints
7. Ausführungszeiten für alle Regelsequenzen berechnen
8. Minimale, maximale und mediane Ausführungszeiten aller Regelsequenzen berechnen
9. Beste, schlechteste und mittlere Regelsequenzen ermitteln
10. Erzeugen eines Ergebnisses

Abbilden des PED-Aufgabenmodells in internes PEDTime-Format (1)

Die mit PED modellierte Prozedur mit ihren Elementen wie Regeln, Conditions, Zielen und Operatoren muss auf das Datenmodell von PEDTime abgebildet werden, da sämtliche Berechnungen der Zeitanalyse auf diesem durchgeführt werden. Dazu gehört auch, dass alle für die Prozedur definierten Constraints vorgehalten werden.

Aufstellen der Regelhierarchie (2)

Die Prozedur, d.h. Zielhierarchie wird anhand der durch das Regelsystem definierten Regeln modelliert. In den Regeln sind alle relevanten Elemente, wie u.a. Conditions, Goals und deren Verknüpfungen untereinander definiert. Um die Anforderung zu erfüllen, ggfs. gewählte Variablenbelegungen nach Erstellung einer Zielsequenz rekonstruieren zu können (s. Kapitel 6.1.2), wird eine Zielsequenz auf eine Regelsequenz abgebildet. Diese Modellierungsentscheidung ist folgendermaßen zu begründen: Die Conditions, in denen die Variablenbelegungen abgefragt werden, sind innerhalb einer Regel definiert. Da ein Ziel durch mehrere Regeln beschrieben werden kann, kann es u.U. auf verschiedenen Wegen durch die Zielhierarchie und mit unterschiedlichen Variablenbelegungen erreicht werden. Die alleinige Angabe eines Ziels lässt demnach nicht eindeutig auf den Weg schließen, mit dem dieses Ziel erreicht wurde. Eine Regel hingegen referenziert hinsichtlich Conditions einen eindeutigen Weg durch die Zielhierarchie. Wird in einer Zielsequenz folglich jedes Ziel durch die Regel ersetzt, die den Weg zur Erreichung

dieses Ziels definiert, so können ebenfalls sämtliche Variablenbelegungen, die zu der Zielsequenz führen, benannt werden.

Um diese Ersetzung vornehmen, d.h. Regelsequenzen erstellen zu können, muss zuvor die Zielhierarchie um eine Regelhierarchie erweitert werden. Dazu wird jeder Regel eine Menge von RHS-Regeln zugeordnet, die alle Regeln zu den Zielen auf der RHS beinhaltet. Dadurch besitzt eine Regel neben Zielen (*rhsGoals*) auch entsprechende Regeln (*rhsRules*) auf ihrer RHS. Die dem Ziel auf der LHS (*lhsGoal*) einer Regel entsprechende Regel ist äquivalent mit der Regel selbst. Abbildung 6.1 veranschaulicht dieses Vorgehen anhand von *Rule1* des Szenarios (s. Kapitel 4). Die *rhsGoals Monitor* und *EditRoute* werden durch die Regeln *Rule2* und *Rule18* beschrieben, welche deshalb als *rhsRules* von *Rule1* definiert werden.

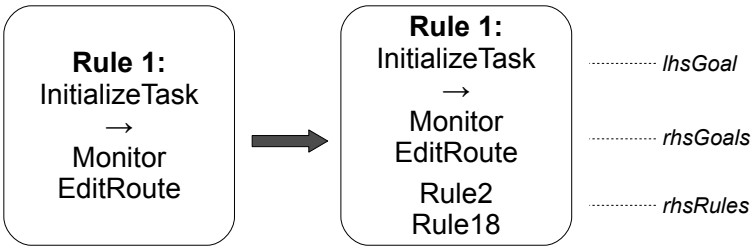


Abb. 6.1: Aufstellen der Regelhierarchie anhand eines Beispiels aus dem Szenario

Initiale Regelsequenzen erstellen und reihenfolgebezogene Constraints prüfen (3)

Zunächst werden alle gültigen Regelsequenzen erstellt, die aus Regeln der ersten und zweiten Hierarchieebene des Aufgabenmodells bestehen. In der ersten Hierarchieebene befinden sich die Wurzelregel und in der zweiten Hierarchieebene alle ihrer *rhsRules*. Bei der Erstellung der Regelsequenzen müssen ODER- und UND-Verknüpfungen, d.h. Disjunktionen und Konjunktionen der *rhsRules* berücksichtigt werden, damit nicht mehrere Regeln zu dem selben Ziel in derselben Regelsequenz enthalten sind. Für jede Menge von Regeln, die Disjunktionen und Konjunktionen berücksichtigt, werden anschließend sämtliche Permutationen der einzelnen Regeln erzeugt. Dabei wird bereits jede Permutation auf die Einhaltung von *nach*-Constraints geprüft. Anschließend wird für jede gültige Permutation eine neue Regelsequenz erstellt, deren erstes Element die Wurzelregel, gefolgt von der Permutation ist. Eine erneute Constraint-Überprüfung ist an dieser Stelle nicht

erforderlich, da die Wurzelregel laut Konzept in keiner Constraint-Beziehung zu einer ihrer *rhsRules* stehen kann (s. Kapitel 5). Zu beachten ist, dass an dieser Stelle lediglich die Reihenfolgebeziehungen, die durch die *nach*-Constraints definiert sind, beachtet werden. Eventuelle mengenbezogene Angaben in den Constraints können erst zu einem späteren Zeitpunkt (6), wenn Ausführungszeiten vorliegen, berücksichtigt werden.

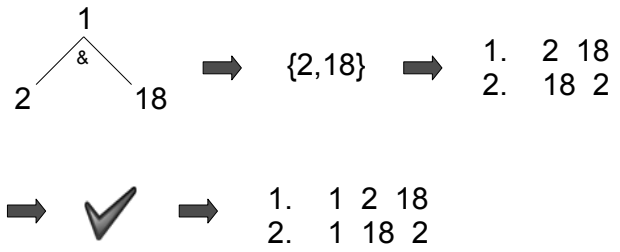


Abb. 6.2: Aufstellen der initialen Regelsequenzen im Szenario

Im Szenario (s. Abbildung 6.2) ist *Rule1* die Wurzelregel mit ihren per Konjunktion verbundenen *rhsRules* *Rule2* und *Rule18*[4]. Für die Menge {2,18} existieren zwei Permutationen. Diese verstoßen gegen keine *nach*-Constraints, da für kein Ziel, welches durch die permutierten Regeln beschriebenen wird, ein *nach*-Constraint definiert ist (s. Kapitel 4.2). Damit erfüllen auch die vollständigen Regelsequenzen alle bisher überprüfbaren Constraints, sodass für das Szenario insgesamt zwei initiale Regelsequenzen erstellt wurden.

Komplette Regelsequenzen erstellen und reihenfolgebezogene Constraints prüfen (4)

Entsprechend der initialen Erstellung der Regelsequenzen werden diese nun immer weiter um die entsprechenden Unterregeln ergänzt, bis alle zu einer – zur Regelsequenz gehörenden – Regel definierten *rhsRules* ebenfalls in dieser Regelsequenz enthalten sind. Genau dann ist die unterste Ebene der Zielhierarchie erreicht, in der sich ausschließlich elementare Aktionen, d.h. Operatoren befinden. Eine solche, komplette Regelsequenz repräsentiert schließlich einen vollständigen und eindeutigen Weg durch die Zielhierarchie.

[4] Auf Grund der Übersichtlichkeit sind die einzelnen Regeln in dieser und weiteren Abbildungen meistens nur über ihre Kennnummer dargestellt

Um alle kompletten Regelsequenzen erstellen zu können, sind mehrere Teilschritte erforderlich, die im Folgenden erläutert werden.

Da die *rhsRules* einer Regel einen Teil der nächsttieferen Ebene der Zielhierarchie repräsentieren, wird zunächst wie bei der Erstellung der initialen Regelsequenzen vorgegangen. Es werden ODER- und UND-Verknüpfungen der *rhsRules* berücksichtigt und entsprechende Regelmengen erstellt, welche anschließend permutiert werden. Jede generierte Permutation wird sofort auf Einhaltung definierter *nach*-Constraints überprüft. Alle gültigen Permutationen werden der Ausgangsregel in Form einzelner Regelsequenzen zugeordnet. Dieses Vorgehen wird anschließend solange für die *rhsRules* der zuletzt betrachteten *rhsRules* fortgeführt, bis schließlich allen Regeln der gesamten Prozedur entsprechende Regelsequenzen zugeordnet sind. Ausgenommen davon ist die Wurzelregel, da deren *rhsRules* bereits im vorherigen Schritt (3) behandelt wurden.

Nachdem sämtlichen Regeln die permutierten Regelsequenzen entsprechend obigem Vorgehen zugeordnet wurden, folgt der Aufbau von Teil-Regelsequenzen, die bereits einen Weg durch mehrere Hierarchien des Aufgabenmodells beschreiben. Dieser Teilschritt stellt den wichtigsten und komplexesten Schritt zum Aufbau der Regelsequenzen dar.

Mit Hilfe der permutierten Regelsequenzen des vorherigen Schritts werden Teil-Regelsequenzen erzeugt. Als Grundlage für den Aufbau der Teil-Regelsequenzen dienen sämtliche permutierten Regelsequenzen *rs1* der Ausgangsregel. In diese Regelsequenzen werden nun für jede darin enthaltene Regel r wiederum deren permutierte Regelsequenzen *rs2* an der Stelle direkt nach der entsprechenden Regel r in die Regelsequenz eingefügt. Da auch sämtliche permutierten Regelsequenzen der Regel r berücksichtigt werden müssen, muss dieser Schritt für alle Regelsequenzen *rs2* wiederholt werden, sodass sich die Anzahl entstehender Regelsequenzen vervielfacht (s. Kapitel 7.5). Abbildung 6.3 illustriert dieses Vorgehen anhand des Szenarios, ausgehend von *Rule2* als Ausgangsregel. Auf Grund entsprechender Constraints ist *Rule2* nur eine permutierte Regelsequenz zugeordnet. *Rule3* sind zwei permutierte Regelsequenzen zugewiesen. Diese werden nun direkt nach dem Auftreten von *Rule3* in der *Rule2* zugeordneten permutierten Regelsequenz in dieselbe eingefügt, sodass zwei neue Regelsequenzen entstehen. In diese neuen Regelsequenzen werden anschließend die permutierten Regelsequenzen für die nächste Regel *Rule4* an entsprechender Stelle eingefügt, sodass schließlich

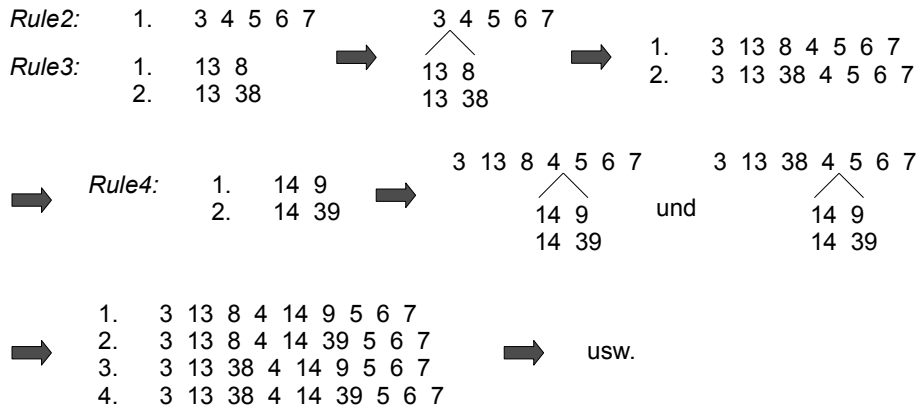


Abb. 6.3: Aufstellen der Teil-Regelsequenzen im Szenario

vier Regelsequenzen entstehen. Dies wird nun für alle weiteren Regeln, d.h. *Rule5*, *Rule6* und *Rule7* fortgeführt.

Dieses Vorgehen wird solange in die Tiefe der Hierarchie fortgeführt, bis in keiner Regelsequenz mehr eine Regel existiert, deren *rhsRules* nicht ebenfalls in der Regelsequenz enthalten sind. Dann ist die unterste Hierarchieebene des Aufgabenmodells erreicht. Die erste Ausgangsregel ist die erste Regel der *rhsRules* der Wurzelregel. Für alle weiteren *rhsRules* der Wurzelregel wird entsprechend verfahren, sodass schließlich zu allen *rhsRules* der Wurzelregel, d.h. ab der 2. Hierarchieebene sämtliche Teil-Regelsequenzen existieren.

Nachdem die Teil-Regelsequenzen erstellt wurden, werden diese in die initialen Regelsequenzen eingefügt, wodurch letztlich die kompletten Regelsequenzen entstehen. Als Basis für die Erstellung der kompletten Regelsequenzen dienen alle initialen Regelsequenzen (3). Für jede darin enthaltene Regel werden nun, äquivalent zu der Erstellung der Teil-Regelsequenzen, an der Stelle direkt nach der entsprechenden Regel deren Teil-Regelsequenzen eingefügt. Am Ende dieses Schrittes liegen alle kompletten Regelsequenzen des Aufgabenmodells vor, die bereits auf Einhaltung reihenfolgebezogener Constraints geprüft wurden.

Ausführungszeiten für alle Regeln berechnen (5)

Um die Ausführungszeiten für alle einzelnen ermittelten Regelsequenzen berechnen zu können, werden zunächst die Ausführungszeiten aller Regeln des Aufgabenmodells berechnet.

Da eine Regel stets den Weg zur Erreichung eines Ziels (*lhsGoal* der Regel) festlegt, tragen alle an diesem Weg beteiligten Prozesse zur tatsächlichen Ausführungszeit des entsprechenden Ziels bei. Alle in Abschnitt 6.2.1 aufgeführten zeitbenötigenden Prozesse können in einer Regel definiert sein. Für die Ausführungszeiten dieser Prozesse werden folgende Standardwerte angenommen:

memory_ store	200ms
percept	200ms
motor	1300ms
memory_ read	600ms
Zielauswahl bzw. Regelfeuern	50ms
Auswerten einer Condition	*memory_ read* $*$ Anzahl Variablen in Condition

Tab. 6.1: Standard-Ausführungszeiten für Operatoren und indirekte Prozesse

Die Ausführungszeiten für die einzelnen Prozesse wurden den Ansätzen ACT-R (Anderson et al., 2004) und KLM-GOMS (Kieras, 2001) entnommen. Die Zeit für den *motor*-Operator bezieht sich auf einen Tastendruck, da alle im Szenario auftretenden *motor*-Aktionen Tastendrücke repräsentieren. Sie setzt sich aus den Durchschnittswerten für das Positionieren des Fingers (1100ms), sowie für das eigentliche Drücken und Loslassen (200ms) zusammen (Kieras, 2001).

Die Auswertung einer Condition beinhaltet implizit den Operator *memory_ read*. Es wird angenommen, dass pro Condition lediglich eine Variable verwendet wird, sodass die Auswertung einer Condition mit o.g. Zeiten stets 600ms beträgt. Dies ist auf die aktuelle Repräsentation einer Condition in PED zurückzuführen, wo eine Condition stets aus einer Variable, einem mathematischen Operator und einem Wert besteht, d.h. dass mehrere Variablen zum aktuellen Zeitpunkt nicht modellierbar sind.

Für jede Regel des Aufgabenmodells wird nun, gemäß der Existenz o.g. Prozesse innerhalb dieser Regel ihre Ausführungszeit berechnet. Dabei ist zu beachten, dass sich diese Ausführungszeit ausschließlich auf Prozesse, die direkt Teil der Regel selbst sind, bezieht. Hat eine Regel folglich *rhsGoals*, die keine Aktionen definieren bzw. noch weiter aufgelöst werden können, sind deren Ausführungszeiten nicht in der Ausführungszeit der Regel berücksichtigt. Dieses Vorgehen vereinfacht die

Berechnung der Ausführungszeiten ganzer Regelsequenzen, wie sie in Schritt 7 erläutert wird.

Nachdem die jeweilige Ausführungszeit berechnet wurde, wird geprüft, ob ggfs. Constraints, die die Dauer des *lhsGoals* einer Regel einschränken, verletzt werden. Diese Überprüfung kann an dieser Stelle ausschließlich für jene Regeln vorgenommen werden, deren *rhsGoals* Aktionen, d.h. Operatoren definieren (s.o.). Wird dabei festgestellt, dass ein für ein bestimmtes Ziel definiertes *Dauer*-Constraint verletzt wird, dann sind alle Regelsequenzen, die die zu diesem Ziel gehörende Regel enthalten, ungültig. Dies beruht auf der Überlegung, dass die Zeiten zur reinen Ausführung von Operatoren nicht optimiert werden können und somit unveränderlich sind. Die Ausführungszeit der entsprechenden Regel kann demnach nicht verkürzt werden, um eine Verletzung des entsprechenden Constraints zu vermeiden. In diesem Fall kann es nur eine Regel geben, die das entsprechende Ziel definiert, da nur jene Ziele betrachtet werden, die ausschließlich Aktionen als *rhsGoals* definiert haben und diese Ziele stets über eine eindeutige Ausführung von Aktionen erreicht werden. Alle ungültigen Regelsequenzen werden in den folgenden Schritten nicht weiter betrachtet.

Überprüfung aller mengenbezogenen Constraints (6)

Bevor die Ausführungszeiten der Regelsequenzen berechnet werden, ist es sinnvoll, erst alle mengenbezogenen Constraints zu überprüfen, um von vornherein keine Zeiten für ungültige Regelsequenzen zu berechnen.

Um *Dauer*-Constraints aller Ziele, d.h. auch jenen, die weder auf der obersten, noch auf der untersten Hierarchie des Aufgabenmodells angesiedelt sind, überprüfen zu können, reichen weder die in Schritt 5 berechneten Zeiten noch die Ausführungszeiten der kompletten Regelsequenzen aus. Deshalb müssen an dieser Stelle die Ausführungszeiten der zu den per Constraint eingeschränkten Zielen gehörenden Regeln separat ermittelt werden.

Zur Überprüfung werden alle Regelsequenzen, die die zum Ziel gehörende Regel bzw. gehörenden Regeln enthalten, ermittelt. In diesen Regelsequenzen werden die Ausführungszeiten derjenigen zur Regel gehörenden Unterregeln aufsummiert, die – neben anderen Regeln – rechts von der Regel in der Regelsequenz stehen. Als Unterregeln gelten hier alle Regeln, die Ziele beschreiben, die zur Erfüllung

des Ziels, das durch die betrachtete Regel beschrieben wird, beitragen. Dabei ist zu beachten, dass ausschließlich diejenigen Unterregeln der Regel betrachtet werden, die tatsächlich in der jeweiligen Regelsequenz enthalten sind. Nachdem die Ausführungszeiten aufsummiert wurden, liegt die Gesamt-Ausführungszeit der zu überprüfenden Regel innerhalb der jeweiligen Regelsequenz vor, sodass jetzt die Überprüfung dieser Regel vorgenommen werden kann. Ist ihre Ausführungszeit länger als durch das Constraint erlaubt, ist die betrachtete Regelsequenz ungültig und wird in den Folgeschritten nicht weiter berücksichtigt.

Da die Ausführungszeiten der einzelnen Regeln bereits berechnet wurden, können jetzt auch alle weiteren mengenbezogenen Constraints wie *max T nach*, *min T nach* und *genau T nach* überprüft werden. Dazu werden alle Regelsequenzen ermittelt, die jene Regeln beinhalten, die die vom Constraint betroffenen Ziele definieren. Pro Regelsequenz wird dabei stets ein vom Constraint betroffenes Regelpaar betrachtet. Falls in der jeweiligen Regelsequenz zwischen den beiden betroffenen Regeln (inkl. deren Unterregeln) weitere Regeln liegen, so muss deren Gesamt-Ausführungszeit berechnet werden. Dazu werden die Ausführungszeiten aller zwischen den beiden betroffenen Regeln (inkl. deren Unterregeln) liegenden Regeln aufsummiert. Anschließend wird diese berechnete Zeit mit der Zeit *T* des Constraints verglichen. Erfüllt die berechnete Zeit das Constraint, so ist die jeweilige Regelsequenz gültig, andernfalls ist sie ungültig und wird in den folgenden Schritten nicht weiter betrachtet. Tabelle 6.2 zeigt, in welchen Fällen eine Regelsequenz gültig ist.

Constraint	**Bedingung für gültige Regelsequenz**
max T nach	Berechnete Zeit ist kleiner oder gleich T
	Zwischen Regelpaar liegen keine anderen Regeln
min T nach	Berechnete Zeit ist größer oder gleich T
genau T nach	Berechnete Zeit ist gleich T

Tab. 6.2: Bedingungen für gültige Regelsequenzen bzgl. mengenbezogener Constraints

Die Abbildung 6.4 veranschaulicht das Vorgehen anhand eines Beispiels aus dem Szenario. Zu sehen ist ein Ausschnitt aus einer Regelsequenz[5]. Die Ziele *MonitorD* und *MonitorC* sind durch ein *max T nach*-Constraint eingeschränkt. Zwischen

[5]Diese Regelsequenz ist auf Grund weiterer Constraints des Szenarios nicht gültig und wird hier lediglich zur Veranschaulichung des oben erläuterten Vorgehens verwendet.

den Regeln, die diese Ziele repräsentieren, liegen weitere Regeln, sodass deren Ausführungszeit gemäß oben erläutertem Verfahren ermittelt wird. Da diese Ausführungszeit kleiner ist, als die durch das Constraint vorgegebene maximale Zeit, die zwischen der Erreichung von *MonitorC* und der Verfolgung von *MonitorD* vergehen darf, ist die Regelsequenz gültig.

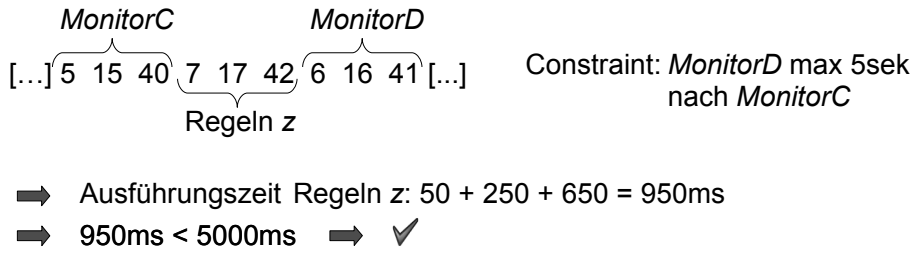


Abb. 6.4: Überprüfung des *max T nach*-Constraints anhand des Szenarios

Zu beachten ist, dass erst alle *min T nach* und *genau T nach*-Constraints überprüft sein müssen, bevor *max T nach*-Constraints überprüft werden können. Dies liegt am ggfs. notwendigen Einfügen von Pufferzeiten auf Grund ersterer Constraints, worauf im Folgenden genauer eingegangen wird.

Da für die Constraints stets gilt $T > 0$, ist zunächst außerdem jede Regelsequenz ungültig, wenn sie von den Constraints *min T nach* oder *genau T nach* betroffen ist und zwischen dem betroffenen Regelpaar keine weiteren Regeln liegen. Denn für diese Fälle müsste $T = 0$ gelten, um die Constraints zu erfüllen. Allerdings kann eine solche Regelsequenz unter bestimmten Umständen dennoch gültig werden. Wenn zwischen dem betroffenen Regelpaar keine weiteren Regeln inklusive deren Unterregeln liegen, kann zur Erfüllung des *min T nach* und des *genau T nach*-Constraints eine zusätzliche Pufferzeit der Länge T für die jeweilige Regelsequenz eingeplant werden, die die künstlich erzeugte Wartezeit T zwischen dem Regelpaar repräsentiert. Dadurch erfüllt die entsprechende Regelsequenz das jeweilige Constraint und bleibt gültig. Diese Pufferzeit wird dem betroffenen Regelpaar, repräsentiert durch eine zusätzliche Regelsequenz bestehend aus den beiden Regeln, die wiederum jeweils beiden Regeln zugeordnet wird, zugewiesen. Dies ist nötig, damit bei ggfs. zusätzlichen *max T nach*-Constraint-Überprüfungen vorher eingefügte Pufferzeiten korrekt für bestimmte Regelpaare berücksichtigt werden können. Dazu muss bei der Überprüfung der *max T nach*-Constraints zusätzlich auf evtl. Pufferzeiten geachtet werden, die Regeln zugeordnet sind, die zwischen

den vom Constraint betroffenen Regeln in der Regelsequenz liegen. Dieses Vorgehen wird anhand des Beispiels[6] in Abbildung 6.5 verdeutlicht. Die Pufferzeiten fließen außerdem später in die Ausführungszeiten der Regelsequenzen (Schritt 7) ein.

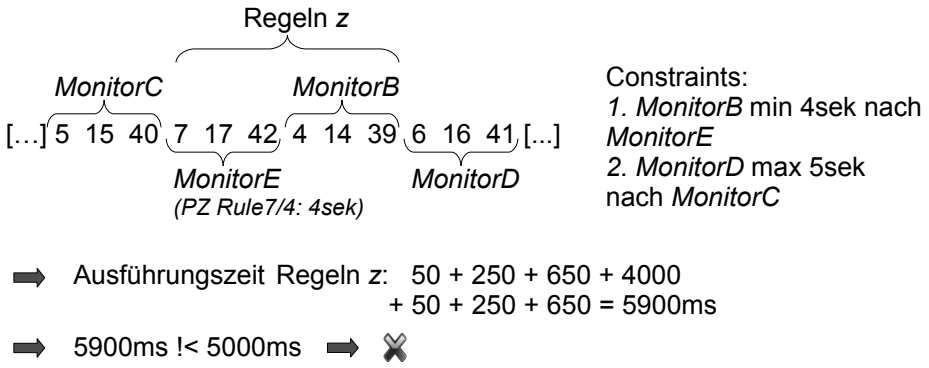


Abb. 6.5: Überprüfung von *min T nach*- und *max T nach*-Constraints anhand des Szenarios

Die Abbildung 6.5 greift das vorherige Beispiel auf und erweitert dies um weitere Regeln, sowie ein weiteres *min T nach*-Constraint. Auf Grund des *min T nach*-Constraints wird eine Pufferzeit (PZ) zwischen den Zielen *MonitorB* und *MonitorE* eingefügt, die durch die (Ober-)Regeln *Rule4* und *Rule7* repräsentiert werden. Die Ausführungszeit der Regeln *z* verlängert sich dadurch um 4000ms. Das *max T nach*-Constraint wird deshalb nicht erfüllt, sodass die Regelsequenz ungültig ist. Das Beispiel verdeutlicht die Notwendigkeit der eindeutigen Zuordnung von Pufferzeiten und deren korrekte Berücksichtigung bei der Constraint-Überprüfung. Wäre die Pufferzeit hier nicht beachtet worden oder das *min T nach*-Constraint erst nach Überprüfung des *max T nach*-Constraints berücksichtigt worden, hätte die Gesamt-Überprüfung der Regelsequenz zu einem falschen Ergebnis geführt.

Ausführungszeiten für alle Regelsequenzen berechnen (7)

Nachdem die Ausführungszeiten aller einzelnen Regeln bekannt sind und alle Constraints überprüft wurden, können nun die Ausführungszeiten der Regelsequenzen berechnet werden. Dazu werden die einzelnen Ausführungszeiten aller in der Regelsequenz enthaltenen Regeln aufsummiert. Dies ist möglich, da sich

[6] Dieses Beispiel ist so nicht Teil des Szenarios, sondern ausschließlich zur Veranschaulichung des oben erläuterten Vorgehens gewählt worden.

die Ausführungszeiten der einzelnen Regeln ausschließlich aus deren direkten Bestandteilen zusammensetzen (s.o.), sodass keine Zeiten mehrfach in die Gesamt-Ausführungszeit einer Regelsequenz einfließen. Außerdem werden evtl. Pufferzeiten innerhalb einer Regelsequenz zu deren Ausführungszeit addiert.

Minimale, maximale und mediane Ausführungszeiten aller Regelsequenzen berechnen (8)

Unter allen berechneten Ausführungszeiten für die gültigen Regelsequenzen werden nun die kürzesten (minimalen), längsten (maximalen) und medianen Ausführungszeiten ermittelt. Die mediane Ausführungszeit ist die tatsächlich durch mindestens eine Regelsequenz erreichte mittlere Ausführungszeit. Bei einer ungeraden Anzahl unterschiedlicher Ausführungszeiten existiert genau ein Median. Liegt eine gerade Anzahl vor, so gibt es zwei Mediane. Alle Mediane repräsentieren gültige mittlere Ausführungszeiten und sind damit Teil der Lösung einer Zeitanalyse mit PEDTime.

Beste, schlechteste und mittlere Regelsequenzen ermitteln (9)

Zu den im vorherigen Schritt (8) ermittelten Ausführungszeiten gehören bestimmte Regelsequenzen, die zu diesen Ausführungszeiten führen. Um diese Regelsequenzen zu ermitteln, werden separat alle gültigen Regelsequenzen festgestellt, die zu der maximalen (schlechtesten), der minimalen (besten) und der bzw. den medianen (mittleren) Ausführungszeit(en) führen. Alle diese besten, schlechtesten und medianen Regelsequenzen sind ebenfalls Teil der Lösung einer PEDTime-Zeitanalyse.

Erzeugen eines Ergebnisses (10)

Nachdem alle für das Ergebnis einer Zeitanalyse mit PEDTime benötigten Teil-Lösungen vorliegen, können diese zu einem Ergebnis zusammengefügt werden. Das Ergebnis beinhaltet mindestens folgende Informationen:

- Minimale, maximale und mediane Ausführungszeiten aller gültigen Regelsequenzen

- Alle Regelsequenzen, die zu den minimale, maximalen und der bzw. den medianen Ausführungszeit(en) führen
- Die durch die Regelsequenzen repräsentierten Zielsequenzen
- Ggfs. die zu setzenden Variablenwerte, die zu den optimalen und medianen Regelsequenzen führen
- Ggfs. eingefügte Pufferzeiten in bestimmten Regelsequenzen

Diese Informationen stellen die Ergebnisse der umfassenden zeitlichen Analyse eines bestimmten Aufgabenmodells durch PEDTime dar und ermöglichen u.a. die Ableitung von Optimierungsstrategien, z.B. hinsichtlich der Sicherheit oder der Usability des durch das Aufgabenmodell repräsentierten Systems.

6.3 Mögliche Probleme des Konzepts

Bei der Erstellung des Konzepts wurden einige Probleme bzw. genauer zu untersuchende Faktoren identifiziert, auf die im Folgenden eingegangen wird.

In Kapitel 5.1 wurde bereits auf die Problematik von reaktiven Regeln in Bezug auf Zeitanalysen hingewiesen. Da sie einen erheblichen Unsicherheitsfaktor darstellen, wurden sie von vornherein nicht in das PEDTime-Konzept aufgenommen. Dennoch erlaubt die Regelsprache von PED die Modellierung dieser reaktiven Regeln, sodass entschieden werden muss, wie mit solchen Regeln umgegangen wird, falls sie in einem mit PEDTime zu analysierenden Aufgabenmodell enthalten sind. Ggfs. können diese Regeln nicht einfach ignoriert werden, sondern sind zur Beibehaltung der Korrektheit des Modells in andere, mit einer Zeitanalyse sinnvoll zu vereinenden Konstrukte zu transferieren.

Ein weiteres Konzept, welches durch die PED-Regelsprache erlaubt wird, sind Zyklen innerhalb eines Aufgabenmodells. Dieses Konzept ist äquivalent nicht sinnvoll in PEDTime zu berücksichtigen. Zyklen erlauben theoretisch unendlich viele Durchläufe eines bestimmten Teils des Aufgabenmodells. Ein Weg durch das Aufgabenmodell, der über einen Zyklus führt, kann demnach ebenfalls unendlich sein. Somit ist die Erstellung von Regelsequenzen mit Zyklen und damit auch die Berechnung von deren Ausführungszeiten nicht möglich. Davon abgesehen erscheint

es rein faktisch nicht sinnvoll, Ausführungszeiten von Wegen, die Zyklen enthalten, zu berechnen, es sei denn die Anzahl möglicher Zyklendurchläufe ist festgelegt. Dann können sinnvolle und verwertbare Aussagen zu maximaler, minimaler und medianer Ausführungszeit getroffen werden.

In dem hier vorgestellten Konzept werden Zyklen vor der Zeitanalyse durch PEDTime aus dem Aufgabenmodell entfernt, sodass die entsprechende Regel, auf deren RHS der Zyklus definiert war dann ggfs. eine leere RHS hat. Auf das Szenario bezogen erscheint dies sinnvoll, da alle dort auftretenden Zyklen nur dann durchlaufen werden, wenn die benötigten Daten nicht oder fehlerhaft vorliegen. Die Wege, die durch die Zyklen beschrieben werden, sind demnach nicht fehlerfrei und damit ggfs. für die Berücksichtigung in einer Zeitanalyse unerwünscht. Auf dieses Problem wird nachfolgend genauer eingegangen.

Im Aufgabenmodell sind alle möglichen Wege modelliert, d.h. auch solche, die nicht den Weg zur korrekten Erreichung eines Ziels beschreiben, sondern Fehler während des Ablaufs berücksichtigen. Im Szenario sind derartige Wege meistens durch Abbruchaktionen, wie das Drücken eines REJECT-Buttons gekennzeichnet, die durch Fehler oder fehlende Informationen ausgelöst werden und stets den Weg durch einen Zyklus zur Folge haben. Die Frage, die sich dabei stellt ist, ob fehlerhafte Wege durch das Aufgabenmodell bei der Zeitanalyse berücksichtigt werden sollen oder nicht. Sicherlich kann auch die Ausführungszeit, die aus Fehlern resultiert gerade im sicherheitskritischen Umfeld eine wichtige Erkenntnis sein. Dennoch sollte vor der Zeitanalyse explizit entschieden werden, ob auch fehlerhafte Wege einbezogen werden, insbesondere dann, wenn diese Zyklen zur Folge haben, da diese die berechnete maximale Ausführungszeit entscheidend verfälschen bzw. deren Berechnung sogar unmöglich machen können.

In die Ausführungszeiten der Regeln fließt u.a. die Zeit zur Auswertung einer Condition ein. Dass für die Durchführung dieses Prozesses eine bestimmte Zeit eingeplant wird, basiert auf Untersuchungen anderer Ansätze, wie z.B. ACT-R (Anderson et al., 2004). In dieser Arbeit wurde das Regelsystem um die Modellierung zeitlicher Bedingungen, d.h. Constraints erweitert. Auf Grund fehlender Untersuchungen, wird für die Auswertung eines Constraints in diesem Konzept keine zusätzliche Zeit eingeplant. Es ist möglich, dass die fehlende Berücksichtigung dieser zusätzlichen Zeit zu weniger realistischen Ergebnissen bei der Zeitanalyse führt.

7 Algorithmische Umsetzung der Zeitanalyse

Dieses Kapitel stellt einen Entwurf zur Umsetzung der Kernelemente des im vorherigen Kapitel beschriebenen Konzepts vor. Neben der Programmstruktur des PEDTime-Prototyps und der Vorstellung aller bedeutenden Methoden wird auch auf die Implementierung ausgewählter Methoden eingegangen. Abgerundet wird das Kapitel durch die Darlegung eines Speicherproblems, das bei der vorgestellten Umsetzung auftreten kann.

7.1 Verwendete Technologien

Für die Umsetzung des vorgestellten Konzepts wurden die nachfolgend aufgeführten Technologien verwendet.

Java SE JDK 6

Die Implementierung wurde in der Programmiersprache Java vorgenommen und basiert auf „Java Platform, Standard Edition Java Development Kit 6“ (Oracle, 2010). Da PED ebenfalls in Java entwickelt wurde, ist so eine leichte Anbindung von PEDTime gewährleistet.

Eclipse IDE for Java Developers

Zur Implementierung der Software wurde die Entwicklungsumgebung „Eclipse IDE for Java Developers, Galileo Package“ (Eclipse Foundation, 2010) in der Version

1.2.2.20100216-1730 inklusive der Version 3.5.2 der „Eclipse Platform“ verwendet.

7.2 Paket- und Klassenstruktur von PEDTime

Um das in Kapitel 6 vorgestellte Konzept durch eine konkrete Implementierung umsetzen zu können, muss zunächst ein struktureller Entwurf von PEDTime vorgenommen werden, der im Folgenden beschrieben wird.

Die Abbildung 7.1 zeigt ein UML-Klassendiagramm, welches die Kommunikation zwischen PEDTime und PED skizziert. PEDTime besteht aus den Unterpaketen *logic*, *model*, *calculator* und *sample*. Das Paket *logic* beinhaltet die Logik von PEDTime, d.h. alle Methoden und Klassen, die direkt für einzelnen Berechnungen der Zeitanalyse zuständig sind. Die für die Zeitanalyse notwendigen unterschiedlichen Teilberechnungen wurden entsprechend auf unterschiedliche Klassen abgebildet. Diese Berechnungen werden auf Objekten des *model*-Pakets vorgenommen, welches das Modell von PEDTime kapselt. Das *calculator*-Paket stellt die Hauptschnittstelle zu PED dar, und seine Klasse *PedTimeCalculator* koordiniert die Durchführung der Zeitanalyse.

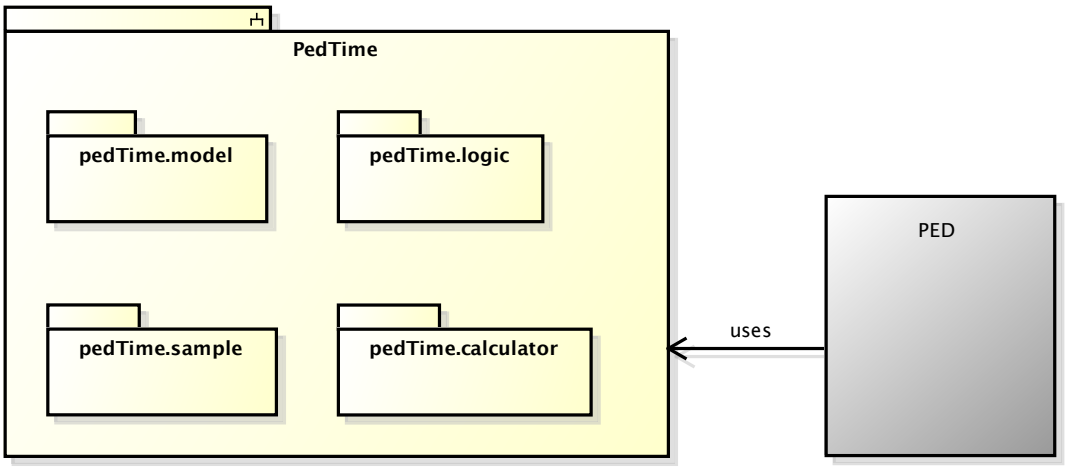


Abb. 7.1: UML-Klassendiagramm des PEDTime-Pakets

Die Erweiterung PEDTime kann von PED verwendet werden, indem aus PED heraus auf Klassen und Methoden der Unterpakete *calculator* und *model* zugegriffen wird. Zunächst wird ein Aufgabenmodell (*TaskModel*) des *model*-Pakets er-

stellt und initialisiert, welches entweder eine neue PED-Prozedur oder ein Beispiel-Aufgabenmodell aus dem *sample*-Paket beinhaltet. Dieses wird anschließend einer neuen Instanz der Klasse *PedTimeCalculator* übergeben, welche dafür sorgt, dass die Zeitanalyse durchgeführt und letztlich ein Ergebnis (*Result*) mit allen Ergebnissen der Zeitanalyse erzeugt wird.

Abbildung 7.2 veranschaulicht die Relationen zwischen den Unterpaketen *calculator*, *logic* und *model*[1] aus Sicht des *PedTimeCalculator*.

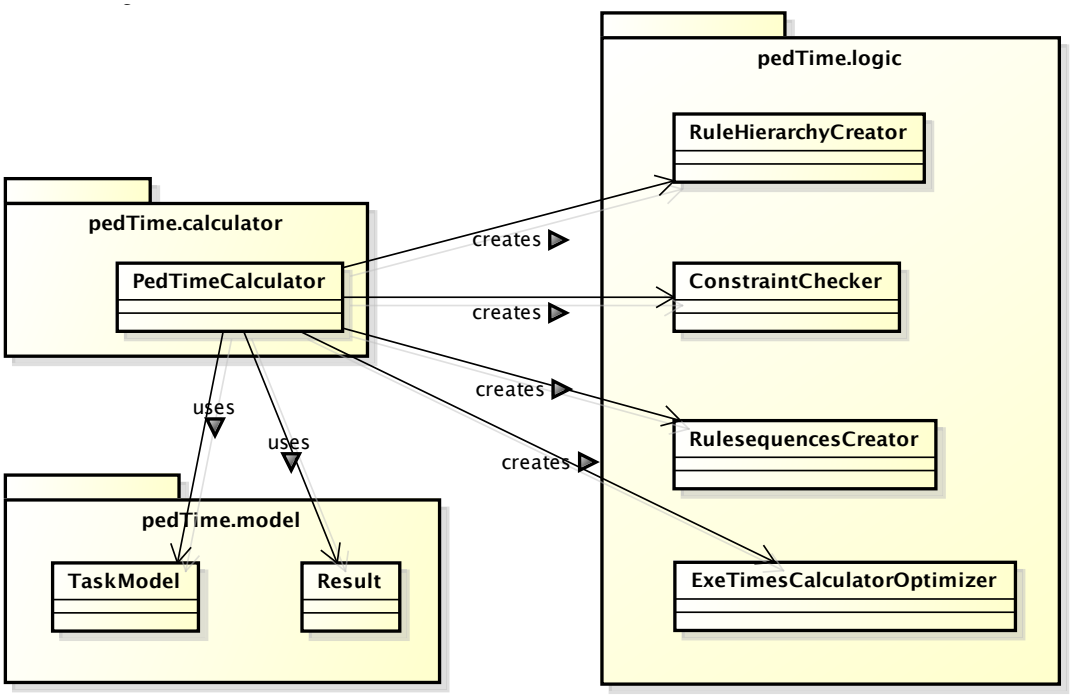


Abb. 7.2: UML-Klassendiagramm der Unterpakete *calculator*, *logic* und *model*

Der PEDTime-Berechner (*PedTimeCalculator*) führt, basierend auf einem Aufgabenmodell (*TaskModel*) die Zeitanalyse durch. Dazu erzeugt er als erstes einen Regelhierarchie-Ersteller (*RuleHierarchyCreator*), der anhand des Aufgabenmodells die Regelhierarchie konstruiert. Anschließend wird ein Constraint-Überprüfer (*ConstraintChecker*) generiert, der alle Methoden beinhaltet, die für die Überprüfung definierter Constraints erforderlich sind. Der Constraint-Überprüfer und das Aufgabenmodell werden vom darauffolgend erzeugten Regelsequenzen-Ersteller

[1] Zur besseren Übersicht wurden auf die Darstellung der für diesen Teil nicht relevanten Elemente des *model*-Pakets verzichtet.

(*RulesequencesCreator*) verwendet, der alle gültigen Regelsequenzen unter Einhaltung bestimmter Constraints aufbaut. Schließlich berechnet der Zeitenberechner und -optimierer (*ExeTimesCalculatorOptimizer*) die Ausführungszeiten aller gültigen Regelsequenzen, überprüft dabei weitere Constraints und ermittelt unter der endgültigen Menge gültiger Regelsequenzen jene mit optimaler und medianer Ausführungszeit. Der Zeitenberechner und -optimierer erstellt schließlich das Ergebnis (*Result*) der Zeitanalyse, welches über den PEDTime-Berechner an PED zurückgegeben wird. Das Klassendiagramm in Abbildung 7.3 skizziert die erläuterten Relationen zwischen Klassen der Pakete *logic* und *model*[1] aus Sicht des Logik.

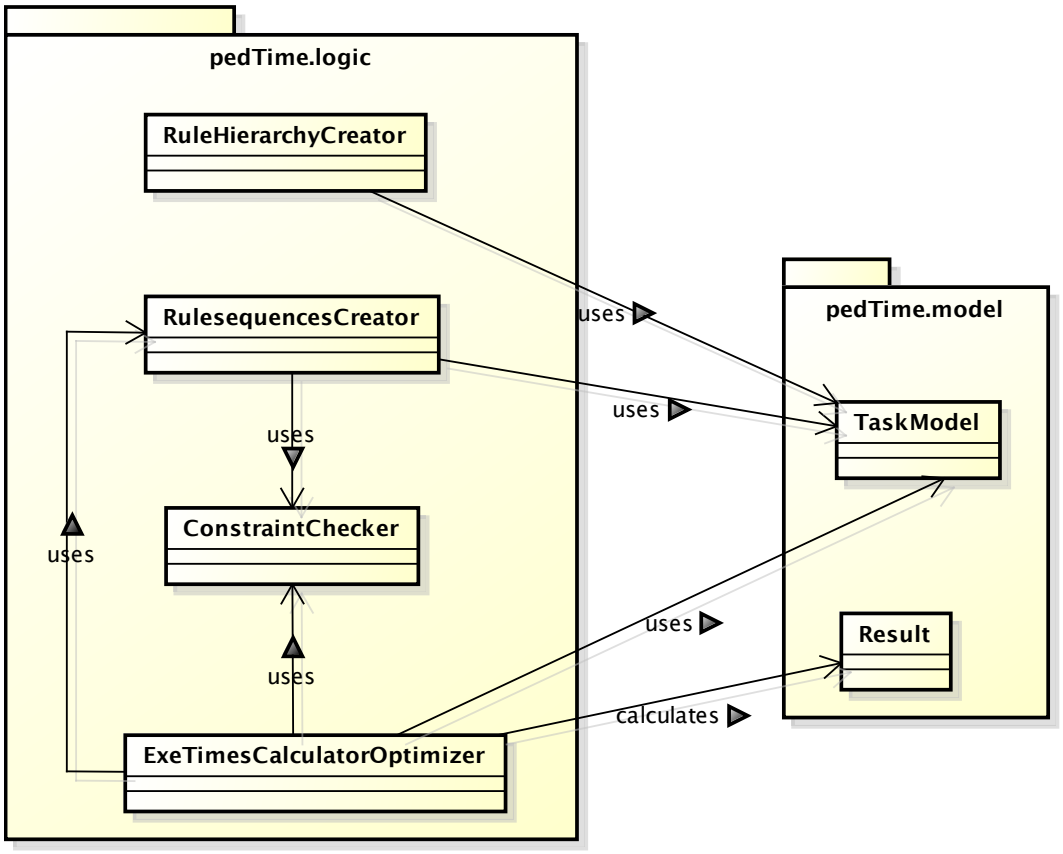


Abb. 7.3: UML-Klassendiagramm der Unterpakete *logic* und *model*

Die Bestandteile des Aufgabenmodells finden sich als Klassen in dem Paket *model* wieder, welches in Abbildung 7.4 anhand eines UML-Klassendiagramms dargestellt ist. Ein Aufgabenmodell (*TaskModel*) besteht aus Regeln (*Rule*) und Constraints (*Constraint*), die jeweils genau einer bestimmten Art von Constraints (*ConstraintType*) angehören. Wie in Kapitel 5.1 erläutert, ist eine Regel mindestens einem Ziel

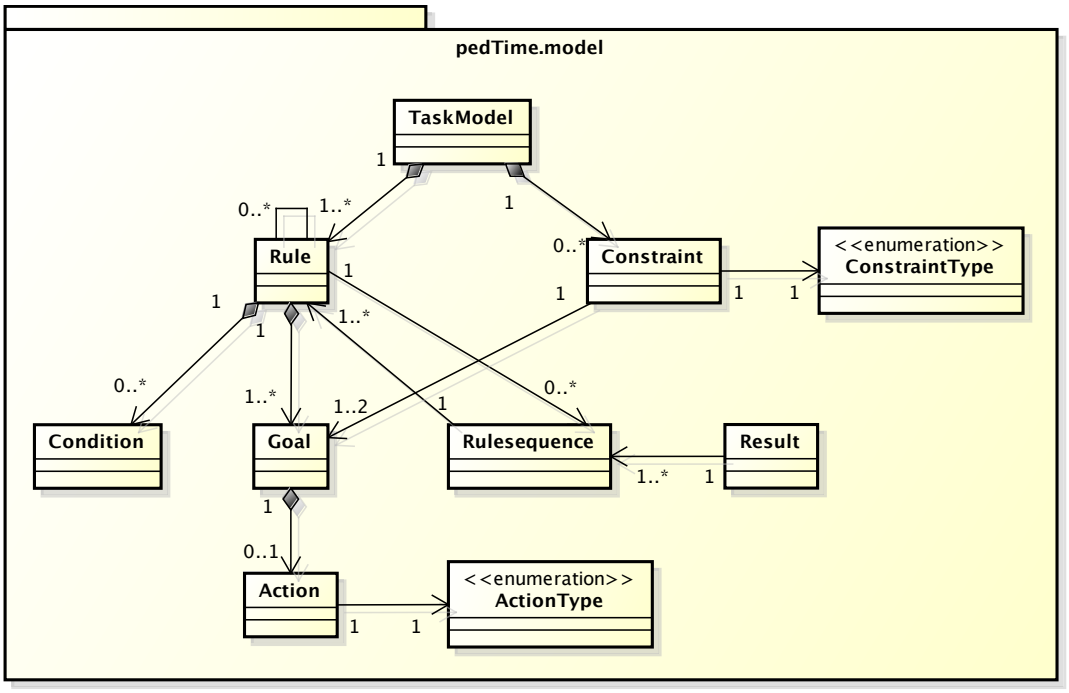


Abb. 7.4: UML-Klassendiagramm des vollständigen Unterpakets *model*

(*Goal*), ihrem sog. *lhsGoal* zugeordnet, kann aber auch Ziele auf ihrer RHS (sog. *rhsGoals*) beinhalten. Darüber hinaus kann eine Regel über beliebig viele Conditions verfügen. Durch den Aufbau der Regelhierarchie (s. Kapitel 6.2.2) kann eine Regel selbst beliebig viele Regeln (sog. *rhsRules*) besitzen. Nach Erstellung der Regelsequenzen (s. Kapitel 6.2.2) können der Regel beliebig viele Regelsequenzen (*Rulesequence*) zugeordnet sein, die z.B. Teil-Wege durch das Aufgabenmodell (sog. *subRulesequences*) ausgehend von der Regel repräsentieren. Eine Regelsequenz setzt sich wiederum aus mehreren Regeln zusammen. Einem Ziel kann kein oder genau ein Operator (*Action*) zugewiesen sein, der dieses Ziel zu einer Aktion der untersten Hierarchieebene des Aufgabenmodells macht. Jeder Operator hat genau einen bestimmten Operatortyp (*ActionType*). Das Ergebnis (*Result*) der Zeitanalyse besteht u.a. aus mehreren Regelsequenzen, die z.B. alle gültigen Wege durch das Aufgabenmodell mit minimaler Ausführungszeit (sog. *minimal-Rulesequences*) beschreiben.

7.3 Fundamentale Methoden von PEDTime

In diesem Abschnitt wird jede fundamentale Methode von PEDTime aufgeführt und ihre Funktion kurz erläutert. Dies schließt alle Konstruktoren und Hilfsmethoden aus, wozu auch gewöhnliche *get()*- und *set()*-Methoden gezählt werden.

7.3.1 Die Methode *createRuleHierarchy()* der Klasse *RuleHierarchyCreator*

Die Klasse *RuleHierarchyCreator* besitzt eine wichtige Methode, die für die Erstellung der Regelhierarchie zuständig ist. Die Methode *createRuleHierarchy(Rule rule, ArrayList<Rule> allRules)* erweitert das Aufgabenmodell, das in Form einer Zielhierarchie vorliegt um eine Regelhierarchie. Dazu weist sie ausgehend von der Wurzelregel rekursiv jeder Regel zu deren *rhsGoals* die entsprechenden *rhsRules* zu.

7.3.2 Methoden der Klasse *ConstraintChecker*

Die Klasse *ConstraintChecker* beinhaltet alle Methoden, die an der Überprüfung von Constraints beteiligt sind. In der aktuellen Implementierung von PEDTime sind vier Methoden zur Überprüfung von zwei unterschiedliche Arten von Constraints umgesetzt, die im Folgenden erläutert werden.

coordinateCheckConstraintsAfter(Rulesequence rs, ArrayList<Constraint> allConstraints)

Diese Methode sorgt dafür, dass eine Regelsequenz hinsichtlich aller definierten *nach*-Constraints, außer des *strikt nach*-Constraints überprüft wird. Darüber hinaus wertet sie die Ergebnisse der Einzelüberprüfungen aus, die durch die Methode *checkConstraintAfter()* durchgeführt werden und erstellt daraus eine Gesamtbewertung, die ausschließlich dann positiv ausfällt, wenn die Regelsequenz hinsichtlich aller *nach*-Constraints gültig ist.

coordinateCheckConstraintStrictlyAfter(Rulesequence rs, ArrayList<Constraint> allConstraints)

Diese Methode koordiniert die Überprüfung aller definierten *strikt nach*-Constraints und verhält sich dabei, unter Berücksichtigung des anderen Constraint-Typs äquivalent zu der vorherigen Methode *coordinateCheckConstraintsAfter()*.

checkConstraintAfter(Rulesequence rs, Constraint c)

In dieser Methode wird die tatsächliche Überprüfung eines *nach*-Constraints durchgeführt. Sie überprüft ob eine Regelsequenz gegen ein bestimmtes *nach*-Constraint verstößt und liefert die Rückmeldung dazu an die sie aufrufende Methode *coordinateCheckConstraintsAfter()* zurück, die das Ergebnis weiterverwertet.

checkConstraintStrictlyAfter(Rulesequence rs, Constraint c)

Entsprechend der Vorgehensweise der Methode *checkConstraintAfter()* verhält sich diese Methode für das *strikt nach*-Constraint.

Neben den vorgestellten Methoden sind zusätzliche Methoden für die Überprüfung der weiteren Constraints (s. Kapitel 5.2.2) vorgesehen, die aus Zeitgründen im Rahmen dieser Arbeit nicht implementiert wurden.

7.3.3 Methoden der Klasse *RulesequencesCreator*

Diese Klasse beinhaltet alle Methoden, die an der Erstellung der Regelsequenzen beteiligt sind. Bereits die Anzahl der nachfolgend aufgeführten Methoden, die essentiell beteiligt sind, lässt auf die Relevanz und Komplexität dieses Teils von PEDTime schließen.

createInitialRulesequences(ArrayList<Rule> allRules, ArrayList<Constraint> allConstraints)

Diese Methode erstellt die initialen Regelsequenzen, d.h. jene Regelsequenzen, die Regeln der ersten und zweiten Hierarchieebene des Aufgabenmodells enthalten.

Dies erfordert zunächst die Zusammenstellung von allen per UND verknüpften Regeln der *rhsRules* der Wurzelregel. Um diese zu erhalten, wird zunächst die Methode *prepareDisjunctionsAndConjunctions()* aufgerufen. Anschließend müssen alle möglichen Permutationen dieser jeweils verundeten Regeln vorgenommen werden, um tatsächlich alle möglichen Regelsequenzen zu erhalten. Dies erledigt an dieser Stelle die Methode *permute()*. Um ausschließlich gültige Regelsequenzen zu erhalten, werden innerhalb der Methode *permute()* Constraint-Überprüfungen veranlasst. Nachdem diese Vorbereitungen getroffen wurden, werden die initialen Regelsequenzen erzeugt. Schließlich gibt die Methode alle überprüften und gültigen Regelsequenzen zurück.

prepareDisjunctionsAndConjunctions(ArrayList<Rule> rhsRules)

Diese Methode bereitet die Zusammenstellung aller per UND verknüpften Regeln der *rhsRules* einer Regel vor. Dazu erstellt sie aus den *rhsRules* ggfs. mehrere Mengen von veroderten und eine Menge von verundeten Regeln, die sie zur Weiterverarbeitung an die Methode *createConjunctiveRhsRules()* weitergibt. Zurückgegeben wird letztlich eine Liste mit jeweils verundeten Regelmengen.

createConjunctiveRhsRules(ArrayList<ArrayList<Rule» allORLists, ArrayList<Rule> ANDList)

In dieser Methode werden die in der vorherigen Methode vorbereiteten Mengen von verundeten und veroderten Regelmengen zu ausschließlich verundeten Regelmengen weiterverarbeitet. Dazu wird aus jeder veroderten Regelmenge jeweils eine Regel mit allen Regeln der verundeten Regelmenge zu einer neuen Regelliste kombiniert. Dies wird für alle Regeln der veroderten Regelmengen wiederholt. Die Methode gibt eine Liste mit allen unterschiedlichen verundeten Regelmengen an die Methode *prepareDisjunctionsAndConjunctions()* zurück.

permute(ArrayList<Rule> listToPermute)

Innerhalb dieser Methode findet die Permutation aller Regeln der übergebenen Regelliste statt. Die Ausgabe ist eine Liste von Regelsequenzen, von denen jede Regelsequenz eine mögliche und gültige Permutation der Regeln der Regelliste

darstellt. Zur Überprüfung der Gültigkeit werden während der Permutation die Methoden zur Überprüfung von *nach-* und *strikt nach*-Constraints aufgerufen. Diese Methode ist maßgeblich daran beteiligt, sämtliche möglichen und gültigen Wege durch das Aufgabenmodell zu ermitteln.

createPermutedLists(Rule rule)

Diese Methode unterstützt den Aufbau der Regelsequenzen, indem sie rekursiv dafür sorgt, dass alle Permutationen der von der Methode *prepareDisjunctions-AndConjunctions()* erzeugten verundeten Regelmengen der übergebenen Regel erzeugt werden. Neben der zuletzt genannten Methode nutzt sie dafür außerdem die Methode *permute()*. Alle erzeugten Permutationen werden schließlich – in Form einer Liste von Regelsequenzen – als Attribut *permutedList* der Regel gespeichert.

coordinateCreateSubRuleSequences(Rule rule)

Diese Methode koordiniert die Erstellung der Teil-Regelsequenzen (s. Kapitel 6.2.2). Sie bekommt eine Regel übergeben, von der sämtliche Teil-Regelsequenzen erstellt werden. Diese fertig erstellten Teil-Regelsequenzen werden letztlich – in Form einer Liste von Regelsequenzen – der entsprechenden Regel über ihr Attribut *subRulesequences* zugewiesen. Innerhalb dieser Methode werden die beiden Methoden *createInitialSubRuleSequences()* und *createAllSubRuleSequences()* aufgerufen, die die eigentliche Erstellung der Teil-Regelsequenzen vornehmen.

createInitialSubRuleSequences(Rule rule)

Diese Methode sorgt für die Bereitstellung von initialen Teil-Regelsequenzen, die von der Methode *createAllSubRuleSequences()* zu sämtlichen kompletten Teil-Regelsequenzen weiterverarbeitet werden. Mit Hilfe der zuvor erstellten Liste (*permutedList*) aller Regeln werden durch geschicktes Kombinieren und Einfügen neue Regelsequenzen erzeugt, die die initialen Teil-Regelsequenzen der übergebenen Regel repräsentieren.

createAllSubRuleSequences(Rule rule)

In dieser Methode werden die initialen Teil-Regelsequenzen zu den kompletten Teil-Regelsequenzen der übergebenen Regel weiterverarbeitet. Dazu werden die initialen Teil-Regelsequenzen schrittweise mit Hilfe der *permutedList* jeder betroffenen Regel vervollständigt. Nachdem diese Methode ausgeführt wurde, liegen alle Teil-Regelsequenzen der übergebenen Regel vor, die Regeln bis einschließlich der 4. Hierarchietiefe des Aufgabenmodells beinhalten.

createRulesequences(ArrayList<Rulesequence> allValidRulesequences, Rule rootRule, ArrayList<Constraint> allConstraints)

Mit Hilfe der erstellten Teil-Regelsequenzen (*subRulesequences*) aller Regeln werden in dieser Methode alle vollständigen Regelsequenzen des gesamten Aufgabenmodells erstellt. Dies geschieht durch eine spezielle Vorgehensweise, bei der jeweils eine Teil-Regelsequenz der *subRulesequences* einer Regel an bestimmter Stelle in die initialen Regelsequenzen eingesetzt wird. Die Methode liefert schließlich alle kompletten und gültigen Regelsequenzen in Form einer Liste zurück.

Neben den soeben vorgestellten Methoden verfügt die Klasse *Rulesequences-Creator* über einige weitere Hilfsfunktionen, die die erläuterten Methoden unterstützen.

7.3.4 Methoden der Klasse *ExeTimesCalculatorOptimizer*

Die Klasse *ExeTimesCalculatorOptimizer* enthält alle Methoden, die an der Berechnung der Ausführungszeiten von Regelsequenzen mitwirken.

calculateExeTimesForRules()

Innerhalb dieser Methode werden die Ausführungszeiten aller Regeln des Aufgabenmodells berechnet. Dazu werden die einzelnen Ausführungszeiten von allen zeitbenötigenden Prozessen der jeweiligen Regel (s. Kapitel 6.2.2) aufsummiert und in das Attribut *ruleExeTime* der entsprechenden Regel gespeichert.

calculateExeTimesForRulesequences()

Diese Methode berechnet mit Hilfe der Ausführungszeiten der einzelnen Regeln die Ausführungszeiten für alle Regelsequenzen, indem sie die Zeiten aller Regeln, die in der jeweiligen Regelsequenz enthalten sind aufsummiert. Ggfs. sind auch spezielle Pufferzeiten zur Ausführungszeit der entsprechenden Regelsequenz hinzuzuaddieren (s. Kapitel 6.2.2). Die Ausführungszeit der jeweiligen Regelsequenz wird letztlich in ihr Attribut *totalExeTime* gespeichert.

calculateMaximalRulesequences()

Diese Methode ermittelt zunächst die maximale Ausführungszeit unter allen berechneten Ausführungszeiten der Regelsequenzen. Anschließend stellt sie alle Regelsequenzen fest, die diese maximale Ausführungszeit beanspruchen und legt diese in einer Liste ab, die letztlich von der Methode zurückgeliefert wird.

calculateMinimalRulesequences()

Entsprechend der Vorgehensweise der Methode *calculateMaximalRulesequences()* verfährt diese Methode für die minimale Ausführungszeit und die entsprechenden Regelsequenzen.

calculateMedianRulesequences(ArrayList<Integer> rsExeTimesList)

Damit diese Methode alle Regelsequenzen mit medianer Ausführungszeit ermitteln kann, muss sie zunächst den Median unter all diesen Ausführungszeiten berechnen. Dazu wird ihr eine Liste mit allen berechneten unterschiedlichen Ausführungszeiten übergeben, die zuvor durch eine Hilfsfunktion erstellt wurde. Nach der Sortierung dieser Liste wird der bzw. werden die Median(e) berechnet, zu dem bzw. zu denen anschließend alle entsprechenden Regelsequenzen ermittelt werden.

getResult()

Diese Methode erzeugt ein neues Objekt vom Typ *Result*, welches das Ergebnis der PEDTime-Zeitanalyse repräsentiert. Innerhalb dieser Methode werden die drei Methoden zur Berechnung der minimalen, maximalen und medianen Regelsequenzen aufgerufen und deren Ausgaben in entsprechende Attribute des Objekts gespeichert. Auch alle berechneten Ausführungszeiten, d.h. minimale, maximale und mediane, werden den dafür vorgesehenen Attributen als Werte zugewiesen.

Über die Attribute des erstellten *Result*-Objekts können alle in Kapitel 6.2.2 geforderten Bestandteile eines Ergebnisses abgefragt werden. Dies betrifft auch die durch die Regelsequenzen repräsentierten Zielsequenzen. Diese Zielsequenzen können über das jeweilige *lhsGoal* der einzelnen Regeln einer Regelsequenz ermittelt werden. Ebenfalls eventuell gesetzte Variablenwerte und definierte Pufferzeiten für einzelne Regelsequenzen sind über entsprechende Attribute des *Result*-Objekts abrufbar.

Auch die aufgeführten Methoden dieser Klasse werden durch einige zusätzliche Hilfsfunktionen unterstützt.

7.3.5 Die Methode *calculateResult()* der Klasse *PedTimeCalculator*

Die Klasse *PedTimeCalculator* besitzt eine bedeutende Methode, die die komplette Zeitanalyse in PEDTime steuert. Die Methode *calculateResult()* stellt sicher, dass alle Methoden ausgeführt werden, die für die Zeitanalyse benötigt werden. Sie gibt das Ergebnis der Zeitanalyse als Objekt vom Typ *Result* zurück, welches alle wichtigen Teilergebnisse enthält.

7.4 Implementierung ausgewählter Methoden von PEDTime

Nachdem die Struktur sowie alle bedeutenden Methoden von PEDTime vorgestellt wurden, folgt nun die detaillierte Erläuterung einiger interessanter Methoden, die

zur Umsetzung des Konzepts entwickelt wurden. Zur Darstellung der Implementierung wird Pseudocode verwendet, da dieser verständlicher formuliert ist, als der entsprechende Java-Code, aber dennoch alle essentiellen Lösungsansätze aufzeigt.

7.4.1 Differenzierung zwischen disjunktiv und konjunktiv verknüpften Regeln

In Kapitel 6.2.2 des Konzepts wurde auf die Notwendigkeit der Differenzierung von UND- und ODER-Verknüpfungen zwischen den Regeln der RHS einer Regel, d.h. *rhsRules* eingegangen. Die beiden folgenden Methoden sind dafür zuständig, alle per UND (konjunktiv) verknüpften Regelmengen der *rhsRules* einer Regel zu identifizieren. Diese Regelmengen werden später zur Erstellung der Regelsequenzen verwendet. Pro Regelsequenz dürfen dabei nur die Regeln aus einer Regelmenge verwendet werden, die ggfs. in Form unterschiedlicher Permutationen einfließen.

Der Algorithmus 1 bekommt die *rhsRules* einer Regel übergeben und gibt eine Liste mit allen in sich per UND verknüpften Regellisten der *rhsRules* zurück. Für den Fall, dass alle *rhsRules* verundet sind, d.h. sie alle unterschiedliche *lhsGoals* haben, entspricht dieses Ergebnis genau einer Regelliste: den *rhsRules* selbst. Für den Fall, dass mehrere Regeln der *rhsRules* dasselbe *lhsGoal* haben, liegt eine ODER-Verknüpfung zwischen diesen Regeln vor und sie werden, sofern noch nicht enthalten, als neue veroderte (disjunktive) Regelmenge der Liste *allORLists* hinzugefügt. Der zuletzt genannte Fall wird für alle verschiedenen *lhsGoals*, die durch die *rhsRules* definiert werden, geprüft, sodass schließlich sämtliche veroderten Regelmengen in der Liste *allORLists* enthalten sind. Alle verundeten Regeln, d.h. alle Regeln, die nicht in der Liste *allORLists* enthalten sind, werden anschließend der Liste *ANDList* hinzugefügt. Mit beiden Listen wird dann die Methode *createConjunctiveRhsRules()* aufgerufen, deren Ergebnis auch das Ergebnis der Methode *prepareDisjunctionsAndConjunctions()* darstellt. Für einige Fall-Abfragen sind Hilfzähler nötig, auf deren Darstellung hier jedoch zu Gunsten der Übersichtlichkeit verzichtet wird.

Die Methode *createConjunctiveRhsRules()* wird anhand von Algorithmus 2 veranschaulicht. Sie verarbeitet die übergebenen Listen ihrer aufrufenden Methode

Algorithmus 1 prepareDisjunctionsAndConjunctions(Regelliste rhsRules)

```
Erzeuge neue Liste allORLists von Regelmengen
Erzeuge neue Liste ANDList von Regeln
for all Regeln r1 in rhsRules do
  Erzeuge neue Liste sameLhsGoalList von Regeln
  for all Regeln r2 in rhsRules do
    if r1 != r2 then
      if lhsGoal von r1 == lhsGoal von rs2 then
        Füge r2 zur Liste sameLhsGoalList hinzu
      end if
    end if
  end for
  if sameLhsGoalList enthält mindestens 1 Element then
    Füge r1 zur Liste sameLhsGoalList hinzu
    if not allORLists enthält 1 Liste mit gleichen Regeln der Liste sameLhs-
    GoalList then
      Füge sameLhsGoalList zur Liste allORLists hinzu
    end if
  end if
end for
if alle Regeln in rhsRules haben unterschiedliche lhsGoals then
  Füge rhsRules zur Liste allORLists hinzu
  return allORLists
end if
for all Regeln in rhsRules do
  if not Regel ist in einer Liste von allORLists enthalten then
    Füge Regel zur Liste ANDList hinzu
  end if
end for
return createConjunctiveRhsRules(allORLists, ANDList)
```

folgendermaßen: Durch Kombinieren jeweils einer Regel der ersten Regelmenge der Liste *allORLists* mit allen verundeten Regeln in *ANDList* wird eine neue Regelliste erzeugt. Diese neue Regelliste wird der finalen Liste *allAndLinkedRules* hinzugefügt. Anschließend wird für alle weiteren Regelmengen der Liste *allORLists* fortgefahren, indem nun die gerade gefüllte Liste *allAndLinkedRules* als Ausgangsliste verwendet wird und alle ihrer Regellisten separat mit den einzelnen Regeln (ebenfalls separat) einer Regelmenge aus *allORLists* zu einer neuen Liste verknüpft werden. Jede dieser neuen Listen wird eine temporären Liste (*tempList*) hinzugefügt, welche vor dem Wechsel zur nächsten Regelmenge in *allORLists* als

neuer Inhalt der Liste *allAndLinkedRules* gesetzt und anschließend geleert wird. Dieser Listenwechsel ist nötig, damit für jede neue Regelmenge in *allORLists* die vorher gefüllte *tempList* die neue Ausgangsliste darstellt, anhand der wiederum neue Listen erstellt werden.

Schließlich enthält *allAndLinkedRules* alle möglichen Regellisten, deren Regeln untereinander per UND verknüpft sind und damit für die Erstellung von Regelsequenzen verwendet werden können.

Algorithmus 2 createConjunctiveRhsRules(Regelmengen-Liste allORLists, Regelliste ANDList)

```
Erzeuge neue finale Liste allAndLinkedRules von Regellisten
Erzeuge neue Liste tempList von Regellisten
for all Regeln in erster Regelmenge von allORLists do
  Erzeuge neue Liste von Regeln
  Füge Regel zur neuen Liste hinzu
  Füge alle Regeln der Liste ANDList zur neuen Liste hinzu
  Füge neue Liste zur Liste allAndLinkedRules hinzu
end for
if Anzahl Listen in allORLists == 1 then
  return allAndLinkedRules
else
  for all Regelmengen rm außer der Ersten in allORLists do
    for all Regellisten rl in allAndLinkedRules do
      for all Regeln r in rm do
        Erzeuge neue Liste von Regeln
        Füge alle Regeln der Regelliste rl zur neuen Liste hinzu
        Füge Regel r zur neuen Liste hinzu
        Füge neue Liste zur Liste tempList hinzu
      end for
    end for
    Setze tempList als neuen Inhalt der Liste allAndLinkedRules
    Leere tempList
  end for
  return allAndLinkedRules
end if
```

7.4.2 Permutation von Regellisten

Um alle möglichen Regelsequenzen erstellen zu können, müssen die Regellisten, die als Ergebnis der Algorithmen des vorherigen Abschnitts 7.4.1 erzeugt wurden, permutiert werden. Das bedeutet, dass für jede einzelne Liste von Regeln alle möglichen Permutationen, d.h. Reihenfolgen von Regeln erzeugt werden.

Der Algorithmus 3 erzeugt aus einer Liste von Regeln alle möglichen Permutationen in Form von Regelsequenzen. Anschließend werden die Permutationen auf reihenfolgebezogene Constraints geprüft. Alle als gültig bewerteten Regelsequenzen werden schließlich zurückgeliefert. Diese Regelsequenzen werden später zur Erstellung der kompletten Wege durch das Aufgabenmodell, repräsentiert durch Regelsequenzen, benötigt.

Enthält die zu permutierende Liste (*listToPermute*) nur eine Regel, so wird diese als neue Regelsequenz zurückgeliefert, da bereits alle möglichen Permutationen vorliegen. Für alle anderen Fälle wird wie folgt vorgegangen: Die Permutation wird mit Hilfe von zwei ineinander verschachtelten For-Schleifen und einer Rekursion durchgeführt. Innerhalb der äußeren For-Schleife, die dafür sorgt, dass alle Regeln der zu permutierenden Liste durchlaufen werden, finden die rekursiven Aufrufe statt. Für jede Regel der zu permutierenden Liste wird eine neue Liste von Regelsequenzen erzeugt. Diese Liste wird mit dem Ergebnis des rekursiven Aufrufs der Methode gefüllt. Die Rekursion wird dabei für die permutierende Liste ohne die aktuelle Regel ausgeführt. Dadurch wird gewährleistet, dass eine fertige Permutation letztlich keine doppelten Regeln enthält.

Die rekursiven Aufrufe gewährleisten, dass tatsächlich alle möglichen Permutationen berücksichtigt werden. Sie bewirken den Aufbau der permutierten Regelsequenzen „von hinten nach vorne“, d.h. es wird immer eine weitere Regel der zu permutierenden Regelliste hinzugenommen, die bei der Permutation berücksichtigt wird. Erst wenn alle möglichen Permutationen dieser hintersten Regel bzw. Regeln erstellt wurden, werden sie durch die innere For-Schleife an die aktuelle Regel der äußeren For-Schleife angefügt, sodass neue Regelsequenzen entstehen. Diese werden durch die Rekursion immer weiter nach „vorne“ gereicht, d.h. wieder von der nächsten Regel der äußeren For-Schleife verwendet, um neue Regelsequenzen zu erzeugen und so fort. Die Rekursion sorgt quasi für die Weitergabe der immer größer werdende Menge von Teil-Permutationen zur nächsten Regel

Algorithmus 3 permute(Regelliste listToPermute)

```
Erzeuge neue Liste permutedRs von Regelsequenzen
Erzeuge neue Liste checkedRs von Regelsequenzen
Erzeuge neue finale Liste validRs von Regelsequenzen

if Anzahl Elemente in listToPermute == 1 then
  Erzeuge neue Regelsequenz
  Fülle neue Regelsequenz mit Regeln aus listToPermute
  return neue Regelsequenz
else
  for all Regeln r in listToPermute do
    Erzeuge neue Liste tempList von Regelsequenzen
    Fülle neue Liste mit permute(Entferne Regel r aus listToPermute)
    for all Regelsequenzen rs in tempList do
      Erzeuge neue Regelsequenz
      Füge Regel r zur neuen Regelsequenz hinzu
      Füge alle Regeln von Regelsequenz rs zur neuen Regelsequenz hinzu
      Füge neue Regelsequenz zur Liste permutedRs hinzu
    end for
  end for
  if not permutedRS ist leer then
    for all Regelsequenzen in permutedRS do
      if coordinateCheckConstraintStrictlyAfter(Regelsequenz, alle Cons-
      traints) ist gültig then
        Füge Regelsequenz zur Liste checkedRs hinzu
      end if
    end for
    for all Regelsequenzen in checkedRs do
      if coordinateCheckConstraintsAfter(Regelsequenz, alle Constraints) ist
      gültig then
        Füge Regelsequenz zur Liste validRs hinzu
      end if
    end for
  end if
  return validRs
end if
```

der äußeren For-Schleife. Das Anwachsen dieser Menge ist auf die Kombination aus der inneren For-Schleife und der Rekursion, sowie schließlich auch der äußeren For-Schleife zurückzuführen.

Bereits während alle möglichen Permutationen der übergebenen Regelliste erzeugt wurden, werden diese einzeln auf bestimmte Constraints geprüft. Dies betrifft

auch schon Teil-Permutationen, d.h., bevor diese durch die Rekursion weitergegeben werden, werden sie überprüft, sodass durch jeden neuen Rekursionsaufruf ausschließlich als gültig bewertete Regelsequenzen zur weiteren Permutation berücksichtigt werden. Diese Überprüfung findet innerhalb der im folgenden Abschnitt vorgestellten Methoden statt. Nur die gültigen Permutationen werden als Ergebnis der Methode *permute()* in Form einer Liste von Regelsequenzen zurückgeliefert.

7.4.3 Überprüfung einer Regelsequenz auf Einhaltung von *nach*-Constraints

In diesem Abschnitt wird das konkrete Vorgehen zur Überprüfung von reihenfolgebezogenen Constraints erläutert. Als Beispiel wird die Überprüfung aller *nach*-Constraints vorgestellt. Damit sind an dieser Stelle alle Constraints bis auf *strikt nach* gemeint, deren Typ das Schlüsselwort *nach* enthält. Allerdings wird an dieser Stelle ausschließlich die Reihenfolge von Regeln überprüft, d.h., dass evtl. Mengenangaben in den Constraints noch ignoriert werden. Das *strikt nach*-Constraint wird in einer separaten Methode überprüft, da es einen Spezialfall der *nach*-Constraints darstellt.

Der Algorithmus 4 zeigt die Methode *coordinateCheckConstraintsAfter()*, die für die Koordination der Constraint-Überprüfung sorgt. Sie stellt sicher, dass die übergebene Regelsequenz hinsichtlich aller *nach*-Constraints überprüft wird. Sämtliche definierten Constraints sind in der Liste *allConstraints* enthalten. Darüber hinaus fängt die Methode einige Spezialfälle ab, die im Folgenden erläutert werden.

Ist die Liste *allConstraints* leer, dann sind gar keine Constraints definiert und es wird *true* zurückgeliefert, d.h. die Constraint-Überprüfung der Regelsequenz war erfolgreich. Falls die Regelsequenz alle definierten *nach*-Constraints erfüllt, so wird ebenfalls *true* zurückgegeben. Für den Fall, dass kein einziges *nach*-Constraint in *allConstraints* enthalten ist, kann auch kein *nach*-Constraint verletzt werden, sodass die Regelsequenz gültig ist (true). Für alle anderen Fälle verletzt die Regelsequenz mindestens ein *nach*-Constraint und es wird *false* zurückgeliefert.

Die tatsächliche Überprüfung der *nach*-Constraints findet in der Methode *checkConstraintAfter()* statt, die durch Algorithmus 5 beschrieben wird. Sie überprüft

Algorithmus 4 coordinateCheckConstraintsAfter(Regelsequenz *rs*, Constraint-Liste *allConstraints*)

```
if Anzahl Elemente in allConstraints == 0 then
  return true
else
  for all Constraints c in allConstraints do
    checkConstraintAfter(rs, c)
  end for
  if Regelsequenz rs ist für alle nach-Constraints gültig then
    return true
  end if
  if not Es gibt mindestens 1 nach-Constraint in allConstraints then
    return true
  end if
  return false
end if
```

die ihr übergebene Regelsequenz auf Einhaltung des ihr übergebenen Constraints und liefert nur dann *true* zurück, wenn das übergebene Constraint tatsächlich ein *nach*-Constraint ist und von der Regelsequenz nicht verletzt wird. Das Abfangen von anderen Constraints und bestimmten Sonderfällen erledigt die soeben vorgestellte Methode in Algorithmus 4 teilweise mit Hilfe bestimmter Zähler, die hier zu Gunsten der Übersichtlichkeit nicht dargestellt sind.

Ein *nach*-Constraint ist immer für ein linkes (*leftGoal*) und ein rechtes Ziel (*rightGoal*) definiert, wobei das linke Ziel nach dem rechten Ziel erfüllt werden muss (s. Kapitel 5.2.3). Zunächst wird überprüft, ob das übergebene Constraint ein *nach*-Constraint ist. Trifft dies zu, so wird für alle Regeln *r1* der Regelsequenz überprüft, ob *r1* für das *leftGoal* des Constraints definiert ist. Falls dem so ist, wird in allen Regeln *r2* der Regelsequenz nach einer Regel gesucht, die für das *rightGoal* des Constraints definiert ist. Wurde auch solch eine Regel *r2* gefunden, wird überprüft, ob *r1* innerhalb der Regelsequenz hinter *r2* platziert ist. Wenn dies zutrifft, ist das Constraint erfüllt und es wird *true* zurückgeliefert. Andernfalls ist das Constraint verletzt, d.h., *false* wird zurückgegeben. Enthält die Regelsequenz zwar eine Regel, die für das *leftGoal* des Constraints definiert ist, aber keine weitere, die zusätzlich für dessen *rightGoal* definiert ist, so ist die Regelsequenz nicht vom Constraint betroffen und kann dies folglich nie verletzen. Deshalb wird für diesen Fall *true* zurückgeliefert. Entsprechendes gilt für den Fall, dass bereits

keine Regel zu dem *leftGoal* des Constraints in der Regelsequenz gefunden wurde. Falls das übergebene Constraint kein *nach*-Constraint ist, wird *false* zurückgegeben. Dies ist u.a. nötig, damit die weiterverarbeitende Methode in Algorithmus 4 eindeutig feststellen kann, ob die Regelsequenz tatsächlich alle *nach*-Constraints erfüllt.

Algorithmus 5 checkConstraintAfter(Regelsequenz *rs*, Constraint *c*)

```
if Constraint-Typ von c ist nach oder genau nach oder max nach oder min
nach then
  for all Regeln r1 der Regelsequenz rs do
    if lhsGoal von r1 == leftGoal von c then
      for all Regeln r2 der Regelsequenz rs do
        if lhsGoal von r2 == rightGoal von c then
          if Index von r1 in rs > Index von r2 in rs then
            return true
          else
            return false
          end if
        end if
      end for
      if Keine Regel r2 in rs == rightGoal von c then
        return true
      end if
    end if
  end for
  if Keine Regel r1 in rs == leftGoal von c then
    return true
  end if
end if
return false
```

7.4.4 Erstellung der Teil-Regelsequenzen

Weitere fundamentale Methoden von PEDTime sind jene, die für die Erstellung der Regelsequenzen sorgen. Wie bereits in Kapitel 6.2.2 erwähnt, stellt die Erstellung von Teil-Regelsequenzen einen sehr wichtigen und komplexen Schritt dar, der von den nachfolgend beschriebenen Methoden durchgeführt wird.

Die Methode *coordinateCreateSubRuleSequences()* koordiniert die Erstellung der Teil-Regelsequenzen (s. Kapitel 7.3.3). Die tatsächliche Erstellung der Teil-Regel-

sequenzen wird in den Methoden *createInitialSubRuleSequences()* und *createAllSubRuleSequences()* vorgenommen. Diese beiden Methoden werden im Folgenden in Form von Pseudocode dargestellt und erläutert. Die Methode *coordinateCreateSubRuleSequences()* und damit auch die beiden o.g. enthaltenen Methoden werden für alle *rhsRules* der ersten Regel der Hierarchie, d.h. der Wurzel ausgeführt.

Der Algorithmus 6 zeigt die initiale Erstellung der Teil-Regelsequenzen. Er verwendet u.a. die zuvor erstellte Liste *permutedList* einer Regel. In dieser Liste *permutedList* einer Regel sind jene Regelsequenzen enthalten, die die verundeten *rhsRules* der Regel in sämtlichen Permutationen darstellen. Somit repräsentieren die Regelsequenzen einer *permutedList* alle möglichen Wege ab der entsprechenden Regel in die nächsttiefere Ebene des Aufgabenmodells. *tempList* bezeichnet eine global definierte Liste von Regelsequenzen, auf die von allen an der Aufstellung der Teil-Regelsequenzen beteiligten Methoden aus zugegriffen wird. Diese Liste enthält Teillösungen, die während der Erstellung der Teil-Regelsequenzen ständig erweitert werden.

Algorithmus 6 createInitialSubRuleSequences(Regel regel)

```
if regel hat rhsRules then
  for all Regelsequenzen rs1 der permutedList von regel do
    for all Regelsequenzen rs2 der permutedList der 1. rhsRule von regel do
      Erzeuge neue Regelsequenz
      Füge alle Regeln von rs1 zur neuen Regelsequenz hinzu
      if Neue Regelsequenz enthält 1. rhsRule von regel then
        Ermittle Stelle, an der 1. rhsRule von regel in neuer Regelsequenz
        steht
        Füge alle Regeln von rs2 direkt hinter dieser Stelle in neue Regelse-
        quenz ein
      end if
      Füge neue Regelsequenz zur tempList hinzu
    end for
  end for
  if regel hat genau 1 rhsRule, die selbst keine rhsRules mehr hat then
    Füge alle Regelsequenzen der permutedList von regel zur tempList hinzu
  end if
end if
```

Die Methode beginnt die Teil-Regelsequenzen der übergebenen Regel zu erstellen. Dazu geht sie zunächst alle Regelsequenzen der *permutedList* der Regel durch. Für jede dieser Regelsequenzen werden wiederum alle Regelsequenzen der *permu-*

tedList der ersten *rhsRule* von der Regel durchlaufen. Die beiden For-Schleifen repräsentieren quasi den Durchlauf von zwei unterschiedlichen Hierarchieebenen des Aufgabenmodells ausgehend von der entsprechenden Regel. Für jede Regelsequenz der *permutedList* der ersten *rhsRule* wird eine neue Regelsequenz erzeugt. Diese wird dann mit allen Regeln der Regelsequenz *rs1* gefüllt und stellt die Basis für den Aufbau der Teil-Regelsequenzen dar. Nun wird innerhalb der neuen Regelsequenz die Stelle ermittelt, an der die erste *rhsRule* der Regel platziert ist, sofern diese überhaupt in der Regelsequenz enthalten ist. Anschließend wird genau hinter dieser Stelle die Regelsequenz *r2* eingefügt und die dadurch ergänzte neue Regelsequenz in einer Liste abgespeichert. Durch diesen Einfügungvorgang wird jede der Ausgangsregelsequenzen bzgl. einer weiteren Hierarchietiefe konkretisiert, wie Abbildung 6.3 in Kapitel 6.2.2 veranschaulicht. Diese Methode nimmt diese Konkretisierung jedoch nur für die erste *rhsRule* der Regel vor. Alle weiteren Regeln werden in der Methode *createAllSubRuleSequences()* betrachtet, die im Anschluss an die soeben vorgestellte Methode ausgeführt wird.

Für den Fall, dass die Regel lediglich eine *rhsRule* hat, die selbst über keine *rhsRules* verfügt, existiert keine weitere Hierarchietiefe, d.h. die zweite For-Schleife wird nie durchlaufen. Deshalb werden dann die Regelsequenzen der *permutedList* der Regel, die in diesem Fall nur aus einer Regelsequenz mit der einzigen *rhsRule* der Regel bestehen, zu den „neuen“ Teil-Regelsequenzen der Regel.

Nachdem in der zuletzt vorgestellten Methode nur die erste *rhsRule* der Regel betrachtet wurde, wird die Erstellung der Teil-Regelsequenzen nun mit Hilfe aller weiteren Regeln in der Methode *createAllSubRuleSequences()* (s. Algorithmus 7) vorgenommen. Die durch die letzte Methode gefüllte *tempList* dient dabei als Ausgangsliste für den weiteren Aufbau der Teil-Regelsequenzen. Die Liste *tempList2* bezeichnet wie *tempList* eine global definierte Liste von Regelsequenzen.

Die Methode *createAllSubRuleSequences()* erhält als Eingabe eine Regel, deren Teil-Regelsequenzen sie weiter aufbaut. Da die erste Regel der *rhsRules* von der übergebenen Regel bereits in der vorherigen Methode *createInitialSubRuleSequences()* berücksichtigt wurde, wird diese hier ausgelassen. Alle anderen Regeln *r1* der *rhsRules* von *regel* werden durchlaufen. Nachdem sichergestellt wurde, dass *r1 rhsRules* hat, werden zwei unterschiedliche Zweige offeriert.

Der erste Zweig wird gewählt, wenn die Liste *tempList* gefüllt ist, der andere Zweig hingegen wenn dieselbe Liste leer ist. Ist diese Liste leer, bedeutet dies gleichzeitig,

Algorithmus 7 createAllSubRuleSequences(Regel regel)

```
for all Regeln r1 der rhsRules von regel außer der 1. do
  if r1 hat rhsRules then
    if not tempList ist leer then
      for all Regelsequenzen rs1 der permutedList von r1 do
        for all Regelsequenzen rs2 in tempList do
          Erzeuge neue Regelsequenz
          Füge alle Regeln von rs2 zur neuen Regelsequenz hinzu
          if Neue Regelsequenz enthält r1 then
            Ermittle Stelle, an der r1 in neuer Regelsequenz steht
            Füge alle Regeln von rs1 direkt hinter dieser Stelle in neue Regel-
            sequenz ein
          end if
          Füge neue Regelsequenz zur tempList2 hinzu
        end for
      end for
      Leere tempList
    else
      for all Regelsequenzen rs1 der permutedList von r1 do
        for all Regelsequenzen rs2 in tempList2 do
          Erzeuge neue Regelsequenz
          Füge alle Regeln von rs2 zur neuen Regelsequenz hinzu
          if Neue Regelsequenz enthält r1 then
            Ermittle Stelle, an der r1 in neuer Regelsequenz steht
            Füge alle Regeln von rs1 direkt hinter dieser Stelle in neue Regel-
            sequenz ein
          end if
          Füge neue Regelsequenz zur tempList hinzu
        end for
      end for
      Leere tempList2
    end if
    Rekursion in die Tiefe ausgehend von r1, d.h. createAllSubRuleSe-
    quences(r1)
  end if
end for
if 1. Regel der rhsRules von regel hat rhsRules then
  Rekursion in die Tiefe nur für 1. Regel der rhsRules von regel
end if
if keine rhsRule von regel hat rhsRules then
  Füge alle Regelsequenzen der permutedList von regel zur tempList hinzu
end if
```

dass die Liste *tempList2* gefüllt ist. Der Grund dafür wird durch die nachfolgende Erläuterung deutlich. Für den Fall, dass *tempList* gefüllt ist, werden alle Regelsequenzen *rs1* der *permutedList* von *r1* durchlaufen. Für alle diese Regelsequenzen werden wiederum sämtliche Regelsequenzen *rs2* der Liste *tempList* durchlaufen. Für jede Regelsequenz *rs2* wird eine neue Regelsequenz erzeugt. Diese neue Regelsequenz wird mit der aktuellen *Regelsequenz* aus der *tempList* gefüllt, die hier die Ausgangsliste zum weiteren Aufbau der Teil-Regelsequenzen darstellt. Anschließend wird die Stelle ermittelt, an der *r1* in der neuen Regelsequenz platziert ist. Direkt hinter dieser Stelle werden dann alle Regeln von *rs1* eingefügt. Schließlich wird die neue Regelsequenz zur Liste *tempList2* hinzugefügt. Dieses Vorgehen entspricht der Vorgehensweise an der entsprechenden Stelle im Algorithmus 6. Am Ende des ersten Zweiges, d.h., wenn alle Regelsequenzen der *permutedList* von *r1* durchlaufen sind und damit die nächste Regel *r1* betrachtet werden kann, wird *tempList* geleert. Innerhalb des zweiten Zweiges wird wie im ersten Zweig vorgegangen, jedoch mit der Abweichung, dass anstelle von *tempList* die Liste *tempList2* durchlaufen wird und jede neue Regelsequenz zur Liste *tempList* hinzugefügt wird, sowie dass am Ende die Liste *tempList2* geleert wird.

Durch die erläuterte Verwendung der beiden Listen *tempList* und *tempList2* wird gewährleistet, dass für jeden Durchlauf der äußeren For-Schleife, d.h. für jede neue Regel *r1* die zuletzt gefüllte Liste die neue Ausgangsliste darstellt. Nur so werden die Teil-Regelsequenzen schrittweise korrekt ergänzt.

Für den Fall, dass die Regel nur *rhsRules* hat, von denen wiederum keine selbst über *rhsRules* verfügt, ist die tiefste Hierarchieebene erreicht und es werden keine neuen Regelsequenzen erstellt, weil die inneren For-Schleifen nie durchlaufen werden. Die Regelsequenzen der *permutedList* der Regel sind in diesem Fall äquivalent zu den Teil-Regelsequenzen der Regel und werden deshalb zur Liste *tempList* hinzugefügt, wodurch sie später als Teil-Regelsequenzen der Regel gespeichert werden.

Nachdem die Methoden *createInitialSubRuleSequences()* und *createAllSubRuleSequences()* aufgerufen wurden, übernimmt die Methode *coordinateCreateSubRuleSequences()* die Zuweisung des Attributs *subRulesequences* der entsprechenden Regel. Eine der Listen *tempList* und *tempList2* ist zu diesem Zeitpunkt leer. In der anderen Liste befinden sich alle Teil-Regelsequenzen der Regel, sodass der Inhalt der vollen Liste nun in die Liste *subRulesequences* der Regel gespeichert

wird.

Die Methoden *createInitialSubRuleSequences()* und *createAllSubRuleSequences()* erstellen genau alle Teil-Regelsequenzen für alle *rhsRules* der Wurzelregel (z.T. durch wiederholte externe Aufrufe) bis in die einschließlich 4. Hierarchietiefe des Aufgabenmodells. Um auch tiefere Hierarchieebenen bei der Erstellung der Teil-Regelsequenzen berücksichtigen zu können, muss an entsprechender Stelle in der Methode *createAllSubRuleSequences()* eine Rekursion eingefügt werden. Im Pseudocode ist diese Rekursion innerhalb der äußersten For-Schleife angedeutet. Ausgehend von der aktuellen Regel *r1* muss ein rekursiver Aufruf der Methode vorgenommen werden. Darüber hinaus muss auch für die 1. Regel der *rhsRules* der Regel eine entsprechende Rekursion durchgeführt werden, da diese in der Methode *createInitialSubRuleSequences()* ebenfalls nur bis einschließlich der 4. Hierarchietiefe behandelt wurde. Um diese Rekursionen so durchführen zu können, muss allerdings u.a. das Verfahren des Listenwechsels zwischen *tempList* und *tempList2* überdacht und ggfs. verändert werden. Die Einbindung dieser Rekursion ist zu diesem Zeitpunkt noch nicht genauer konzipiert, sodass in der aktuellen Implementierung keine tieferen Hierarchieebenen als einschließlich der 4. berücksichtigt werden.

7.5 Speicherverbrauch zur Laufzeit als Problem des implementierten Programms

Die Ausführung des implementierten PEDTime-Prototyps kann zu einem markanten Speicherverbrauch führen, wenn im Rahmen der Zeitanalyse zu viele gültige Regelsequenzen erstellt werden. Der erhöhte Speicherverbrauch kann bereits bei dem in dieser Arbeit betrachteten Szenario problematisch werden, sofern keine Constraints berücksichtigt werden. Dies wird im Folgenden anhand des Szenarios verdeutlicht.

Die Entwicklung der Anzahl an Regelsequenzen während der Konzeptschritte, die in Kapitel 6.2.2 erläutert wurden, verhält sich für das Szenario folgendermaßen:

Zunächst wird die Anzahl der Regelsequenzen bei der Erstellung der Teil-Regelsequenzen von allen *rhsRules* der Wurzelregel berechnet.

Für jede Regel r der *rhsRules* der jeweiligen *rhsRule* der Wurzelregel gilt:

Anzahl neuer Regelsequenzen = Anzahl Ausgangs-Regelsequenzen ∗ Anzahl permutierter Regelsequenzen der Regel r

Die Tabelle 7.1 zeigt diese Berechnung für alle *rhsRules* der Wurzelregel.

Ausgangsregel *rule2*:			
Anzahl permutierter Regelsequenzen der Ausgangsregel:			120
Anzahl Regelsequenzen inkl. aufgelöster Regel (*rule3*):	120*4	=	480
Anzahl Regelsequenzen inkl. aufgelöster Regel (*rule4*):	480*4	=	1'920
Anzahl Regelsequenzen inkl. aufgelöster Regel (*rule5*):	1920*4	=	7'680
Anzahl Regelsequenzen inkl. aufgelöster Regel (*rule6*):	7680*4	=	30'720
Anzahl Regelsequenzen inkl. aufgelöster Regel (*rule7*):	30720*4	=	122'880
Ausgangsregel *rule18*:			
Anzahl permutierter Regelsequenzen der Ausgangsregel:			2
Anzahl Regelsequenzen inkl. aufgelöster Regel (*rule19*):	2*4	=	8
Anzahl Regelsequenzen inkl. aufgelöster Regel (*rule33*):	8*4	=	32

Tab. 7.1: Entwicklung der Anzahl an Regelsequenzen im Szenario

Für die erste Regel *rule3* wird von 120 Regelsequenzen ausgegangen, die mit den 4 unterschiedlichen permutierten Regelsequenzen dieser Regel multipliziert werden. Das Ergebnis (480) stellt nun die Anzahl der Ausgangs-Regelsequenzen für die nächste Regel *rule4* dar. So wird für alle Regeln r fortgefahren. Diese komplette Berechnung wird für alle Ausgangsregeln, d.h. *rhsRules* der Wurzelregel separat wiederholt. Im Szenario existieren die beiden Ausgangsregeln *rule2* und *rule18*.

Anschließend werden die kompletten Regelsequenzen erstellt, indem die Teil-Regelsequenzen gemäß dem Vorgehen in Kapitel 6.2.2 in die initialen Regelsequenzen eingesetzt werden. Dadurch vervielfacht sich die Anzahl der Regelsequenzen erneut pro aufgelöster Ausgangsregel, d.h. *rhsRule* der Wurzelregel entsprechend der Formel:

Anzahl neuer Regelsequenzen = Anzahl Ausgangs-Regelsequenzen ∗ Anzahl Teil-Regelsequenzen der Regel

Die Tabelle 7.2 zeigt, wie sich die Anzahl an Regelsequenzen für das Szenario entwickelt.

Anzahl initialer Regelsequenzen: 2			
Anzahl Regelsequenzen inkl. aufgelöster Regel *rule2*:	2*122880	=	245'760
Anzahl Regelsequenzen inkl. aufgelöster Regel *rule18*:	245760*32	=	7'864'320

Tab. 7.2: Entwicklung der endgültigen Anzahl an Regelsequenzen im Szenario

Das Endergebnis 7'864'320 drückt die Anzahl aller möglichen Regelsequenzen des Aufgabenmodells aus, sofern keine Constraints berücksichtigt werden. Zu beachten ist bei diesem Wert, dass ausschließlich Regeln bis einschließlich der 4. Hierarchieebene des Aufgabenmodells bei der Erstellung der Regelsequenzen betrachtet wurden. Für weitere, tiefere Ebenen vervielfacht sich die Anzahl der Regelsequenzen entsprechend. Das Aufgabenmodell des Szenarios erreicht z.B. eine Tiefe von bis zu 7 Hierarchien. Auf weitere Berechnungen wird an dieser Stelle jedoch verzichtet, da hier das Ausmaß der Menge an Regelsequenzen bereits verdeutlicht sein sollte.

Mit zunehmender Anzahl an Regelsequenzen erhöht sich auch der Speicherverbrauch zur Laufzeit. Einige Testmessungen zur Laufzeit des PEDTime-Prototyps[2] ergaben, dass zur Verarbeitung von z.B. 1'474'560 Regelsequenzen ca. 450MB Speicher benötigt werden. Bei 2'703'360 Regelsequenzen sind dies bereits über 800MB und bei 3'440'640 Regelsequenzen sind es ca. 1700MB. Um alle 7'864'320 Regelsequenzen des Szenarios verarbeitet zu können, werden ca. 2400MB Speicher benötigt. Für die vollständige Ausführung des Programms wird zusätzlicher Speicher benötigt, zu dessen Volumen auf Grund begrenzter verfügbarer Hardware keine genauen Angaben gemacht werden können.

Die Laufzeit, d.h. Dauer der Ausführung erweist sich bei ausreichend freigegebenem Speicher an die „Java Virtual Machine“ als unproblematisch. In durchgeführten Testmessungen dauerte die Verarbeitung der 7'864'320 Regelsequenzen bei 2500MB verfügbarem Speicher zwischen 26 und 32 Sekunden.

Die hier aufgezeigte Problematik ist insbesondere dann kritisch, wenn große Aufgabenmodelle mit wenigen definierten Constraints analysiert werden sollen. Denn die Berücksichtigung von – insbesondere reihenfolgebezogenen – Constraints kann die Anzahl der zu verarbeitenden Regelsequenzen stark verringern und dadurch

[2]Verwendetes Testsystem: MacBook Pro, Intel Core 2 Duo, 2,53 GHz, 4 GB RAM, Mac OS X 10.6.4

Speicherprobleme verhindern. Dies trifft z.B. auf das Szenario zu: Die Durchführung der kompletten Zeitanalyse durch den PEDTime-Prototypen von dem hier betrachteten Szenario inklusive der Berücksichtigung aller dafür definierten *nach*-Constraints verursachte keinen im Byte-Bereich messbaren Speicherbedarf und dauerte insgesamt ca. 400ms.

8 Zusammenfassung und Ausblick

In diesem Kapitel werden die wichtigsten Ergebnisse dieser Arbeit zusammengefasst. Darüber hinaus gibt das Kapitel einen Ausblick über Möglichkeiten zur sinnvollen Anknüpfung an diese Arbeit.

8.1 Zusammenfassung

In den ersten drei Kapiteln der vorliegenden Arbeit wurden die wesentlichen Grundlagen der Usability-Bewertung interaktiver Systeme mit Hilfe von Aufgabenanalysen in Verbindung mit konkreter zeitlicher Ablaufplanung dargestellt. Als zu erweiterndes System innerhalb dieser Arbeit wurden der *Procedure Editor* (PED) als Aufgabenmodellierungsumgebung und das ihm zugrunde liegende Regelsystem vorgestellt. Darüber hinaus wurde insbesondere auf unterschiedliche Ansätze zur Definition und Modellierung von Zeitkonzepten allgemein und speziell im Rahmen von Aufgabenmodellen eingegangen.

Auf der Basis dieses Wissens wurde in dieser Arbeit – angelehnt an ein Szenario aus der Luftfahrt – ein Konzept zur Zeitanalyse anhand von Aufgabenmodellen entwickelt, welches die Aufgabenanalyse mit PED erweitert. Nachdem das konkrete Cockpit-Szenario definiert wurde, folgte die Erweiterung des Regelsystems von PED. Dazu wurden zunächst Zeitkonzepte definiert, die im Kontext des Szenarios sinnvolle Modellierungsmöglichkeiten bieten. Dabei erwiesen sich vor allem mengenbezogene zeitliche Bedingungen zwischen Aufgaben als passend, da diese im sicherheitskritischen Anwendungsbereich besonders relevant sind, in dem die sekundengenaue Bearbeitung von Aufgaben einen entscheidenden Sicherheitsfaktor darstellen kann. Anschließend wurde die formale Syntax einer durch das Regelsystem beschriebenen Prozedur (entspricht einem Aufgabenmodell) hinsichtlich der

definierten Zeitkonzepte erweitert. Darauf aufbauend ist ein umfassendes Konzept zur Zeitanalyse mit PED, genannt PEDTime erstellt worden. Das Konzept beschreibt detailliert alle einzelnen Schritte, die erforderlich sind, um eine Zeitanalyse von einem mit PED modellierten Aufgabenmodell durchzuführen. Dabei werden die Erstellung von Aufgabensequenzen (in PEDTime durch Regelsequenzen repräsentiert), d.h. von einzelnen kompletten „Wegen“ durch das Aufgabenmodell und die Überprüfung der zeitlichen Bedingungen profund behandelt. Auch die anschließende Berechnung der minimalen, maximalen und medianen Ausführungszeiten der einzelnen Aufgabensequenzen ist Teil des Konzepts. Mögliche Probleme des Konzepts wurden insbesondere bei der sinnvollen Abbildung der durch das Regelsystem erlaubten, mächtigen Modellierungskonzepte auf für Zeitanalysen geeignete Konzepte aufgezeigt.

Schließlich wurde das Konzept algorithmisch umgesetzt, d.h., es wurde ein Prototyp implementiert, der eine eingeschränkte PEDTime-Zeitanalyse mit bestimmten zeitlichen Bedingungen des Konzepts ermöglicht. Als markantes Problem der Konzept-Umsetzung wurde auf den hohen Speicherverbrauch zur Laufzeit des Prototyps hingewiesen, der auf der großen Anzahl zu verarbeitender Aufgabensequenzen beruht und bereits bei Aufgabenmodellen von der Größe des Szenario-Aufgabenmodells relevant sein kann.

Die Anwendung des im Rahmen dieser Arbeit erarbeiteten Konzepts zur Zeitanalyse mit Berücksichtigung von zeitlichen Bedingungen kann zukünftige Aufgabenanalysen sinnvoll erweitern.

8.2 Ausblick

Da das Konzept auf eine Einbindung in PED ausgelegt ist, ist eine entsprechende Integration von PEDTime in PED der nächste naheliegende Schritt. Dazu sollten zunächst alle Teile des Konzepts vollständig umgesetzt werden. Dies beinhaltet die Implementierung weiterer Methoden zur Überprüfung aller im Konzept eingeplanten zeitlichen Bedingungen (Constraints), insbesondere der mengenbezogenen. Der im Rahmen dieser Arbeit entwickelte Prototyp überprüft alle reihenfolgebezogenen Constraints. Außerdem ist der Aufbau von Aufgabensequenzen dahingehend zu erweitern, dass beliebig tiefe Aufgabenhierarchien berücksichtigt werden

können. Der Prototyp behandelt bis zu vier Hierarchieebenen (s. Kapitel 7.4.4). Darüber hinaus erfordert die Integration von PEDTime in PED ebenso die Implementierung einer Methode zur Abbildung eines PED-Aufgabenmodells in das interne PEDTime-Format (s. Kapitel 6.2.2). Schließlich muss die GUI von PED die neue PEDTime-Funktionalität bereitstellen und u.a. auch die Ergebnisse einer Zeitanalyse geeignet präsentieren, wie schon in den Anforderungen in Kapitel 6.1.2 angemerkt wurde.

In Kapitel 6.1.2 wurde bereits erläutert, dass das Constraint *verschachtelt* auf Grund seiner Komplexität und Andersartigkeit bezogen auf die weiteren Constraints des Konzepts nicht weiter verfolgt wurde. Eine Ausweitung des hier vorgestellten Konzepts auf die Integration dieses Constraints würde völlig neuartige Aussagen basierend auf einer entsprechenden PEDTime-Zeitanalyse ermöglichen. Dadurch könnten zusätzlich die zeitlichen Auswirkungen von nebenläufig, d.h. ineinander verschachtelt ausgeführten Aufgaben analysiert werden.

Darüber hinaus kann die Berücksichtigung weiterer Constraints die Aussagekraft einer Zeitanalyse erweitern. Beispielsweise könnten konkrete Zeitpunkte von Aufgaben, wie Start- und Endzeitpunkt zueinander in Beziehung gesetzt werden (s. Ansatz von Vilain (1982), beschrieben in Kapitel 3.3.1) und dadurch exaktere Modellierungs- und Analysemöglichkeiten bieten. Ob und ggfs. welche zusätzlichen Constraints berücksichtigt werden, sollte von der Anwendungsdomäne abhängig gemacht werden, für die Zeitanalysen durchgeführt werden sollen.

Innerhalb einer PEDTime-Zeitanalyse wird geprüft, ob eine bestimmte Anordnung von Aufgaben bestimmte Constraints verletzt. Angenommen, es existiere keine einzige Anordnung von Aufgaben hinsichtlich des zu untersuchenden Aufgabenmodells, die alle definierten Constraints erfüllt, dann wäre das Ergebnis der entsprechenden Zeitanalyse leer. Ein solches Ergebnis erschwert dem Analysten nachzuvollziehen, welche Constraints letztlich die stärksten Auswirkungen auf den Ausgang dieses Ergebnisses hatten und welche Stellen innerhalb des Aufgabenmodells ggfs. zeitlich problematisch sind. Um auch in dem Fall, dass keine einzige Aufgabensequenz alle definierten Constraints erfüllt ein aussagekräftiges Ergebnis liefern zu können, ist z.B. die Darlegung von den konkreten verletzten Constraints erforderlich. Auch die explizite Angabe, welche Kombinationen von Constraints sich ggfs. widersprechen bzw. miteinander konkurrieren wäre in diesem Kontext wünschenswert, damit der Analyst u.a. auch auf eventuelle Fehler

bzw. Widersprüche bei der Modellierung aufmerksam gemacht werden kann.

In Kapitel 6.3 wurde bereits auf die Zeit eingegangen, die ein Mensch evtl. für die Auswertung eines Constraints benötigt. Es gilt zu klären, ob und inwiefern der Mensch Zeit für die kognitive Leistung, die mit der Auswertung bzw. Berücksichtigung von Constraints verbunden ist benötigt. Es ist anzunehmen, dass hierfür eine gewisse Zeit beansprucht wird. Darüber hinaus liegt z.B. die Vermutung nahe, dass die Auswertung reihenfolgebezogener Constraints weniger Zeit als die von mengenbezogenen Constraints beansprucht. Diesen Vermutungen sollte in weiteren Untersuchungen nachgegangen werden, um dann ggfs. zusätzliche Zeiten für Constraint-Auswertungen in die Berechnung der Ausführungszeiten einfließen lassen zu können.

Um vor allem für große Aufgabenmodelle den Speicherverbrauch während der Durchführung einer Zeitanalyse (s. Kapitel 7.5) einzuschränken, könnte das Konzept und entsprechend die Implementierung dahingehend erweitert werden, dass ggfs. vorhandenen Variablen im Vorfeld feste Werte zugewiesen werden. Dies hätte zur Folge, dass für die Zeitanalyse von vornherein ausschließlich jene Aufgabensequenzen betrachtet würden, die mit den entsprechenden Variablenwerten erreicht werden können, was wiederum den Speicherverbrauch zur Laufzeit der Zeitanalyse verringern würde.

Ist das komplette hier vorgestellte Konzept umgesetzt, wäre die anschließende Durchführung einer kompletten PEDTime-Zeitanalyse anhand des Szenarios erfreulich. Daraus resultieren sollte dann auch eine Einschätzung der Ergebnisse der Zeitanalyse hinsichtlich Realitätsnähe und Aussagekraft im Hinblick auf Usability-Bewertungen.

Angelehnt an die Möglichkeit, das Problem der Zeitanalyse als Ablaufplanungsproblem zu interpretieren (s. Kapitel 6.1.3) kann eine intensive Recherche zu Lösungsverfahren von Ablaufplanungsproblemen getätigt werden, um anschließend die Zeitanalyse mit Hilfe dieser bekannten Verfahren zu konzipieren. Ob dies jedoch zu einem besseren oder effizienteren als dem hier vorgestellten Konzept führen kann, ist auf Grund des bisherigen Kenntnisstands nicht vorherzusagen.

Literaturverzeichnis

[Adamczyk und Bailey 2004] Adamczyk, Piotr D. ; Bailey, Brian P.: If Not Now, When?: The Effects of Interruption at Different Moments Within Task Execution. In: *CHI 2004* Bd. 6, 2004, S. 271–278

[Allen 1983] Allen, James F.: Maintaining knowledge about temporal intervals. In: *Commun. ACM* 26 (1983), Nr. 11, S. 832–843. – ISSN 0001-0782

[Anderson 2001] Anderson, John R.: *Kognitive Psychologie.* 3. Spektrum Akademischer Verlag, 2001. – ISBN 978-3827410245

[Anderson et al. 2004] Anderson, John R. ; Bothell, Daniel ; Byrne, Michael D. ; Douglass, Scott ; Lebiere, Christian ; Qin, Yulin: An integrated theory of the mind. In: *Psychologial Review* 111 (2004), Nr. 4, S. 1036–1060

[Bailey und Iqbal 2008] Bailey, Brian P. ; Iqbal, Shamsi T.: Understanding Changes in Mental Workload during Execution of Goal-Directed Tasks and its Application for Interruption Management. In: *ACM Transactions on Computer-Human Interaction* Bd. 14, 2008

[Bolognesi und Brinksma 1987] Bolognesi, Tommaso ; Brinksma, Ed: Introduction to the ISO specification language LOTOS. In: *Comput. Netw. ISDN Syst.* 14 (1987), Nr. 1, S. 25–59. – ISSN 0169-7552

[Bovair et al. 1990] Bovair, Susan ; Kieras, David E. ; Polson, Peter G.: The Acquisition and Performance of Text-Editing Skill: A Cognitive Complexity Analysis. In: *Human Computer Interaction* Bd. 5, 1990, S. 1–48

[Brumby et al. 2007] Brumby, Duncan P. ; Howes, Andrew ; Salvucci, Dario D.: A cognitive constraint model of dual-task trade-offs in a highly dynamic driving task. In: *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems.* New York, NY, USA : ACM, 2007, S. 233–242. – ISBN 978-1-59593-593-9

[Brumby et al. 2009] Brumby, Duncan P. ; Salvucci, Dario D. ; Howes, Andrew: Focus on driving: how cognitive constraints shape the adaptation of strategy when dialing while driving. In: *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*. New York, NY, USA : ACM, 2009, S. 1629–1638. – ISBN 978-1-60558-246-7

[Card et al. 1983] Card, Stuart K. ; Moran, Thomas P. ; Newell, Allen: *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983. – ISBN 0-89859-859-1

[Carnegie Mellon University 2010] Carnegie Mellon University: *The CogTool Project*. 2010. – URL `http://cogtool.hcii.cs.cmu.edu/`. – Abruf: 21. Juni 2010

[Cutrell et al. 2001] Cutrell, Edward ; Czerwinski, Mary ; Horvitz, Eric: Notification, Disruption, and Memory: Effects of Messaging Interruptions on Memory and Performance. In: *Human Computer Interaction INTERACT '01*, IOS Press, 2001, S. 263–269

[DATech 2009] DATech Deutsche Akkreditierungsstelle Technik: *Leitfaden Usability*. 2009. – URL `http://www.datech.de/share/files/Leitfaden-Usability.pdf`. – Abruf: 18. Mai 2010

[de Kock et al. 2009] de Kock, Estelle ; van Biljon, Judy ; Pretorius, Marco: Usability evaluation methods: Mind the gaps. In: *SAICSIT'09*, 2009, S. 122–131

[Diaper und Stanton 2004] Diaper, Dan ; Stanton, Neville A.: *The handbook of task analysis for human-computer interaction*. Lawrence Erlbaum Associates, 2004. – ISBN 0-8058-4432-5

[Dix et al. 2004] Dix, Alan ; Finlay, Janet ; Abowd, Gregory D. ; Beale, Russell: *Human-Computer Interaction Third Edition*. Pearson Education Limited, 2004. – ISBN 0130-461091

[Eclipse Foundation 2010] Eclipse Foundation: *Eclipse IDE for Java Developers | Eclipse Packages*. 2010. – URL `http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/galileosr2`. – Abruf: 18. November 2010

[Fortmann 2008] Fortmann, Jutta: *Bewertung der Usability eines Fahrkartenautomaten mit Hilfe von Engineering-Techniken zur Quantifizierung von Benutzungsoberflächen*, Carl von Ossietzky Universität Oldenburg, Bachelorarbeit, 2008

[Freed et al. 2003] Freed, Michael ; Matessa, Michael ; Remington, Roger ; Vera, Alonso: How Apex Automates CPM-GOMS. In: *Proceedings of the Fifth International Conference on Cognitive Modeling*, ICCM, 2003, S. 93–98

[Giese et al. 2008] Giese, Matthias ; Mistrzyk, Tomasz ; Pfau, Andreas ; Szwillus, Gerd ; Detten, Michael von: AMBOSS: A Task Modeling Approach for Safety-Critical Systems. In: Forbrig, Peter (Hrsg.) ; Paternò, Fabio (Hrsg.): *Engineering Interactive Systems* Bd. 5247. Springer Berlin / Heidelberg, 2008, S. 98–109

[Gil et al. 2009] Gil, Guk-ho ; Kaber, David ; Kim, Sang-Hwan ; Kaufmann, Karl ; Veil, Theo: Modeling pilot cognitive behavior for predicting performance and workload effects of cockpit automation. In: *Proceedings of the 15th International Symposium on Aviation Psychology*, Wright State University, 2009, S. 101–106

[Gillie und Broadbent 1989] Gillie, Tony ; Broadbent, Donald: What makes interruptions disruptive? A study of length, similarity, and complexity. In: *Psychological Research* Bd. 50, Springer Verlag, 1989, S. 243–250

[Gray et al. 1994] Gray, Phil ; England, David ; McGowan, Steve: XUAN: enhancing UAN to capture temporal relationships among actions. In: *HCI '94: Proceedings of the conference on People and computers IX*. New York, NY, USA : Cambridge University Press, 1994, S. 301–312. – ISBN 0-521-48557-6

[Hartson und Gray 1992] Hartson, H. R. ; Gray, Philip D.: Temporal aspects of tasks in the user action notation. In: *Human-Computer Interaction* 7 (1992), Nr. 1, S. 1–45. – ISSN 0737-0024

[Janssen et al. 2010] Janssen, Christian P. ; Brumby, Duncan P. ; Garnett, Ray: Natural Break Points: Utilizing Motor Cues when Multitasking. In: *Proceedings of the 54th meeting of the Human Factors and Ergonomics Society*, Human Factors and Ergonomics Society, 2010

[Jin und Dabbish 2009] Jin, Jing ; Dabbish, Laura A.: Self-interruption on the computer: a typology of discretionary task interleaving. In: *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*. New York, NY, USA : ACM, 2009, S. 1799–1808. – ISBN 978-1-60558-246-7

[John 1990] John, Bonnie E.: Extensions of GOMS analyses to expert performance requiring perception of dynamic visual and auditory information. In: *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM, 1990, S. 107–116. – ISBN 0-201-50932-6

[John und Kieras 1994] John, Bonnie E. ; Kieras, David E.: *The GOMS Family of Analysis Techniques: Tools for Design and Evaluation*. 1994. – URL `ftp://ftp.eecs.umich.edu/people/kieras/GOMS/John-Kieras-TR94.pdf`. – Abruf: 26. Mai 2010

[Kahn et al. 2007] Kahn, Svenja ; Klug, Tobias ; Flentge, Felix: Modeling temporal dependencies between observed activities. In: *TMR '07: Proceedings of the 2007 workshop on Tagging, mining and retrieval of human related activity information*. New York, NY, USA : ACM, 2007, S. 27–34. – ISBN 978-1-59593-870-1

[Kieras 2001] Kieras, David: *Using the Keystroke-Level Model to Estimate Execution Times*. 2001. – URL `ftp://ftp.eecs.umich.edu/people/kieras/GOMS/KLM.pdf`. – Abruf: 25. November 2010

[Kieras 1996] Kieras, David E.: *A Guide to GOMS Model Usability Evaluation using NGOMSL*. 1996. – URL `ftp://ftp.eecs.umich.edu/people/kieras/GOMS/NGOMSL_Guide.pdf`. – Abruf: 7. Juni 2010

[Kieras 2006] Kieras, David E.: *A Guide to GOMS Model Usability Evaluation using GOMSL and GLEAN4*. 2006. – URL `ftp://www.eecs.umich.edu/people/kieras/GOMS/GOMSL_Guide.pdf`. – Abruf: 21. Juni 2010

[Kieras und Polson 1985] Kieras, David E. ; Polson, Peter G.: An approach to the formal analysis of user complexity. In: *International Journal of Man-Machine Studies* 22 (1985), Nr. 4, S. 365–394. – ISSN 0020-7373

[Kumar 1992] Kumar, Vipin: Algorithms for Constraint-Satisfaction Problems: A Survey. In: *AI Magazine* 13 (1992), S. 32–44. – URL `http://portal.acm.`

org/citation.cfm?id=140370.140377. – Abruf: 25. November 2010. – ISSN 0738-4602

[Lacaze und Palanque 2004] Lacaze, Xavier ; Palanque, Philippe: Comprehensive Handling of Temporal Issues in Tasks Models: What is needed and How to Support it? In: *CHI '04: Workshop on the Temporal Aspects Of Work For HCI*, 2004

[Lee und Taatgen 2002] Lee, Frank J. ; Taatgen, Niels A.: Multitasking as Skill Acquisition. In: *Proceedings of the twenty-fourth annual conference of the cognitive science society*, Lawrence Erlbaum Associates, 2002, S. 572–577

[Lewis et al. 1990] Lewis, Clayton ; Polson, Peter G. ; Wharton, Cathleen ; Rieman, John: Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In: *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 1990, S. 235–242. – ISBN 0-201-50932-6

[Lüdtke 2005] Lüdtke, Andreas: *Kognitive Analyse Formaler Sicherheitskritischer Steuerungssysteme auf Basis eines integrierten Mensch-Maschine-Modells*. Aka, 2005. – ISBN 3-89838-288-5

[Lüdtke et al. 2009] Lüdtke, Andreas ; Weber, Lars ; Osterloh, Jan-Patrick ; Wortelen, Bertram: Modeling Pilot and Driver Behavior for Human Error Simulation. In: *Proceedings of the 2nd International Conference on Digital Human Modeling: Held as Part of HCI International 2009*. Berlin, Heidelberg : Springer-Verlag, 2009 (ICDHM '09), S. 403–412. – ISBN 978-3-642-02808-3

[McFarlane 1999] McFarlane, D.: Coordinating the interruptions of people in human-computer interaction. In: *INTERACT '99*, IOS Press, Inc., 1999, S. 295–303

[Monk et al. 2008] Monk, Christopher A. ; Trafton, J. G. ; Boehm-Davis, Deborah A.: The Effect of Interruption Duration and Demand on Resuming Suspended Goals. In: *Journal of Experimental Psychology: Applied* Bd. 14, 2008, S. 299–313

[Mori et al. 2010] Mori, Giulio ; Paternò, Fabio ; Santoro, Carmen: *The ConcurTaskTrees Environment*. 2010. – URL http://giove.isti.cnr.it/tools/ctte/. – Abruf: 9. August 2010

[NASA 2006a] NASA: *APEX Autonomy Software Download*. 2006. – URL `http://sourceforge.net/projects/apex-autonomy/`. – Abruf: 22. Juni 2010

[NASA 2006b] NASA: *Nasa: Intelligent Systems: APEX*. 2006. – URL `http://ti.arc.nasa.gov/projects/apex/`. – Abruf: 21. Juni 2010

[Nielsen 1993] Nielsen, Jakob: *Usability Engineering*. Academic Press, 1993. – ISBN 0125184050

[Nielsen und Molich 1990] Nielsen, Jakob ; Molich, Rolf: Heuristic Evaluation of User Interfaces. In: *CHI '90 Proceedings*, 1990, S. 249–256

[Oppermann et al. 1992] Oppermann, R. ; Murchner, R. ; Reiterer, H. ; Koch, M.: *Software-ergonomische Evaluation. Der Leitfaden EVADIS II*. Walter de Gruyter Verlag, 1992

[Oracle 2010] Oracle: *Java SE Downloads - Sun Developer Network (SDN)*. 2010. – URL `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. – Abruf: 18. November 2010

[Paternò et al. 1997] Paternò, Fabio ; Mancini, Cristiano ; Meniconi, Silvia: ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In: *INTERACT '97: Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction*. London, UK, UK : Chapman & Hall, Ltd., 1997, S. 362–369. – ISBN 0-412-80950-8

[Polson et al. 1992] Polson, Peter G. ; Lewis, Clayton ; Rieman, John ; Wharton, Cathleen: Cognitive walkthroughs: A method for theory-based evaluation of user interfaces. In: *International Journal of Man-Machine Studies* Bd. 36, 1992, S. 741–773

[Rieman et al. 1995] Rieman, John ; Franzke, Marita ; Redmiles, David: Usability Evaluation with the Cognitive Walkthrough. In: *CHI'95 Mosaic of creativity*, 1995, S. 387–388

[Salvucci 2005] Salvucci, Dario D.: A Multitasking General Executive for Compound Continuous Tasks. In: *Cognitive Science* Bd. 29, 2005, S. 457–492

[Salvucci 2010] Salvucci, Dario D.: On reconstruction of task context after interruption. In: *CHI '10: Proceedings of the 28th international conference on*

Human factors in computing systems, ACM, 2010, S. 89–92. – ISBN 978-1-60558-929-9

[Salvucci und Bogunovich 2010] Salvucci, Dario D. ; Bogunovich, Peter: Multitasking and monotasking: the effects of mental workload on deferred task interruptions. In: *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, ACM, 2010, S. 85–88. – ISBN 978-1-60558-929-9

[Sarodnick und Brau 2006] Sarodnick, Florian ; Brau, Henning: *Methoden der Usability Evaluation.* Verlag Hans Huber, 2006. – ISBN 3-456-84200-7

[Sauer 2004] Sauer, Jürgen: *Intelligente Ablaufplanung in lokalen und verteilten Anwendungsszenarien.* B.G. Teubner Verlag, 2004. – ISBN 3-519-00473-9

[Sauro und Kindlund 2005] Sauro, Jeff ; Kindlund, Erika: A method to standardize usability metrics into a single score. In: *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems.* New York, NY, USA : ACM, 2005, S. 401–409. – ISBN 1-58113-998-5

[Shneiderman und Plaisant 2010] Shneiderman, Ben ; Plaisant, Catherine: *Designing the User Interface : Strategies for effective Human-Computer Interaction.* 5. Addison-Wesley/Pearson, 2010. – ISBN 0-321-60148-3

[Smith und Mosier 1986] Smith, Sydney L. ; Mosier, Jane N.: Guidelines for Designing User Interface Software / The MITRE Corporation. Bedford, MA, 1986. – Forschungsbericht

[Tsang 1993] Tsang, Edward: *Foundations of Constraint Satisfaction.* Academic Press, 1993. – URL `http://cswww.essex.ac.uk/Research/CSP/edward/FCS.html`. – Abruf: 25. November 2010. – ISBN 0-12-701610-4

[Tullis und Albert 2008] Tullis, Tom ; Albert, Bill: *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics.* Morgan Kaufmann, 2008. – ISBN 978-0-12-373558-4

[University of Paderborn 2009] University of Paderborn: *AMBOSS.* 2009. – URL `http://mci.cs.uni-paderborn.de/pg/amboss/`. – Abruf: 18. August 2010

[U.S. Dept. of Health and Human Services 2006] U.S. Dept. of Health and Human Services: *Research-based Webdesign- und Usability-Guidelines*. 2006. – URL `http://www.usability.gov/guidelines/guidelines_book.pdf`. – Abruf: 18. Mai 2010

[Vilain 1982] Vilain, Marc B.: A system for reasoning about time. In: *AAAI-82 Proceedings*, Bolt, Beranek and Newman, 1982, S. 197–201

[Wandmacher 2002] Wandmacher, Jens: *GOMS-Analysen mit GOMSED*. Mai 2002. – URL `http://www1.tu-darmstadt.de/fb/fb3/psy/kogpsy/GOMS-Analysen%20mit%20%20GOMSED.pdf`. – Abruf: 21. Juni 2010

[Wharton et al. 1992] Wharton, Cathleen ; Bradford, Janice ; Franzke, Marita: Applying Cognitive Walkthroughs to more complex user interfaces: experiences, issues and recommendations. In: *CHI'92*, 1992, S. 381–388

[Wickens 2002] Wickens, Christopher D.: Multiple resources and performance prediction. In: *Theoretical Issues in Ergonomics Science* Bd. 3, Taylor & Francis Ltd, 2002, S. 159–177

Teil I

Anhang: Aufgabenmodell des Szenarios in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org
/2001/XMLSchema-instance" xmlns:notation="http://www.eclipse.org/gmf/runtime/1.0.2/notation"
xmlns:procedure="procedure">
  <procedure:Procedure idCounter="47">
    <goals name="InitializeTask"/>
    <goals name="Monitor"/>
    <goals name="EditRoute"/>
    <goals name="MonitorA"/>
    <goals name="MonitorB"/>
    <goals name="MonitorC"/>
    <goals name="MonitorD"/>
    <goals name="HandleException"/>
    <goals name="CheckHorizontal"/>
    <goals name="CheckVertical"/>
    <goals name="GenerateTrajectory"/>
    <goals name="UpdateAutoPilot"/>
    <goals name="MonitorE"/>
    <goals name="MonitorA0"/>
    <goals name="MonitorB0"/>
    <goals name="MonitorC0"/>
    <goals name="MonitorD0"/>
    <goals name="MonitorE0"/>
    <goals name="GenerateTrajectory0"/>
    <goals name="CheckHorizontal0"/>
    <goals name="CheckVertical0"/>
    <goals name="UpdateAutoPilot0"/>
    <goals name="PressGenerate"/>
    <goals name="PressReject"/>
    <goals name="PerceptGoalCon"/>
    <goals name="PerceptGoalAlt"/>
    <goals name="PerceptGoalWp"/>
    <goals name="PressSendTo"/>
    <goals name="PerceptGoalWp"/>
    <goals name="PerceptGoalA"/>
    <goals name="PerceptGoalB"/>
    <goals name="PerceptGoalC"/>
    <goals name="PerceptGoalD"/>
    <goals name="PerceptGoalE"/>
    <rules lhsgoal="/0/@goals.0" rhsgoals="/0/@goals.1 /0/@goals.2"/>
    <rules lhsgoal="/0/@goals.1" rhsgoals="/0/@goals.3 /0/@goals.4 /0/@goals.5 /0/@goals.6
/0/@goals.12" id="1"/>
    <rules lhsgoal="/0/@goals.13" id="2">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.0" value="stateA"/>
      <lhs xsi:type="procedure:Condition" condition="stateA == adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.13" rhsgoals="/0/@goals.7" id="3">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.0" value="stateA"/>
      <lhs xsi:type="procedure:Condition" condition="stateA != adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.14" id="4">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.1" value="stateB"/>
      <lhs xsi:type="procedure:Condition" condition="stateB == adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.30" id="6">
      <rhs xsi:type="procedure:Percept" value="stateB" perceptedVariable="/0/@variables.1"/>
    </rules>
    <rules lhsgoal="/0/@goals.14" rhsgoals="/0/@goals.7" id="7">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.1" value="stateB"/>
      <lhs xsi:type="procedure:Condition" condition="stateB != adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.2" rhsgoals="/0/@goals.10 /0/@goals.11" id="8"/>
    <rules lhsgoal="/0/@goals.10" rhsgoals="/0/@goals.18 /0/@goals.28" id="9"/>
    <rules lhsgoal="/0/@goals.18" rhsgoals="/0/@goals.8 /0/@goals.9 /0/@goals.22" id="10">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.5" value="waypoints"/>
      <lhs xsi:type="procedure:Condition" condition="waypoints == adequate"/>
```

```
    </rules>
    <rules lhsgoal="/0/@goals.18" rhsgoals="/0/@goals.10 /0/@goals.23" id="11">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.5" value="waypoints"/>
      <lhs xsi:type="procedure:Condition" condition="waypoints != adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.25" id="13">
      <rhs xsi:type="procedure:Motor" value="1" instrumentVariable="/0/@variables.8"/>
      <rhs xsi:type="procedure:Percept" value="altitude" perceptedVariable="/0/@variables.10"/>
    </rules>
    <rules lhsgoal="/0/@goals.19" id="14">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.10" value="altitude"/>
      <lhs xsi:type="procedure:Condition" condition="altitude == adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.19" rhsgoals="/0/@goals.10" id="15">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.10" value="altitude"/>
      <lhs xsi:type="procedure:Condition" condition="altitude != adequate"/>
      <rhs xsi:type="procedure:Motor" value="1" instrumentVariable="/0/@variables.7"/>
    </rules>
    <rules lhsgoal="/0/@goals.26" id="17">
      <rhs xsi:type="procedure:Motor" value="1" instrumentVariable="/0/@variables.9"/>
      <rhs xsi:type="procedure:Percept" value="waypoints" perceptedVariable="/0/@variables.5"/>
    </rules>
    <rules lhsgoal="/0/@goals.20" rhsgoals="/0/@goals.27" id="18">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.5" value="waypoints"/>
      <lhs xsi:type="procedure:Condition" condition="waypoints == adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.20" rhsgoals="/0/@goals.10" id="19">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.5" value="waypoints"/>
      <lhs xsi:type="procedure:Condition" condition="waypoints != adequate"/>
      <rhs xsi:type="procedure:Motor" value="1" instrumentVariable="/0/@variables.7"/>
    </rules>
    <rules lhsgoal="/0/@goals.11" rhsgoals="/0/@goals.21 /0/@goals.24" id="20"/>
    <rules lhsgoal="/0/@goals.21" id="21">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.13"
value="atcConfirmation"/>
      <lhs xsi:type="procedure:Condition" condition="atcConfirmation == true"/>
      <rhs xsi:type="procedure:Motor" value="1" instrumentVariable="/0/@variables.12"/>
    </rules>
    <rules lhsgoal="/0/@goals.21" rhsgoals="/0/@goals.10" id="22">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.13"
value="atcConfirmation"/>
      <lhs xsi:type="procedure:Condition" condition="atcConfirmation != true"/>
    </rules>
    <rules lhsgoal="/0/@goals.31" id="23">
      <rhs xsi:type="procedure:Percept" value="stateC" perceptedVariable="/0/@variables.2"/>
    </rules>
    <rules lhsgoal="/0/@goals.15" id="24">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.2" value="stateC"/>
      <lhs xsi:type="procedure:Condition" condition="stateC == adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.15" rhsgoals="/0/@goals.7" id="25">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.2" value="stateC"/>
      <lhs xsi:type="procedure:Condition" condition="stateC != adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.32" id="26">
      <rhs xsi:type="procedure:Percept" value="stateD" perceptedVariable="/0/@variables.3"/>
    </rules>
    <rules lhsgoal="/0/@goals.16" id="27">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.3" value="stateD"/>
      <lhs xsi:type="procedure:Condition" condition="stateD == adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.16" rhsgoals="/0/@goals.7" id="28">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.3" value="stateD"/>
      <lhs xsi:type="procedure:Condition" condition="stateD != adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.33" id="29">
```

```
      <rhs xsi:type="procedure:Percept" value="stateE" perceptedVariable="/0/@variables.4"/>
    </rules>
    <rules lhsgoal="/0/@goals.17" rhsgoals="/0/@goals.1" id="30">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.4" value="stateE"/>
      <lhs xsi:type="procedure:Condition" condition="stateE == adequate"/>
    </rules>
    <rules lhsgoal="/0/@goals.17" rhsgoals="/0/@goals.7" id="31">
      <lhs xsi:type="procedure:MemoryRead" retrieveVariable="/0/@variables.4" value="stateE"/>
      <lhs xsi:type="procedure:Condition" condition="stateE != adequate "/>
    </rules>
    <rules lhsgoal="/0/@goals.29" id="32">
      <rhs xsi:type="procedure:Percept" value="stateA" perceptedVariable="/0/@variables.0"/>
    </rules>
    <rules lhsgoal="/0/@goals.22" id="34">
      <rhs xsi:type="procedure:Motor" value="1" instrumentVariable="/0/@variables.6"/>
    </rules>
    <rules lhsgoal="/0/@goals.23" id="36">
      <rhs xsi:type="procedure:Motor" value="1" instrumentVariable="/0/@variables.7"/>
    </rules>
    <rules lhsgoal="/0/@goals.24" id="37">
      <rhs xsi:type="procedure:Percept" value="atcConfirmation" perceptedVariable="/0
/@variables.13"/>
    </rules>
    <rules lhsgoal="/0/@goals.8" rhsgoals="/0/@goals.25 /0/@goals.19" id="38"/>
    <rules lhsgoal="/0/@goals.9" rhsgoals="/0/@goals.20 /0/@goals.26" id="39"/>
    <rules lhsgoal="/0/@goals.27" id="40">
      <rhs xsi:type="procedure:Motor" value="1" instrumentVariable="/0/@variables.11"/>
    </rules>
    <rules lhsgoal="/0/@goals.28" id="41">
      <rhs xsi:type="procedure:Percept" value="waypoints" perceptedVariable="/0/@variables.5"/>
    </rules>
    <rules lhsgoal="/0/@goals.3" rhsgoals="/0/@goals.29 /0/@goals.13" id="42"/>
    <rules lhsgoal="/0/@goals.4" rhsgoals="/0/@goals.30 /0/@goals.14" id="43"/>
    <rules lhsgoal="/0/@goals.5" rhsgoals="/0/@goals.15 /0/@goals.31" id="44"/>
    <rules lhsgoal="/0/@goals.6" rhsgoals="/0/@goals.32 /0/@goals.16" id="45"/>
    <rules lhsgoal="/0/@goals.12" rhsgoals="/0/@goals.33 /0/@goals.17" id="46"/>
    <variables name="stateA"/>
    <variables name="stateB"/>
    <variables name="stateC"/>
    <variables name="stateD"/>
    <variables name="stateE"/>
    <variables name="waypoints"/>
    <variables name="generateButton"/>
    <variables name="rejectButton"/>
    <variables name="horButton"/>
    <variables name="vertButton"/>
    <variables name="altitude"/>
    <variables name="sendToAtcButton"/>
    <variables name="engageButton"/>
    <variables name="atcConfirmation"/>
  </procedure:Procedure>
 </xmi:XMI>
```

Teil II

Anhang: Maße zur Bewertung von Usability

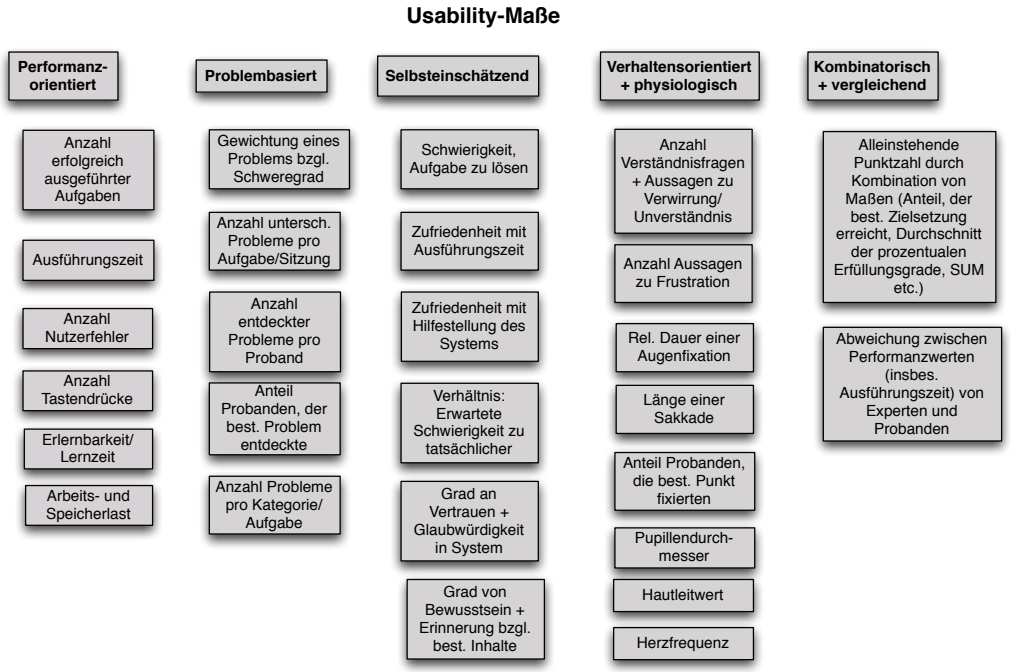
Maße zur Bewertung von Usability

Es gibt eine Reihe von Maßen, anhand derer versucht wird, die Usability eines Systems zu messen. Diese orientieren sich an den in DIN EN ISO 9241-11 aufgeführten Usability-Leitkriterien: “Effizienz, Effektivität und Zufriedenheit”. Allerdings erschweren u.a. die unpräzisen Definitionen von Usability die Ernennung eines allgemeingültigen Usabilitymaßes. Erstens ist die Gebrauchstauglichkeit eines Systems von zahlreichen, vor allem kontextbezogenen Faktoren abhängig und zweitens spielen dabei auch subjektive Empfindungen der Zielnutzer eine entscheidende Rolle.

Usability-Maße sind quantitativ messbare Kriterien, deren Messwerte es ermöglichen, Aussagen hinsichtlich der Usability eines Systems zu machen. Die Quantitativität ermöglicht zudem, verschiedene Systeme hinsichtlich bestimmter Usability-Maße zu vergleichen. Die auf der nächsten Seite folgende Abbildung zeigt eine Aufteilung von gängigen Usability-Maßen nach performanzorientiert, problembasiert, selbsteinschätzend, verhaltensorientiert und physiologisch, sowie kombinatorisch und vergleichend.

Performanz-Maße

Maße, die Aussagen hinsichtlich der Performanz der Interaktion zwischen Nutzer und System treffen, werden häufig zur Usability-Bewertung herangezogen. Dies liegt vor allem daran, dass sie anhand des Verhaltens von Nutzern erhoben werden und somit i.d.R. ein direkter Zusammenhang zwischen Messung und Verbesserungspotentialen erkennbar ist. Performanz-Maße werden gemessen, während Nutzer festgelegte Aufgaben mit dem zu testenden System bearbeiten, wie es bei



Usability-Tests (s. Anhang III) der Fall ist. Es folgt die Auflistung einiger gängiger Performanz-Maße.

Anzahl erfolgreich ausgeführter Aufgaben

Dieses Maß bewertet die Effektivität des Systems. Effektiv ist ein System dann, wenn der Nutzer in der Lage ist, seine Aufgaben erfolgreich auszuführen. Die Definition vom Aufgabenerfolg kann dabei unterschiedlich sein. Dieser kann z.B. in der Beendigung und kompletten Zielerfüllung einer Aufgabe liegen oder in Form von Abstufungen die Erreichung unterschiedlicher Teilziele oder Inanspruchnahme von Hilfe berücksichtigen.

Ausführungszeit

Die Ausführungszeit, die ein Nutzer zur Durchführung einer Aufgabe benötigt, gibt Hinweise auf die Effizienz eines Systems. Braucht ein Nutzer länger als angemessen für die Durchführung einer Aufgabe, so deutet dies i.d.R. auf eine ineffiziente Interaktion hin. Besonders wichtig ist die Effizienz bei wiederholt ausgeführten

oder sicherheitskritischen Aufgaben, bei denen im schlimmsten Fall Menschenleben von der Ausführungszeit abhängen können.

Anzahl Nutzerfehler

Fehler sind inkorrekte Aktionen, die zum Misserfolg einer Aufgabe führen können. Im schlimmsten Fall können sie folglich die Effektivität einer Interaktion beeinflussen. Aber auch wenn Fehler nicht zum Misserfolg einer Aufgabe führen, so können sie die Durchführung dieser erschweren und zeitlich verlängern. Sie sind daher insbesondere ein Maß für die Effizienz eines Systems. Macht der Nutzer Fehler während der Interaktion, kann dies für ein ineffizientes System sprechen.

Anzahl Tastendrücke

Der physische Aufwand, ausgedrückt z.B. in der Anzahl an Tastendrücken oder Mausklicks, kann Auskunft darüber geben, wie effizient eine Interaktion ist. Sind zur Durchführung einer Aufgabe viele Tastendrücke notwendig, kann dies auf eine ineffiziente Interaktion hindeuten. Je mehr Tastendrücke ein Nutzer benötigt, desto mehr Zeit wird die Bearbeitung der Aufgabe voraussichtlich auch in Anspruch nehmen.

Erlernbarkeit/Lernzeit

Die Erlernbarkeit eines Systems wird anhand des Aufwands ausgedrückt, mit dem ein Nutzer über einen Zeitraum Fertigkeiten im Umgang mit dem System erlernt. Für das Maß ist dabei wichtig, wieviel Zeit und kognitiven Aufwand der Nutzer investieren muss, um geübt mit dem System umgehen zu können. Das Maß macht folglich Aussagen über die Effizienz einer Interaktion. Erhoben werden kann die Lernzeit z.B. empirisch, indem die Ausführungszeiten von Nutzern bzgl. einer bestimmten Aufgabe mehrmals gemessen und addiert werden, bis die einzelne Ausführungszeit nahe der Zeit liegt, die ein Experte benötigt. Die Summe der bis dahin gemessenen Ausführungszeiten ergibt dann die Lernzeit, die der Nutzer für den erreichten Geübtheitsgrad benötigt hat.

Arbeits- und Speicherlast

Die kognitive Komplexität einer Interaktion kann durch die Arbeits- und die Speicherlast ausgedrückt werden, die bei der Durchführung von Aufgaben anfällt. Fallen diese Lasten gering aus, so unterstützt das System den Nutzer bei einer effizienten Interaktion. Die Arbeits- und Speicherlast eines Nutzers erhöht sich beispielsweise, wenn dieser Funktionen auf der Benutzungsoberfläche des Systems sucht, eine Entscheidung bzgl. der Ausführung einer Aktion trifft oder Ergebnisse ausgeführter Aktionen interpretiert. Gemessen werden kann diese Komplexität mit Hilfe formaler Analysen, bei denen z.B. die Anzahl von Informationsblöcken (sog. *Chunks*[1]), die sich zu einem bestimmten Zeitpunkt im Kurzzeitgedächtnis des Nutzers befinden, ermittelt wird.

Problembasierte Maße

Usability-Probleme sind konkrete Schwachstellen in der Interaktion, die den Nutzer bei der effektiven, effizienten, fehlerfreien oder zufriedenstellenden Ausführung einer Aufgabe behindern. Diese Schwachstellen können z.B. während Usability-Tests (s. Anhang III) aufgedeckt werden und liefern wertvolle Hinweise auf konkrete Optimierungspotentiale. Auch wenn aufgedeckte Usability-Probleme selbst qualitativer Natur sind, so können quantitative Aussagen über diese konkreten Probleme gemacht werden. Im Folgenden sind einige mögliche solcher problembasierten Maße aufgeführt:

Gewichtung eines Problems hinsichtlich seines Schweregrades

Ein aufgedecktes Usability-Problem kann hinsichtlich seines Einflusses auf den Aufgabenerfolg gewichtet werden. Diese Gewichtung wird i.d.R. in drei- oder fünfstufiger Form vorgenommen und hilft, die Bedeutung eines Problems für die Inter-

[1] Chunks sind Speichereinheiten im Kurzzeitgedächtnis, die aus Gruppierungen von zusammenhängenden Informationen bestehen. Es wird angenommen, dass maximal 7 +/- 2 Chunks im Kurzzeitgedächtnis gespeichert werden können, wobei die Informationsmenge pro Chunk sehr hoch sein kann.

aktion einzuschätzen und Prioritäten bei der Behebung der Probleme festzulegen. Die Gewichtung kann z.B. anhand des Einflusses auf Nutzerzufriedenheit, Aufgabenziel oder auf Grund der Häufigkeit, mit der das Problem aufgedeckt wird vorgenommen werden. Auch die Kombination mehrerer dieser Kriterien kann bei der Gewichtung sinnvoll sein.

Anzahl unterschiedlicher Probleme pro Sitzung/pro Aufgabe

Dieses Maß beschreibt, wieviele unterschiedliche Probleme von allen Probanden insgesamt aufgedeckt wurden. Dabei kann sich die Zahl der Probleme auf eine bestimmte Aufgabe beschränken, oder sie kann die Probleme einer kompletten Testsitzung einbeziehen. Aussagekräftig ist dieses Maß insbesondere bei der Darlegung der Veränderung der Usability eines Systems über mehrere Iterationen hinweg.

Anzahl entdeckter Probleme pro Proband

Auch die Anzahl an Problemen, die jeweils pro Proband aufgedeckt wurden, kann für die Bewertung der iterativen Veränderung der Usability eines Systems nützlich sein. Sinkt beispielsweise die Anzahl unterschiedlicher Probleme, während die der Probleme pro Proband konstant bleibt, so weist dies darauf hin, dass nur solche Probleme behoben wurden, die von einzelnen, wenigen Probanden erkannt wurden.

Anteil an Probanden, der bestimmtes Problem entdeckt

Der Anteil an Probanden, der ein bestimmtes Problem entdeckte, hilft zu klären, ob und inwieweit sich die Usability eines speziellen Designelements über mehrere Iterationen hinweg verändert hat.

Anzahl entdeckter Probleme pro Problem-Kategorie oder Aufgabe

Um Usability-Probleme genauer bestimmten Designproblemen der Benutzungsoberfläche zuordnen zu können, kann es sinnvoll sein, aufgedeckte Probleme hinsichtlich Kategorien oder Aufgaben einzuordnen. Dabei sind je nach Anwendung und Analyseziel die verschiedensten Kategorieeinteilungen, wie z.B. Navigation, Terminologie, Inhalt und Funktionalität, denkbar.

Selbsteinschätzende Maße

Selbsteinschätzende Maße sind Bewertungen, die die Nutzer aus ihrer Sicht subjektiv vornehmen. Dazu gehören persönliche Empfindungen, Emotionen und Einschätzungen bzgl. der Interaktion mit dem untersuchten System. Erhoben werden diese Maße daher überwiegend nach der Durchführung eines Usability-Tests. Dies geschieht insbesondere mit Hilfe von Fragebögen, die die Bewertungen in Form von bipolaren oder *Likert-Skalen*[2] abfragen und dadurch messbar und vergleichbar machen. Es gibt sehr viele Maße, die die Zufriedenheit des Nutzers bewerten können. Neben allgemeineren Fragen zur Performanz, können z.B. auch system- oder aufgabenspezifische Maße herangezogen werden. Auch Kombinationen verschiedener Maße, wie sie z.B. in standardisierten Fragebögen vorkommen, werden häufig zur Messung herangezogen. Es folgt die Auflistung einiger Beispiele für selbsteinschätzende Maße:

- **Empfundene Schwierigkeit, mit der eine Aufgabe gelöst wurde**
- **Zufriedenheit mit der benötigten Ausführungszeit pro bestimmter Aufgabe**
- **Zufriedenheit mit unterstützenden Informationen und Hilfen seitens des Systems**

[2]Die Likert-Skala dient dazu, die Einstellung einer befragten Person zu einem Thema zu erfassen. Bei Likert-Skalen handelt es sich um Ratingskalen, mittels derer Befragte auf einer mehrstufigen, ungeraden Intervallskala Zustimmung, Neutralität oder Ablehnung ausdrücken können.

- **Verhältnis zwischen erwarteter Schwierigkeit vor dem Test und tatsächlich erfahrener nach dem Test**
- **Grad des Vertrauens und der Glaubwürdigkeit in das System**
- **Grad des Bewusstseins für bzw. der Erinnerung an bestimmte Inhalte oder Designelemente nach Testdurchführung**

Verhaltensorientierte und physiologische Maße

Neben Leistungsmessungen, Problemeinordnung und Selbsteinschätzung der Nutzer können auch ihr Verhalten und physiologische Veränderungen während Usability-Tests beobachtet und gemessen werden. Solche verhaltensorientierten und physiologischen Maße können teilweise durch einfaches Beobachten erfasst werden, erfordern zur Messung aber teilweise auch spezielle Geräte, wie z.B. *Eye-Tracker*[3], die Blickbewegungen aufzeichnen und aufbereiten. Im Folgenden werden einige solcher Maße vorgestellt.

Anzahl gestellter Verständnis-Fragen und Aussagen, die Verwirrung und Unverständnis ausdrücken

Stellt ein Nutzer Fragen zum Verständnis während der Durchführung von Aufgaben im Rahmen eines Usability-Tests, so kann dies z.B. auf eine unzureichende Selbstbeschreibungsfähigkeit oder Schwächen in Struktur, Navigation oder verwendeten Terminologie des Systems hindeuten. Diese Fragen und entsprechenden Aussagen drücken Unsicherheit und Verständnisprobleme aus. Als Maß kann die Anzahl solcher auftgetretener Fragen und Aussagen ermittelt und zur Bewertung der Gesamt-Usability des Systems, herangezogen werden.

[3]Eye-Tracker sind Geräte und Systeme, die die Aufzeichnung von Blickbewegungen vornehmen und deren Analyse ermöglichen. Dabei werden Fixationen (anvisierte Punkte), Sakkaden (schnelle Augenbewegungen) und Regressionen (Sakkaden, die zu bereits erfassten Punkten zurückführen) erfasst.

Anzahl Aussagen, die Frustration ausdrücken

Dieses Maß spiegelt insbesondere die Nutzerzufriedenheit wieder und kann ebenfalls zur Bewertung der Gesamt-Usability herangezogen werden. Wie die bisher genannten verhaltensorientierten Maße, ist dieses Maß insbesondere für Usability-Vergleiche verschiedener Iterationen eines Systems geeignet.

Relative Dauer einer Augenfixation auf bestimmten Punkt

Die Dauer der Fixation der Augen auf einen bestimmten Punkt der Benutzungsoberfläche liefert Hinweise auf die Aufmerksamkeitsverteilung des Nutzers. Es kann festgestellt werden, ob und wie lange ein Nutzer einen bestimmten Punkt fixiert hat. Lang andauernde Fixationen verursachen häufig eine hohe kognitive Last, da sie i.d.R. dort auftreten, wo Nutzer viel Zeit zum Erfassen oder Analysieren der dargestellten Information benötigen. Für die Usability des untersuchten Systems bedeutet dies ggfs. Optimierungsbedarf hinsichtlich der Effizienz.

Länge einer Sakkade

Die Länge einer Augenbewegung zwischen zwei Fixationen (Sakkaden) kann ebenfalls Auskunft über die Effizienz der Interaktion geben. Denn es ist anzunehmen, dass der Nutzer die Augenbewegung zurücklegt, um eine bestimmte Information zu erhalten. Ist zum Erhalt dieser Information eine lange Augenbewegung erforderlich, so liegt vermutlich ein Optimierungspotential bzgl. der Weglänge, d.h. Effizienz vor.

Anteil Probanden, die bestimmten Punkt fixierten

Dieses Maß kann Aufschluss darüber geben, ob ein bestimmtes Element der Benutzungsoberfläche registriert wird und falls dem so ist, wie häufig dies geschieht.

Wird beispielsweise festgestellt, dass nur eine geringe Anzahl an Nutzern ein wichtiges Element bewusst wahrnimmt, so deutet dies auf die Notwendigkeit eines Redesigns hin, welches das entsprechende Element stärker in den Fokus den Nutzers setzt.

Prozentuale Abweichung des Pupillendurchmessers vom Normalwert

Eye-Tracker sind neben der Blickverfolgung ebenfalls in der Lage, den Pupillendurchmesser zu erfassen. Dieser gibt u.a. Auskunft über emotionale Empfindungen eines und kognitive Vorgänge im Menschen/s. Sind die Pupillen erweitert, kann dies bedeuten, dass die Person besonders erregt, interessiert oder kognitiv belastet ist, d.h. gerade Informationen verarbeitet. Somit kann dieses Maß auch zur Usability-Bewertung beitragen. Allerdings steht der Pupillendurchmesser in Wechselwirkung mit vielen Faktoren, sodass er ein unsicheres Maß im Rahmen von Usability-Tests darstellt.

Hautleitwert

Der Hautleitwert eines Menschens kann Hinweise auf den Stress liefern, den diese Person empfindet. Sind Menschen gestresst, produzieren sie Schweiß, der die Leitfähigkeit der Haut erhöht. Hinsichtlich der Usability eines Systems kann ein erhöhter Hautleitwert auf Schwächen im Systemdesign hindeuten, die die Nutzer frustrieren. Untersuchungen mit Webseiten haben diesen Zusammenhang bestätigt (Tullis und Albert, 2008, S. 183).

Herzfrequenz

Auch die Herzfrequenz ist ein Indikator für Stress. Fühlt sich ein Mensch gestresst, so erhöht sich dessen Herzfrequenz. Wie auch der Hautleitwert wurde dieses Maß bereits erfolgreich zur Usability-Bewertung von Webseiten eingesetzt. Allerdings sind die Messgeräte für Hautleitwert und Herzfrequenz noch sehr aufdringlich, da sie unmittelbar am Körper angeschlossen werden. Aus diesem Grund könnten sie

die Probanden zusätzlich unter Druck setzen und werden deshalb nur selten in Usability-Tests eingesetzt.

Kombinatorische und vergleichende Maße

Neben der Verwendung einzelner Usability-Maße ist auch die Kombination mehrerer Maße zu einem zusammengesetzten Usability-Maß möglich. Usability-Maße können auf verschiedene Arten miteinander zu einer alleinstehenden Usability-Punktzahl kombiniert werden. Ein anderer Ansatz ist der Vergleich von Experten- und Probandenwerten bzgl. der Performanz, die während der Interaktion erbracht wird. Auf beide Ansätze wird im Folgenden näher eingegangen.

Alleinstehende Usability-Punktzahl

Mehrere unterschiedliche Usability-Maße können zu einem neuen alleinstehenden Usability-Maß kombiniert werden. Dieses Maß kann dann zur Bewertung der Gesamt-Usability eines Systems oder auch von bestimmten Aspekten, die z.B. in Form von Aufgaben angesprochen werden, herangezogen werden. Die Kombination von Maßen, die jeweils in unterschiedlichen Maßeinheiten ausgedrückt werden, kann auf mehrere Arten vorgenommen werden. Beispielsweise kann der Anteil an Probanden ermittelt werden, der eine zuvor definierte Zielsetzung erreicht hat. Dabei kann es sich um Zielsetzungen wie z.B. die erfolgreiche Ausführung von 80% der Aufgaben in weniger als 70 Sekunden handeln. Ein anderer Ansatz ist die Kombination anhand der jeweils prozentual angegebenen Erfüllungsgrade der verschiedenen Größen. Dabei wird der Wert jedes Maßes einzeln entsprechend seiner speziellen Skala in einem Prozentwert ausgedrückt und anschließend der Durchschnitt aller dieser Prozentwerte berechnet. So ergibt sich ein neuer Prozentwert, der die Gesamt-Usability kombiniert aus allen verwendeten Einzelmaßen ausdrückt. Ein dritter Ansatz ist die Verwendung von z-Scores. Ein z-Score drückt aus, wieviele Einheiten ein Wert von dem Durchschnittswert einer Verteilung nach oben oder unten abweicht. Mehrere, auf unterschiedlichen Maßen basierende, z-Scores werden kombiniert, indem sie addiert und anschließend wiederum ihr Durchschnitt berechnet wird, sodass sich ein neuer Usability-Wert ergibt. z-Scores eignen sich

besonders um zwei Datensätze zu vergleichen. Mit SUM[4] stellen Sauro und Kindlund (2005) ein quantitatives Modell vor, mit dem mehrere Usability-Maße in Form einer alleinstehenden Usability-Punktzahl kombiniert werden. Dabei werden Effektivität, Effizienz und Zufriedenheit in Form der Maße Aufgabenerfolg, Ausführungszeit, Fehleranzahl und Zufriedenheitsbewertung berücksichtigt. Die Methode eignet sich sowohl zur Bewertung der Gesamt-Usability einzelner Aufgaben, als auch zu der eines kompletten Aufgabensatzes bzw. Systems.

Vergleich mit Expertenwerten

Die Usability-Bewertung kann auch anhand des Vergleichs von Probanden- und Expertendaten vorgenommen werden. Finden z.B. Performanzmessungen statt, so können diese mit zuvor erhoben Expertenwerten verglichen werden. Dadurch kann festgestellt werden, wie stark die Probandenwerte von den Expertenwerten abweichen und ob dementsprechend ggfs. Optimierungsbedarf besteht. Als Performanz-Maß für solche Vergleiche bietet sich insbesondere die Ausführungszeit an, da andere Maße wie Fehleranzahl oder Aufgabenerfolg als Expertenwert generell „null“ bzw. voller Erfolg definieren und somit keine überraschenden, neuen Erkenntnisse bieten. Generell gilt: Je geringer die Abweichung der Probandenwerte von denen der Experten, desto besser ist die Usability des Systems.

Allgemein können die meisten genannten Usability-Maße sowohl aufgabenspezifisch als auch für die gesamte Sitzung des zugrundeliegenden Usability-Tests erhoben werden.

[4]Single Usability Metric

Teil III

Anhang: Methoden zur Usability-Evaluation

Methoden zur Usability-Evaluation

Die Ausprägungen von Usability-Maßen, die mit Hilfe sogenannter Usability-Evaluationen erhoben werden, kann sowohl von kontextbezogenen Faktoren, als auch von subjektiven Empfindungen der Zielnutzer abhängig sein. Darüber hinaus kann auch die Erfahrung und das Engagement der Analysten entscheidend für den Ausgang der Messung sein (de Kock et al., 2009). Die Usability-Bewertung eines Systems kann grob von zwei unterschiedlichen Perspektiven aus durchgeführt werden. Einerseits können konkrete Usability-Probleme identifiziert werden, andererseits kann eine Aussage über die Gesamt-Usability des Systems vorgenommen werden. Zur Aufdeckung konkreter Usability-Probleme dient die formative Evaluation, die meistens während der Entwicklung eines Systems an Prototypen vorgenommen wird und Hinweise auf sinnvolles Redesign gibt. Bei der summativen Evaluation wird i.d.R. ein existierendes System hinsichtlich seiner allgemeinen Gebrauchstauglichkeit analysiert (Sarodnick und Brau, 2006, S. 20).

Im Folgenden werden einige gängige Methoden, die zur Usability-Evaluation eingesetzt werden, vorgestellt.

Gestaltungsrichtlinien

Gestaltungsrichtlinien bestehen aus einzelnen Prinzipien, die Gestaltungsanforderungen formulieren und gehören zu den ersten Usability-Evaluationsmethoden. Die Berücksichtigung dieser Anforderungen soll zu einer guten Usability des betrachteten Systems beitragen. Gestaltungsrichtlinien können unterschiedlich ausgelegt sein. Während einige beispielsweise sehr grob formuliert sind, beziehen sich andere auf spezifische Designentscheidungen. Auch gibt es z.B. Richtlinien, die zur Vereinheitlichung von Systemen führen sollen, d.h. sog. *Styleguides*, wie sie

z.B. für Programme verwendet werden, die für ein bestimmtes Betriebssystem konzipiert sind, während andere aus vielen, nach Kategorien geordneten allgemeinen Anforderungen bestehen. Bei der Usability-Bewertung analysieren ein oder mehrere Evaluatoren das System hinsichtlich der Gestaltungsrichtlinen. Es wird überprüft, ob und inwiefern die Richtlinien berücksichtigt wurden. Dabei gibt es keine einheitlichen Vorgaben hinsichtlich des Vorgehens oder der Dokumentation bei der Analyse. Der Vorteil einer solchen Evaluation ist die Bandbreite an Usability-Metriken, die dabei berücksichtigt wird. Je nach gewählten Richtlinien ist es möglich, in nahezu alle Bereiche der qualitativen Schnittstellenbewertung Einblick zu erhalten. Da die Wahl der Richtlinen ausschlaggebend für den Erfolg dieser Methode ist, sollte stark auf die Qualität der Herkunft geachtet werden. Der weiten Bandbreite gegenüber steht der hohe Evaluationsaufwand und der Einfluss der Kompetenz des Analysten auf die Evaluationsergebnisse (Sarodnick und Brau, 2006). Da die Richtlinien i.d.R. in keinem bestimmten Nutzungskontext stehen, sondern allgemein anwendbar sind, können sie keine Aussagen über die spezielle Interaktion während einer bestimmten Aufgabe machen. Dieser Mangel hat die Anwendung von Gestaltungsrichtlinien seltener gemacht, da sich im Laufe der Jahre gerade der Nutzungskontext als entscheidender Usability-Einfluss herausgestellt hat (vgl. DIN EN ISO 9241-11 in Kapitel 1.1). Bekannte Beispiele für Gestaltungsrichtlinien sind die *Guidelines for Designing User Interface Software* (Smith und Mosier, 1986) oder die *Research-based Webdesign- und Usability-Guidelines* (U.S. Dept. of Health and Human Services, 2006).

Heuristische Evaluation

Der heuristische Ansatz greift auf Heuristiken zurück, um Gestaltungsmängel aufzudecken. Diese Heuristiken sind Regeln, die bestimmte Aspekte beschreiben, die bei der Gestaltung von Nutzerschnittstellen berücksichtigt werden sollten, damit diese gebrauchstauglich sind. Es gibt verschiedene Heuristiken, auf deren Grundlage eine Evaluation durchgeführt werden kann. Nielsen (1993) zählt die, seiner Ansicht nach zehn wichtigsten Heuristiken auf. Diese sind

1. Sichtbarkeit des Systemstatus
2. Übereinstimmung zwischen System und Realität

3. Benutzerkontrolle und -freiheit
4. Konsistenz und Standards
5. Fehlervermeidung
6. Wiedererkennen vor Erinnern
7. Flexibilität und effiziente Nutzung
8. Ästhetisches und minimalistisches Design
9. Unterstützung beim Wiedererkennen, Verstehen und Bearbeiten von Fehlern
10. Hilfe und Dokumentation

Andere bekannte Heuristiken sind z.B. *Shneiderman's Eight Golden Rules Of Interface Design* (Shneiderman und Plaisant, 2010). Ein neuerer Satz allgemeiner Heuristiken, der sich stark an der DIN EN ISO 9241-10 orientiert und neue Erkenntnisse berücksichtigt, wird von Sarodnick und Brau (2006, S. 140f) vorgestellt. Anhand der gewählten Heuristiken untersuchen drei bis fünf Experten unabhängig voneinander die Bedienerführung eines Systems (Nielsen und Molich, 1990). Sie versuchen aus der Perspektive potentieller Nutzer des Systems Verstöße gegen die Heuristiken zu finden. Nachdem diese Untersuchungen durchgeführt wurden, werden alle Beobachtungen zusammengefasst und den einzelnen Heuristiken zugeordnet. Anschließend werden die Beobachtungen hinsichtlich ihres Schweregrades bewertet. Dabei wird berücksichtigt, wie oft ein Problem auftritt und wieviel Einfluss es auf die Bedienung des Geräts hat. So kann eine Prioritätenliste erstellt werden, nach der die einzelnen Probleme behoben werden sollen. Abschließend formulieren die Experten Optimierungsvorschläge zu jedem Problem.

Die heuristische Evaluation gilt als kostengünstige und wenig Zeit benötigende Evaluationsmethode. Sie eignet sich besonders, um Usability-Erkenntnisse während der frühen Entwicklungsphase eines Systems zu erhalten. Ein Kritikpunkt ist allerdings, dass die Experten, die diese Untersuchungen durchführen, keine tatsächlichen Nutzer des Geräts sind und somit i.d.R. nicht über nötiges Domänenwissen oder Erfahrungen bzgl. der Benutzungsoberfläche verfügen, von denen jedoch ausgegangen wird, dass reale Nutzer sie hätten. Auch wenn die Analysten versuchen, die Untersuchungen aus Nutzersicht vorzunehmen, so hängt die Qualität der heuristischen Evaluation stets von der Fähigkeit, dem Domänenwissen

und der Denkart der Analysten ab und sollte mit weiteren Methoden kombiniert werden (Nielsen, 1993; Sarodnick und Brau, 2006; Nielsen und Molich, 1990).

Expertenleitfäden

Expertenleitfäden sind streng definierte Prüfverfahren, die von Usability-Experten durchgeführt werden. Dabei untersuchen und bewerten diese ein System anhand von Fragen und Aussagen, die in Form von Prüflisten vorliegen. Wie auch bei den Gestaltungsrichtlinien steht hier der Nutzungskontext nicht im Vordergrund, sondern die Aufdeckung von gestalterischen Usability-Mängeln. Im Gegensatz zu den Gestaltungsrichtlinien ist die Durchführung von Expertenleitfäden streng vorgeschrieben. Im deutschsprachigen Raum ist EVADIS[5] II (Oppermann et al., 1992) ein bekannter Expertenleitfaden. Dieser umfassende Leitfaden kombiniert verschiedene Evaluationsmethoden. Neben der Bewertung anhand der typischen Prüfliste finden eine Nutzerbefragung mittels Fragebögen, sowie darauf aufbauend eine Arbeitsanalyse statt. Die Arbeitsanalyse dient der Festlegung des Evaluationsfokusses und hilft bei der Auswahl einer geeigneten Prüfaufgabe. Die Prüfaufgabe gibt dem Analysten vor, wie er bei der Bewertung mittels der Prüflisten durch das System navigieren soll. So stellt EVADIS II ein umfassendes Konzept dar, das auch auf den Nutzungskontext eingeht. Allerdings geschieht die Usability-Bewertung dabei nur in der Hinsicht, dass der Erfüllungsgrad der Gestaltungsrichtlinen nach DIN EN ISO 9241-10 in Prozentwerten dargestellt wird, aber keine genauen Problembeschreibungen ermittelt werden. Als Nachteil stellt sich der hohe zeitliche Aufwand dar, den diese Methode erfordert (Sarodnick und Brau, 2006).

Cognitive Walkthrough

Das vermutliche bekannteste Usability-Walkthrough-Verfahren ist der *Cognitive Walk-through* (Lewis et al., 1990). Dabei handelt es sich um eine Inspektionsmethode, bei der das explorative Verhalten von Laien bzw. die Erlernbarkeit eines

[5] Evaluation von Dialogsystemen

Systems erforscht wird. Dazu durchlaufen einige Usability-Experten einen simulierten Weg eines Nutzers durch die Bedienerführung einer Benutzungsoberfläche. Dieser Weg wird zuvor für jede mögliche Aufgabe als ideeller Pfad durch die Nutzungsschnittstelle festgelegt. Die Analysten versetzen sich in die Nutzer und analysieren die vorgegebenen Handlungsabläufe. Sie treffen Vorhersagen über die Wahrscheinlichkeit, mit der Nutzer die ideellen Pfade tatsächlich selbstständig durchlaufen werden. Die Analyse geschieht unter der Annahme, dass ein Nutzer stets den Weg des geringsten kognitiven Aufwands gehen wird. Dazu wird für jeden ideellen Interaktionspfad überprüft, ob dieser auch mittels beliebig häufiger Wiederholungen der vier kognitiven Schritte aus der CE+–Theorie explorativen Lernens (Polson et al., 1992) vermutlich gewählt werden wird. Diese vier kognitiven Schritte beschreiben die Interaktion eines Menschens mit einem Computer wie folgt:

1. Der Nutzer legt ein Ziel fest, das mit dem System erreicht werden soll.
2. Der Nutzer durchsucht die Schnittstelle nach momentan verfügbaren Aktionen.
3. Der Nutzer wählt eine Aktion aus, mit der er dem Ziel vermutlich näher kommt.
4. Der Nutzer führt die ausgewählte Aktion aus und wertet die Rückmeldung des Systems aus, um zu überprüfen, ob er sich dem Ziel tatsächlich nähert.

Diese Methode erfordert, dass die Experten Kenntnis über vermutliches Vorwissen und Verhalten der Nutzer haben, was wiederum eine aufwendige Vorbereitungsphase vor dem eigentlichen Walkthrough voraussetzt. Geeignet ist ein Cognitive Walkthrough inbesondere, um herauszufinden, wie leicht die Bedienung eines Geräts erlernt werden kann. Außerdem reichen frühe Designmodelle bzw. Prototypen zur Evaluation aus. Anhand der Ergebnisse eines Cognitive Walkthrough können Rückschlüsse auf Schwachstellen des Designs gezogen werden. Es ist festzustellen, ob, wo und warum das verwendete Design die Interaktion zwischen Nutzer und System stört und welche Anforderungen und Spezifikationen sich daraus für das System ergeben. Dabei ist zu betonen, dass diese Methode nicht geeignet ist, um Probleme mit einer Schnittstelle zu identifizieren, sondern um Diskrepanzen zwischen Nutzungshinweisen bzw. -vorgaben durch das System und den Zielen aus Nutzersicht aufzudecken (Wharton et al., 1992; Rieman et al., 1995). Am Ende

der Inspektion werden Verbesserungsvorschläge für aufgedeckte Problemstellen erarbeitet (Sarodnick und Brau, 2006).

Fragebögen

Fragebögen dienen der Erhebung subjektiver Empfindungen oder soziodemografischer Daten der Nutzer eines Systems. Soziodemografische Daten wie Alter, Geschlecht, Beruf oder Computererfahrung helfen den Analysten, die Zielgruppe sicherzustellen, Details zum Nutzerwissen zu erhalten und dadurch z.B. Erklärungen für unterschiedliche Verhaltensweisen bei der Interaktion mit einem System zu finden. Aus diesem Grund werden diese Tests häufig vor der Durchführung von Usability-Tests eingesetzt. Soll die Interaktion eines Systems bewertet werden, so setzt dies die vorherige Auseinandersetzung mit diesem voraus. Daher werden Fragebögen zur Erhebung der Nutzerempfindungen oft nach oder während der Durchführung eines Usability-Tests eingesetzt. Dabei können Fragebögen auf bestimmte Aufgabenbereiche während der Interaktion abzielen oder allgemeine Fragen zu dem Umgang mit dem System stellen. Neben standardisierten Fragebögen, die nummerische Antworten anhand bipolarer Skalen erfordern, können auch solche eingesetzt werden, die *Multiple Choice* verwenden oder freie Antwortmöglichkeiten zulassen. Für die Vergleichbarkeit und einfachere Auswertung der Ergebnisse bietet sich die Verwendung ersterer Fragebögen an. Die Konstruktion eigener Fragebögen ist sehr aufwändig und unterliegt hohen wissenschaftlichen Ansprüchen. Es gibt bereits einige veröffentlichte, standardisierte Fragebögen, die wissenschaftlich fundiert erstellt wurden und bereits auf zahlreiche verschiedene Systeme angewendet wurden. Als bekannte Beispiele seien an dieser Stelle QUIS[6], SUMI[7], IsoNorm 9241/10 und SUS[8] genannt. Während QUIS und SUMI verstärkt bzw. ausschließlich auf die subjektive Nutzerzufriedenheit mit einem System eingehen und diese u.a. anhand von persönlichen Gefühlen und Eindrücken messen, formulieren IsoNorm und SUS Fragen, die nach persönlichen Einschätzungen hinsichtlich des Systems fragen und orientieren sich streng an den sieben Grundsätzen aus der ISO-Norm 9241-110 (Sarodnick und Brau, 2006). SUS hat sich auch für

[6] Questionnaire for User Interaction Satisfaction
[7] Software Usability Measurement Inventory
[8] System Usability Scale

Tests mit einer geringen Anzahl an Probanden (z.B. 8-10) als robust erwiesen (Tullis und Albert, 2008).

Fragebögen geben keine konkreten Fehlerbeschreibungen, sondern evtl. Hinweise auf von den Nutzern erkannte Schwachstellen des Systems. Sie sind ohne viel Aufwand flexibel einsetzbar, objektiv hinsichtlich des geringen Evaluatoreinflusses auf die Ergebnisse und bieten eine einfache Möglichkeit, die subjektiven Eindrücke und die Zufriedenheit von Nutzern zu ermitteln. Wie allgemein für alle Methoden gilt auch für Fragebögen, dass sie weniger als eigenständiges Usability-Maß zu verwenden sind, sondern als nützliche Ergänzung zu anderen Methoden wie z.B. Usability-Tests (Shneiderman und Plaisant, 2010).

Aufgabenanalyse

Die Aufgabenanalyse wurde bereits in Abschnitt 2.2 vorgestellt. Im Rahmen dieses Abschnitts ist sie insbesondere als Verfahren zur Usability-Evaluation interessant. Aufgabenanalysen können helfen, Schwächen in der Usability eines Systems aufzudecken. Anhand der Ergebnisse einer solchen Aufgabenanalyse muss der Analyst Hypothesen aufstellen, die die Fehler erklären und letztlich zu Verbesserungsvorschlägen führen, die vom Analysten formuliert werden. Eine anschließende Validierung der aufgestellten Hypothesen ist empfehlenswert. Eine Aufgabenanalyse ist u.a. folglich nur ein Hilfsmittel zur Aufdeckung von Problemen, liefert aber keine Erklärungen oder Lösungsvorschläge. Somit sind die Erfahrung und das Wissen des Analysten entscheidend für den Erfolg einer solchen Evaluation (Diaper und Stanton, 2004).

Im Folgenden wird genauer auf die Hierarchische Aufgabenanalyse als spezielles Verfahren eingegangen.

Bei der Hierarchischen Aufgabenanalyse (Diaper und Stanton, 2004) wird eine Aufgabe funktional in Form von Operationen bzw. Zielen betrachtet. Die Aufgabe stellt das oberste Ziel dar, welches in Subziele zerlegt wird, deren Erreichung bzw. Ausführung zusammen wiederum das oberste Ziel ergibt. Subziele können in weitere Subziele zerlegt werden, sodass eine hierarchische Struktur entsteht, bei der die unterste Ebene die feingranularsten Ziele beinhaltet, die zur Erreichung des obersten Ziels notwendig sind. Je nach Ziel der Analyse kann der Detailgrad der

untersten Ebene unterschiedlich ausgeprägt sein. Häufig sind die elementarsten Ziele Operationen wie motorische oder kognitive Handlungen. An die Ausführung einer Operation können Konditionen geknüpft sein. Eine Kondition kann beispielsweise darin bestehen, dass erst ein Ziel erreicht sein oder dass eine Variable einen bestimmten Wert haben muss, bevor ein bestimmtes anderes Ziel bearbeitet werden darf. Auch das Verhalten an Abzweigungen innerhalb der Hierarchie wird durch solche, an Bedingungen geknüpfte, Regeln gesteuert. Daneben ist auch die Modellierung von Zeitkonzepten denkbar. So kann z.B. spezifiziert werden, ob bestimmte Ziele untereinander nur sequentiell oder auch parallel ausgeführt werden können.

Die Durchführung einer hierarchischen Aufgabenanalyse geschieht i.d.R. in sieben Schritten (Diaper und Stanton, 2004, S. 73), die jeweils, je nach Ziel der Analyse stärker oder schwächer berücksichtigt werden können:

1. Stelle die Absichten der Analyse fest.
2. Definiere mit allen Interessenvertretern einheitliche Aufgabenziele und Kriterien-Maße.
3. Identifiziere verfügbare Quellen für die Sammlung von Aufgabeninformationen und wähle Mittel zur Datenakquise aus.
4. Erwerbe die Daten und zerlege diese. Halte diese Zerlegung in Form eines Diagramms oder einer Tabelle fest.
5. Überprüfe die Validität der Zerlegung zusammen mit den Interessenvertretern.
6. Ermittle signifikante Operationen angesichts der Absichten der Analyse.
7. Erstelle und, sofern möglich, teste Hypothesen bzgl. Faktoren, die das Erlernen und die Leistung beeinflussen.

Die Absichten der Analyse haben Einfluss auf die Art der Datenerhebung, die Ergebnisse und die Granularität der Zielhierarchie. Sind die Absichten definiert, so können die groben Ziele mit allen Interessenvertretern gemeinsam formuliert werden. Auch Leistungsindikatoren und -kriterien sollten geklärt werden. Dabei ist z.B. zu diskutieren, welches objektive Kriterium belegt, dass ein Ziel erreicht wurde oder welche Auswirkungen es hat, wenn ein Ziel nicht erreicht wird. Im

nächsten Schritt werden Datenquellen ermittelt, anhand derer die Aufgabenanalyse durchgeführt werden soll. Das können z.B. Aufzeichnungen früherer Interaktionen (gewählte Pfade, Fehler- und Erfolgsraten), Interviews mit Systemdesignern und -experten, Nutzerobservationen oder Aufzeichnungen von Simulationen und Experimenten sein. Schritt vier sieht die Erstellung der Zielhierarchie mit Hilfe der erworbenen Daten vor. Um aussagekräftig zu sein, muss die Zerlegung stark nutzerorientiert vorgenommen werden. Dabei gilt es zuzuordnen, was die Benutzer machen (Operationen/Ziele), warum sie dies tun (Erreichen von Zielen) und welche Auswirkungen es hat, wenn sie dies nicht korrekt machen (Bedingungen). Valide gilt eine Zerlegung, wenn alle Subziele untereinander exklusiv und zusammen vollständig sind, d.h. dass alle Subziele zusammen das entsprechende übergeordnete Ziel definieren. Notiert wird die Zielhierarchie i.d.R. in Form eines Hierarchie-Diagramms oder einer Tabelle. Auch Ablaufpläne und Haltepunkte (Ebene mit dem höchsten Detailgrad) sollten in der Notation verdeutlicht werden. Im Anschluss sollte die Überprüfung der Zerlegung unter Absprache mit den Interessenvertretern vorgenommen werden. Bestenfalls geschieht dies in einem iterativen Prozess, der letztlich eine verlässliche und ausgereifte Analyse ermöglicht. Nach der Überprüfung ermittelt der Analyst signifikante Operationen bzw. Ziele unter Berücksichtigung der Analyseabsicht. Solche Operationen sind z.B. jene, deren Scheitern erhebliche negative Konsequenzen auf den gesamten Systemablauf hat. Auch Operationen, die eine hohe Arbeitsbelastung, Teamarbeit oder spezielles Wissen erfordern, können signifikante Operationen sein. Der letzte Analyseschritt besteht darin, die aufgedeckten Fehlerquellen bestimmten Fähigkeiten, Regeln oder Wissen zuzuordnen. Diese Zuordnungen helfen anschließend plausible Lösungen bzw. Verbesserungen vorzuschlagen. Sofern möglich sollten die Vorschläge hinterher validiert werden.

Eine bekannte Evaluationsmethode, die auf der Hierarchischen Aufgabenanalyse aufbaut, ist GOMS[9]. In GOMS wird ein Aufgabenmodell anhand von Zielen, Operatoren, Methoden und Selektionsregeln repräsentiert. Ziele sind die Aufgaben, die der Nutzer durchführen möchte. Das oberste Ziel stellt die Hauptaufgabe da und wird in Unterziele aufgebrochen, die die Teilaufgaben repräsentieren, die der Nutzer erfüllen muss, um das oberste Ziel zu erreichen. So entsteht die für eine Hierarchische Aufgabenanalyse typische Aufgaben- bzw. Zielhierarchie. Die Operatoren sind die feingranularsten Ziele, d.h. die Ziele der untersten Hierarchieebe-

[9] Goals, Operators, Methods, Selection Rules

ne, die z.B. einzelne Motoroperationen repräsentieren. Methoden sind Sequenzen von Operatoren und Unterzielen, die ausgeführt werden müssen, um ein bestimmtes Ziel zu erreichen. Selektionsregeln werden benötigt, wenn mehrere Methoden bzw. Unterziele geeignet sind, um ein bestimmtes Ziel zu erreichen. Die Regeln klären dann, welche Methode ausgeführt bzw. welches Ziel verfolgt wird.

Das ursprüngliche CMN[10]-GOMS wurde zu einer Familie von GOMS-Verfahren weiterentwickelt. Die Verfahren der GOMS-Familie (John und Kieras, 1994) ermöglichen u.a. die Abschätzung von Ausführungs- und Lernzeiten einer Aufgabe, die in Form eines hierarchischen Aufgabenmodells organisiert wird. Die Grundidee von GOMS ist das Prinzip der hierarchischen Aufgabenanalyse: die Interaktion zwischen Mensch und Maschine auf elementare Aktionen zu reduzieren und den Ablauf dieser in einem Modell darzustellen um dann die Effizienz dieser Schritte zu ermitteln. Anzumerken ist hier, dass Dix, Finlay, Abowd und Beale (2004) die Aufgabenanalyse von zielorientierten kognitiven Ansätzen wie GOMS abgrenzen. Während das Ziel ersterer primär in der reinen Beschreibung von äußerlich beobachtbarem Nutzerverhalten bestehe, fokussiere z.B. GOMS stärker auf interne kognitive Prozesse des Nutzers. Laut Dix et al. (2004) ist das primäre Ziel der Aufgabenanalyse die Erstellung eines konzeptuellen Modells, welches z.B. zur Dialogstrukturierung verwendet werden kann. Mit Hilfe der Aufgabenanalyse können darüber hinaus Systemschwächen ermittelt werden, die insbesondere anhand des beobachtbaren Nutzerverhaltens erkennbar sind. Zielorientierte kognitive Ansätze nutzen die erstellte Zielhierarchie zur weiteren Analyse von z.B. Komplexität oder Erlernbarkeit, d.h., deren Ziel geht von vornherein über die Erstellung des Modells hinaus und liegt in der Ermittlung von Schwachstellen des Systems, die vor allem durch kognitive Prozesse aufgedeckt werden können.

Empirische Usability-Tests

Beim empirischen *Usability-Testing* lösen potentielle Nutzer vorgegebene, realistische Aufgaben mit dem zu testenden System. Dabei wird ihr Verhalten von passiven Usability-Experten beobachtet und oft durch technische Hilfsmittel wie Videokameras, Mikrofone und Eingabeprotokolle aufgezeichnet. Häufig finden auch

[10] Card, Moran, Newell

Performanz- und Fehlermessungen statt, die zusammen mit den Beobachtungen Hinweise auf Usability-Probleme geben können. Dank der Vielseitigkeit der untersuchbaren Maße können Usability-Tests sowohl konkrete Problemstellen bzw. Verbesserungspotentiale in der Interaktion eines Systems aufzeigen, also auch zur Bewertung der allgemeinen Usability eines Systems oder zum Vergleich mehrerer ähnlicher Systeme herangezogen werden. Je nach Entwicklungsstand des Systems variiert die Zielausrichtung der Evaluation und damit auch deren Ablauf. So kann ein Usability-Test z.B. im Usability-Labor oder in einer realen Umgebung mit realen Störfaktoren durchgeführt werden. Auch die Wahl der Testaufgaben wird nach dem Ziel der Evaluation ausgerichtet. Die Probanden sollten die Zielgruppe so realistisch wie möglich repräsentieren. Für die Aufdeckung möglichst vieler Interaktionsprobleme wird empfohlen, Probanden zu wählen, die kaum bzw. keine Erfahrung mit dem zu testenden System haben (Sarodnick und Brau, 2006; Shneiderman und Plaisant, 2010). Usability-Tests werden häufig zur Usability-Bewertung und -Verbesserung eingesetzt, da sie auf Auswertungen tatsächlicher Nutzer beruhen und somit eine reale, nutzerzentrierte Usability-Analyse ermöglichen. Darin sieht auch Nielsen (1993) die Stärke dieser Methode und begründet so seine Bewertung des Usability-Testing als „fundamentalste Usability-Methode“.

Unterstützt werden kann ein Usability-Test durch das sogenannte *Eye-Tracking*. Dabei erfasst ein Gerät die genauen Blickbewegungen des Probanden zu einzelnen Elementen und Bildschirmpositionen des Systems. Dadurch wird die Analyse von Fixationen und Blickverläufen ermöglicht, die wiederum Rückschlüsse auf die Ausrichtung der Aufmerksamkeit des Probanden zulässt. Daraus können dann z.B. Gestaltungsanforderungen an das System abgeleitet werden.

Auch das *Thinking Aloud*[11] ist ein nützliches Hilfsmittel bei der Durchführung von Usability-Tests (Shneiderman und Plaisant, 2010). Beim Thinking Aloud wird der Proband aufgefordert, seine Gedanken auszusprechen und dadurch seine Interaktion mit dem zu testenden System zu kommentieren. Dabei sollte der Analyst nicht auf die Kommentare des Probanden eingehen, sondern diesen seine eigenen Gedankengänge und Lösungsansätze formulieren lassen. Die Kommentare des Probanden sollten schriftlich oder per Tonaufzeichnung festgehalten werden, da sie im Anschluss an den Test aufschlussreiche Hinweise zu der Passung zwischen Nutzer und System und damit auf Schwächen im Systemdesign geben können. Soll

[11] dt.: Lautes Denken

während des Usability-Tests die Ausführungszeit für die Testaufgaben gemessen werden, so wird diese durch die zusätzliche kognitive Belastung des Probanden durch das Thinking Aloud wahrscheinlich unpräziser. Um dennoch die Gedanken der Nutzer, wenn auch nicht so akkurat, und Zeitmessungen erfassen zu können, bietet sich das *Retrospective Thinking Aloud* an, bei dem die Probanden nach Vollendung des Tests zu bestimmten Interaktionsschritten des Tests befragt werden (Shneiderman und Plaisant, 2010, S. 161f).

Insgesamt ist das *Usability-Testing* als zeitintensive Methode zur Usability-Evaluation zu bewerten. Auch die Verfügbarkeit von Probanden aus der Zielgruppe kann sich als schwierig und kostspielig erweisen und ggfs. einen hohen organisatorischen Aufwand erfordern. Dem gegenüber steht die nutzerzentrierte Erfassung vielseitiger Usability-Maße, deren Auswertung maßgeblich zur Optimierung der Usability eines Systems beitragen kann.

Kognitive Architekturen

Kognitive Architekturen sind Software-Programme, die kognitiv plausibel beschreiben wie Wissen verarbeitet wird. Sie bestehen meistens aus einer Wissensbasis (Langzeitgedächtnis), einem Arbeitsgedächtnis (Kurzzeitgedächtnis), einer Wissensverarbeitung, einer Lernkomponente, einer Perzept- und einer Motorkomponente. Während die Wissensverarbeitung aktuell benötigtes Wissen aus der Wissensbasis in das Arbeitsgedächtnis lädt, manipuliert die Lernkomponente die Wissensbasis (Lüdtke, 2005). Ein Produktionensystem beinhaltet neben deklarativem Wissen, also reinem Faktenwissen, prozedurales Wissen. Dieses Wissen entsteht durch Erfahrungen und beschreibt Zusammenhänge, genauer gesagt „Handlungswissen“. Prozedurenwissen wird z.B. in Form von Produktionsregeln vorgehalten. Diese sind Wenn-Dann-Regeln, die Anweisungen enthalten, welche Aktionen in welchen Situationen oder für das Erreichen welcher Ziele auszuführen sind. Produktionsregeln steuern folglich die Handlungen eines Menschen. Deklaratives Wissen kann man demnach eher als „Wissen, was“ und prozedurales Wissen als „Wissen, wie“ bezeichnen.

Kognitive Architekturen können zur Usability-Evaluation eingesetzt werden. Dazu wird die Architektur, entsprechend des zu untersuchenden Systems, mit deklara-

tivem und prozeduralem Wissen angereichert. Anschließend kann die Architektur die Interaktion eines Nutzers mit dem System simulieren. So sind z.B. automatisierte Usability-Tests möglich, die keine realen Probanden erfordern. Von Vorteil ist diese Methode, wenn die Zielgruppe klein ist und aus Spezialisten besteht, deren Verfügbarkeit kostspielig und eingeschränkt ist. Außerdem lassen sich so deutlich mehr Tests in derselben oder kürzeren Evaluationszeit durchführen. Dem gegenüber steht allerdings ein sehr hoher Modellierungsaufwand, da die kognitive Architektur zunächst mit dem kompletten Wissen, das für die Interaktion mit dem zu testenden System benötigt wird, angereichert werden muss.

Bekannte Beispiele für kognitive Architekturen sind ACT-R[12], SOAR[13], EPIC[14] und CCT[15]. CCT (Bovair et al., 1990) nutzt beispielsweise die Komplexität und Anzahl der Produktionsregeln zur Berechnung von Lernzeiten und Komplexität der Interaktion. Anwendung findet CCT z.B. in NGOMSL[16] (Kieras, 1996), einer Methode der GOMS-Familie mit der Vorhersagen hinsichtlich der Lernzeit für bestimmte Interaktionen mit einem System gemacht werden können.

[12] Adaptive Control of Thought-Rational
[13] State, Operator and Result
[14] Executive Process-Interactive Control
[15] Cognitive Complexity Theory
[16] Natural GOMS Language

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Oldenburg, den 29.10.2010

Jutta Fortmann