Sonderforschungsbereich 378
Ressourcenadaptive kognitive Prozesse

KI-Labor am Lehrstuhl für Informatik IV

Leitung: Prof. Dr. W. Wahlster

REAL

Universität des Saarlandes
FB 14 Informatik IV
Postfach 151150
D-66041 Saarbrücken
Germany

Tel. 0681 / 302-2363

Memo Nr. 69

Ein Java-basiertes System zum Ressourcenmanagement in Anytime-Systemen

Frank Wittig

Januar 1999

Sonderforschungsbereich 378
Ressourcenadaptive
Kognitive Prozesse

ISSN 0944-7822

69

Ein Java-basiertes System zum Ressourcenmanagement in Anytime-Systemen

Frank Wittig

Fachbereich 14 - Informatik
Universität des Saarlandes
Saarbrücken

21. Januar 1999

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Es ist eine Alltagserfahrung, daß bei der Lösung gestellter Probleme eventuell vorhandene Ressourcenbeschränkungen zu beachten sind. Sei es bei einem Autokauf die zur Verfügung stehende Geldmenge oder seien es die zeitlichen Ressourcen eines Fahrernavigationssystems zur Bestimmung einer Wegbeschreibung. Ein Kommando wie "Biege links ab!" eines solchen Systems ist nutzlos, wenn die betreffende Kreuzung vom Benutzer bereits passiert wurde.

Im Forschungsgebiet der Künstlichen Intelligenz wurden die vorhandenen Ressourcen wie Zeit, Speicherkapazität und Prozessorleistung meist idealisierend als unbegrenzt angenommen. Erst in jüngerer Vergangenheit wurden auch die Ressourcenbeschränkungen als Gegenstand der Forschung betrachtet.

Der Schwerpunkt liegt hierbei bislang auf Untersuchungen im Hinblick auf zeitliche Restriktionen beim Lösen der gestellten Aufgaben. Insbesondere in der Robotik und bei der Entwicklung von Navigationssystemen wurden solche Aspekte betrachtet. Im Rahmen dieser Arbeit wird anhand des Beispiels der Entwicklung eines Objektlokalisationssystems diese Problematik thematisiert.

1.1 Einordnung

Diese Arbeit entstand im Rahmen des von der Deutschen Forschungsgemeinschaft (DFG) geförderten Sonderforschungsbereichs 378 "Ressourcen-adaptive kognitive Prozesse" (SFB, 1997; Wahlster & Tack, 1997) an der Universität des Saarlandes. Dort wird im Projekt "Interaktion von Objektlokalisation und Sprachproduktion" (REAL[1]) (REAL, 1996; Wahlster, Blocher, Baus, Stopp & Speiser, 1998) am Beispiel der Generierung sprachlicher Raumbeschreibungen in Dialogsituationen die Interaktion von ressourcenbeschränkter Objektlokalisation und inkrementeller Sprachproduktion untersucht. Abbildung 1.1 gibt einen Überblick über die Architektur von REAL.

Das System soll dem Benutzer *WO*-Fragen über die aktuelle visuelle Umgebung in verschiedenen Szenarien beantworten können. Anwendungsgebiete sind beispielsweise Navigationssysteme und (autonome) mobile Roboter. Es soll möglich sein, jederzeit die Berechnung der Antwort unterbrechen zu können, um eine Näherungslösung zu erhalten. Deshalb liegt dem System eine *Anytime-Architektur* zu-

[1] Ressourcen-adaptive Lokalisation

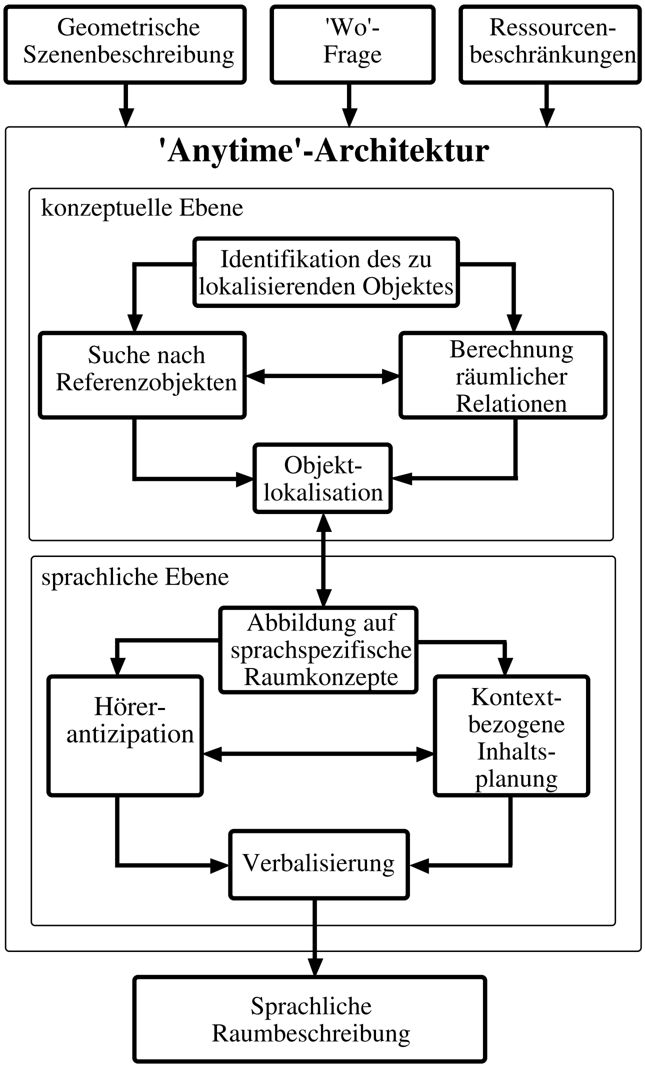

Abbildung 1.1: Architektur des Projekts REAL

grunde. Das bedeutet, je mehr Zeit der Berechnung zur Verfügung steht, desto besser ist die Qualität der Antwort, die bis zu diesem Zeitpunkt bestimmt wird. Innerhalb dieser Architektur werden zwei Ebenen unterschieden. Die *konzeptuelle Ebene* dient der sprachunabhängigen Bestimmung abstrakter Raumrelationen. Dazu wird in einem ersten Schritt das zu lokalisierende Objekt identifiziert, das heißt, die geometrische Entität, die das Objekt repräsentiert, wird auf ein Objekt der konzeptuellen Ebene abgebildet. Danach werden ein oder mehrere Referenzobjekte gesucht und räumliche Relationen bestimmt, die die Referenzobjekte und das zu lokalisierende Objekt in Beziehung setzen. Beide Aspekte zusammen ergeben eine (konzeptuelle) Objektlokalisation. Auf der *sprachlichen Ebene* wird diese auf sprachspezifische Raumkonzepte abgebildet. Die Planung der natürlichsprachlichen Äußerung erfolgt unter Berücksichtigung einer Hörerantizipation und durch eine kontextbezogene Inhaltsplanung. Den Abschluß bildet die Verbalisierung in einer bestimmten Zielsprache.

Das Thema dieser Arbeit ist die Konzeption und Implementation der umgebenden Anytime-Architektur.

1.2 Ziele

Das in der vorliegenden Arbeit vorgestellte System JAMES (Java Anytime Management & Editor System) stellt die Werkzeuge zur Entwicklung beliebiger Anytime-Systeme bereit. Es generiert Prozesse für die einzelnen Teilaufgaben dieser Systeme, verwaltet sie und weist ihnen gemäß einer gegebenen Strategie Ressourcen zu, die sie während ihrer nebenläufigen Berechnung nutzen können.

Die so konstruierten Systeme sind jederzeit unterbrechbar und stellen zu jedem Zeitpunkt eine (Näherungs-)Lösung der gestellten Aufgabe zur Verfügung. Die Qualität dieser (Zwischen-)Ergebnisse wächst mit den investierten Ressourcen.

Weiterhin werden in einer späteren Anwendungsphase Informationen aus bereits stattgefundenen Durchläufen der entwickelten Systeme im Rahmen der Ressourcenverwaltung genutzt. Damit kann die Performanz optimiert und ein *ressourcenadaptierendes Verhalten* induziert werden.

Das beschriebene System JAMES erlaubt es, den erzeugten Prozessen Verwaltungsinstanzen zuzuordnen, um *Subsysteme* zur Bearbeitung untergeordneter Teilaufgaben zu verwenden. Diese Verwaltungsinstanzen reichen ihre Informationen an übergeordnete durch, so daß alle Daten des Gesamtsystems auf allen Ebenen transparent sind.

Zur Anzeige und Manipulation dieser Informationen steht eine als *Werkbank* konzipierte Benutzerschnittstelle zur Verfügung, die insbesondere die Manipulation der Ressourcenverteilung der Prozesse ermöglicht.

1.3 Gliederung

Da das Thema dieser Arbeit im Forschungsgebiet der ressourcenbeschränkten Berechnung angesiedelt ist, werden in Kapitel 2 die zugehörigen Grundlagen und bisher erzielten Ergebnisse dargelegt. Die wesentlichen Arbeiten, die als Basis des vorgestellten Systems dienen, sind Inhalt von Kapitel 3. Anschließend folgt in Kapitel 4 die Beschreibung der Konzeption und Implementation des System zum Ressourcenmanagement in Anytime-Systemen. Seine Verwendung zur Konstruktion beliebiger Anytime-Systeme wird in Kapitel 5 am Beispiel des Objektlokalisationssystems BOLA dargestellt. Abschließend werden in Kapitel 6 die erzielten Ergebnisse zusammengefaßt und Erweiterungsmöglichkeiten aufgezeigt.

Kapitel 2

Ressourcenbeschränkte Berechnung

Bisher wurden in der Künstlichen Intelligenz die gegebenen Ressourcen wie Zeit, Speicherkapazität und Prozessorleistung meistens idealisierend als unbegrenzt angenommen. Erst in jüngerer Vergangenheit ist man dazu übergegangen auch eventuelle Beschränkungen zu berücksichtigen und als Gegenstand der Forschung zu betrachten. Insbesondere in Realzeitsystemen im Bereich der Robotik und der automatischen Wegeplanung wurden dadurch auftretende Problemstellungen untersucht.

Dieses Kapitel soll eine Einführung in das Forschungsgebiet der ressourcenbeschränkten Berechnung geben. Dazu wird zuerst der Ressourcenbegriff untersucht. Darauf aufbauend werden unterschiedliche Formen der Ressourcensensitivität erläutert, gefolgt vom Begriff der beschränkten Optimalität. Den Abschluß bildet ein Überblick über verschiedene praktische Ansätze, die bisher entwickelt wurden.

2.1 Ressourcenbegriff

Da der Begriff der *Ressource* in dieser Arbeit ein wichtige Rolle spielt, wird zu Beginn dieses Kapitels seine Bedeutung und die Art seiner Verwendung für den weiteren Verlauf festgelegt. Dies geschieht in diesem Abschnitt anhand des Artikels von Jameson und Buchholz (Jameson & Buchholz, 1998), der den Begriff aus der Sicht des Kognitionswissenschaftlers untersucht.

Danach versteht man unter einer Ressource im allgemeinen ein beliebiges Hilfsmittel, das zum Erreichen eines Ziels verwendet werden kann, wobei der Ausdruck des Hilfsmittels im weitesten Sinne aufgefaßt wird. Als einfaches Beispiel sei hier die Fähigkeit des Lesens und Schreibens in einer Prüfungssituation angeführt.

Nach Jameson und Buchholz besitzen verschiedene *Ressourcentypen* unterschiedliche Eigenschaften. Einige werden im Rahmen ihrer Verwendung verbraucht, wie z.B. Strom, andere hingegen, wie Wissen, das innerhalb eines Computersystems in einer Datenbank abgelegt sein kann, können beliebig oft angewendet werden. Eine weitere solche Eigenschaft ist die Aufteilbarkeit in kleinere Einheiten. Man kann die Ressource Zeit beliebig fein portionieren, wohingegen ein einem System zur Verfügung stehender Prozeß nur als Ganzes sinnvoll bei der Berechnung benutzt werden kann.

Nach diesen informalen Betrachtungen wird nun die abstrakte Formulierung des Ressourcenkonzepts von Jameson und Buchholz gegeben. Dazu wird die folgende Definition eines Agenten nach (Russell & Norvig, 1995) eingeführt:

Definition 2.1.1 (Agent) *Ein* Agent *ist eine Entität, die ihre Umgebung durch Sensoren wahrnimmt und in ihr durch Effektoren agiert.*

Der Mensch als Beispiel eines Agenten besitzt unter anderem Augen und Ohren, um seine Umgebung wahrzunehmen, Hände, Mund etc. stellen Effektoren dar. Roboter nutzen zu diesen Zwecken Kameras und Infrarotsensoren bzw. mehrere Motore und künstliche Gliedmaßen.

Die zu betrachtende Situation zur Festlegung des Ressourcenbegriffs nach Jameson und Buchholz, die in Abbildung 2.1 dargestellt ist, besteht aus einem Agenten A, der innerhalb eines gewissen Zeitraums eine oder mehrere Aufgaben $A_1, \ldots, A_n$ bearbeiten und die Ergebnisse $E_1, \ldots, E_n$ bestimmen soll. Dazu stehen ihm die Ressourcen $R_1, \ldots, R_m$ zur Verfügung. Wird R_i zur Lösung der Aufgabe A_j genutzt, so ist die *Verwendung* V_{ij} ein Maß für die benutzte Menge dieser Ressource.

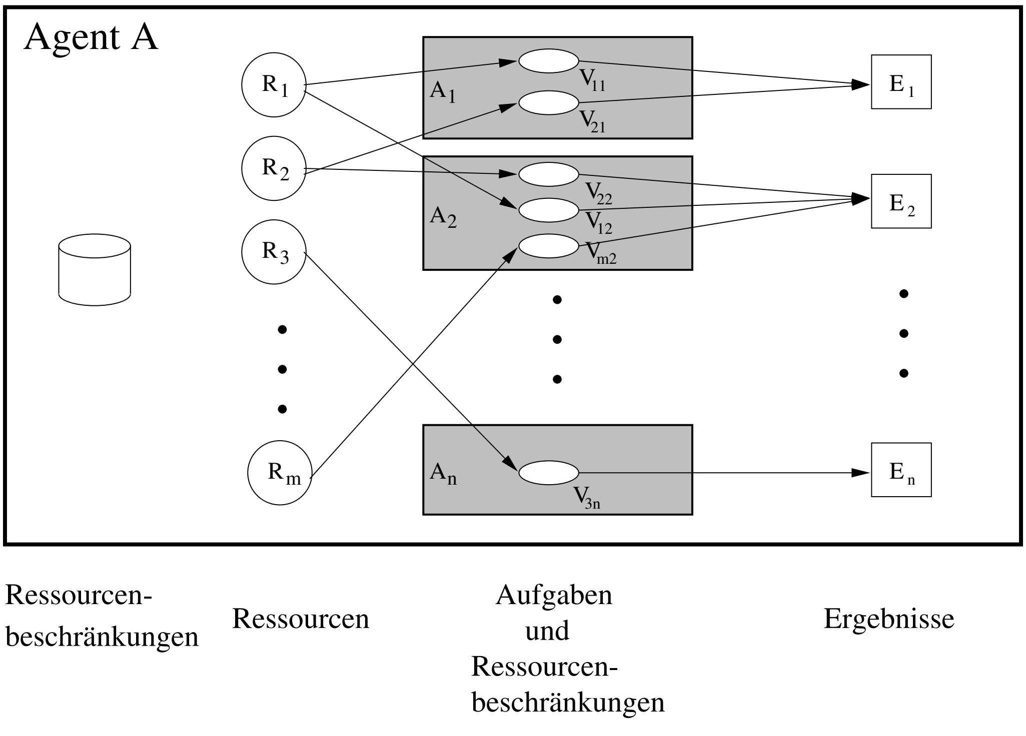

Abbildung 2.1: Ressourcen eines Agenten (nach [Jameson & Buchholz 98])

Diese Verwendungen unterliegen bestimmten *Ressourcenbeschränkungen*. Die Zeit etwa darf nicht die dem Agenten vorgegebene Restriktion für die Lösung der Aufgaben überschreiten. Ein Beispiel für eine komplexere Ressourcenbeschränkung ist der Zeitverlust beim Wechsel zwischen der Bearbeitung verschiedener Aufgaben. Handelt es sich hierbei um Prozeßscheduling in Betriebssystemen, dann spricht man von einem sogenannten *Kontextwechsel* (s. (Tanenbaum, 1992)). Auch in dem in dieser Arbeit vorgestellten System zum Ressourcenmanagement in Anytime-Systemen werden Kontextwechsel zwischen Prozessen, die die Bearbeitung der einzelnen Teilaufgaben als Aufgabe besitzen, durchgeführt.

Ein weiterer Aspekt in diesem Zusammenhang sind die *Kosten* einer Ressource. Sie teilen dem Agenten mit, in wie weit es für ihn (un-)günstig ist, die betreffende Ressource zu nutzen, unabhängig von den vorliegenden Ressourcenbeschränkungen. So sind im allgemeinen die Kosten für einen Plattenzugriff im Vergleich zu einem Zugriff auf den Hauptspeicher, bedingt durch die längere Zugriffszeit, wesentlich höher.

Aus den bisherigen Ausführungen ergeben sich nach den Autoren des Artikel Fragestellungen, deren Untersuchung zentraler Inhalt dieses und der weiteren Kapitel dieser Arbeit sind (dies bezieht sich hauptsächlich auf die beiden ersten Fragen):

- Wie können die verfügbaren Ressourcen effektiv an die zu erledigenden Aufgaben zugeteilt werden?
- Wie können auch diejenigen Ressourcen effektiv eingesetzt werden, deren Verfügbarkeit nicht von vornherein bekannt ist?
- Wie können unterschiedliche Ressourcen zusammen ausgenützt werden?

In den weiteren Kapiteln der vorliegenden Arbeit werden zwei Typen von Ressourcen genauer betrachtet, die *Zeit* und die Anzahl der dem System zur Verfügung stehenden *Prozesse*. Wenn nicht anders erwähnt, so wird dort der Begriff der Ressource mit dem Begriff der Zeit gleichgesetzt.

2.2 Ressourcensensitivität

In diesem Abschnitt werden verschiedene theoretische Ansätze einer möglichen Vorgehensweise bei der Beantwortung der von Jameson und Buchholz formulierten Fragestellungen erläutert.

Werden vom Agenten die Ressourcenbeschränkungen bei der Lösung der ihm gestellten Aufgaben in irgendeiner Form berücksichtigt, so spricht man von *ressourcensensitivem Verhalten* (Wahlster & Tack, 1997). Je nach Art der Berücksichtigung unterscheidet man die drei Fälle:

- **ressourcenadaptiertes Verhalten:** Der Agent ist auf bekannte und feste Ressourcenbeschränkungen hin optimiert. Sein Verhalten bezüglich der Verwendung von Ressourcen ist vor dem Beginn seiner Berechnung festgelegt und kann während deren Verlauf nicht eventuell auftretenden Umgebungsveränderungen angepaßt werden. Ein Beispiel hierfür wäre ein mobiler Roboter, der eine feste Zeitspanne zugewiesen bekommt, um sich an einen bestimmten Ort zu begeben und dort eine festgelegte Tätigkeit durchzuführen. Dabei wären die Zeiten, die er zur Bewältigung seines Weges, sowie der Ausführung seiner Tätigkeit zur Verfügung hat, zum Zeitpunkt seiner Konstruktion festgelegt worden, beispielsweise zwei Minuten für die Bewegung und eine Minute zum Lösen der Aufgabe.

- **ressourcenadaptives Verhalten:** Bei der Bearbeitung des gestellten Problems verfolgt der Agent eine feste Verarbeitungsstrategie, die aber durch die vorliegenden Ressourcenbeschränkungen beeinflußt wird. So entscheidet er zur Laufzeit mittels Metawissen wieviele Ressourcen welche Teilaspekte der Berechnung zugewiesen bekommen. Der Unterschied zur Vorgehensweise im obigen Beispiel wäre, daß die verfügbare Gesamtzeit variabel wäre. Eine Kontrolleinheit des Roboters würde zur Laufzeit darüber entscheiden, welche Zeitspannen den beiden Arbeitsphasen zugewiesen würden.

- **ressourcenadaptierendes Verhalten:** Ähnlich wie im vorigen Fall werden zur Laufzeit durch die Verwendung von Metawissen Entscheidungen über das Vorgehen bei der Berechnung getroffen. Hier bezieht sich dies auch auf die Vorgehensweise. Der Agent hat die Wahl zwischen verschiedenen Strategien, die je nach Ressourcenaufwand unterschiedlich gute Ergebnisse liefern. Geht der Agent davon aus, daß er viele Ressourcen zur Verfügung hat, wird er eine Strategie wählen, die zwar eine größere Menge benötigt, aber ein besseres Resultat bestimmt. Der andere Fall besteht in der Auswahl einer Strategie mit geringerem Aufwand bei weniger verfügbaren Ressourcen und einem schlechterem erzielten Ergebnis. In diesem Szenario hätte der mobile Roboter unterschiedlich "gute" Algorithmen, um seinen Weg zu bestimmen. Anhand der vorgegebenen Zeitspanne würde seine Kontrolleinheit zur Laufzeit entscheiden, welchen Algorithmus er benutzt und wieviele Ressourcen dazu verwendet werden.

Abbildung 2.2 zeigt eine Gegenüberstellung dieser Arten von ressourcensensitivem Verhalten.

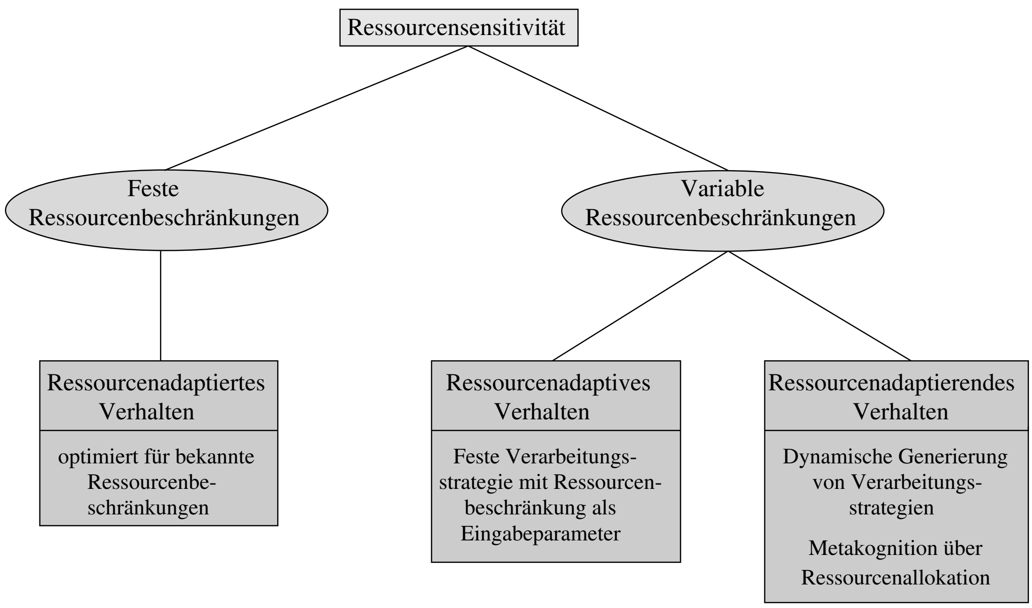

Abbildung 2.2: Drei Typen der Ressourcensensitivität (nach einer Folie von W. Wahlster)

Die drei Verhaltensweisen stellen theoretische Ansätze zur Lösung der im vorhergehenden Abschnitt aufgeführten Problemstellungen dar. Vor der Konstruktion eines Agenten muß man entscheiden, welches Verhalten für ihn adäquat ist. Dabei müssen beispielsweise solche Aspekte beachtet werden, wie das Vorhandensein unterschiedlicher Lösungsstrategien mit verschiedenem Ressourcenverbrauch und damit einhergehenden unterschiedlichen Qualitäten der Resultate. Ist dies nicht der Fall, so scheidet die Möglichkeit der Modellierung von ressourcenadaptierendem Verhalten von vorne herein aus.

Ein weiterer Forschungsansatz, der sich mit den Fragestellungen aus Abschnitt 2.1 beschäftigt wird im folgenden behandelt.

2.3 Beschränkte Optimalität

2.3.1 Begriffsbildung

Es ist wünschenswert, daß ein Agent in einem bestimmten Zustand die bestmögliche Aktion ausführt, das heißt die Aktion, die die Qualität des zu ermittelnden Resultats maximiert. In diesem Fall spricht man von einem *perfekt rationalen* Agenten. I. J. Good (Good, 1971) nennt diese Art des Handelns *Typ I Rationalität*. Seine *Typ II Rationalität* berücksichtigt zusätzlich, daß der Agent "nachdenken" muß bevor er handelt. Ein mehr technischer Ansatz hierzu stellt der Begriff der *operationalen Rationalität* von Shlomo Zilberstein (Zilberstein, 1993) dar. Er trennt zwischen Lösungsverfahren und Kontrollinstanz. Der Agent optimiert nur auf Ebene dieser Kontrollinstanz die Zuteilung der Ressourcen an die Lösungsverfahren. Diese selbst sind kein Gegenstand eines "Nachdenkens" des Agentens.

Zur Lösung der in Abschnitt 2.1 aufgestellten Fragen, muß ein Agent *Metawissen* verwenden, das heißt er benutzt Wissen über die ihm zur Verfügung stehenden Ressourcen und seiner Lösungsstrategien, um zu entscheiden, wie er bei der Berechnung der gestellten Aufgaben vorgeht. Die Eigenschaft, das bestmögliche Ergebnis durch die bestmögliche Verwendung der Ressourcen zu bestimmen, wird als *beschränkte Optimalität* bezeichnet (Russell & Subramanian, 1995).

Abbildung 2.3 verdeutlicht diese Situation anhand des Beispiels der Ressource "Zeit". Der optimale Fall wäre die Bestimmung des idealen Ergebnisses ohne jeglichen Zeitaufwand. Dies kann von einem realen Agenten nicht geleistet werden. Der klassische Ansatz liefert das optimale Ergebnis nach einer gewissen Zeitspanne. Vorher liegen keinerlei Informationen vor. Der im folgenden betrachtete Ansatz besteht in einem Tradeoff von Zeit und Ergebnisqualität. Je mehr Zeit investiert wird, desto qualitativ besser entwickelt sich das Ergebnis. Wird nur die Qualität des Ergebnisses ohne Berücksichtigung der zu seiner Berechnung aufgewendeten Zeit betrachtet, so liefert dies nicht unbedingt eine Aussage über die beste Quantität der zu verwendenden Ressourcen. Denn es besteht die Möglichkeit, daß ein zwar mit sehr guter Qualität bestimmtes Ergebnis nach einer gewissen Zeitspanne keinen Nutzen mehr für den Agenten besitzt. Als Beispiel hierfür kann ein Fahrernavigationssystem dienen, bei dem eine Anweisung wie "Biege links ab!" nutzlos ist, wenn die betreffende Kreuzung bereits passiert wurde. Dies bedeutet, daß man die Kosten der Ressourcen (s. Abschnitt 2.1) mit berücksichtigen muß. Das Maximum der Kombination beider

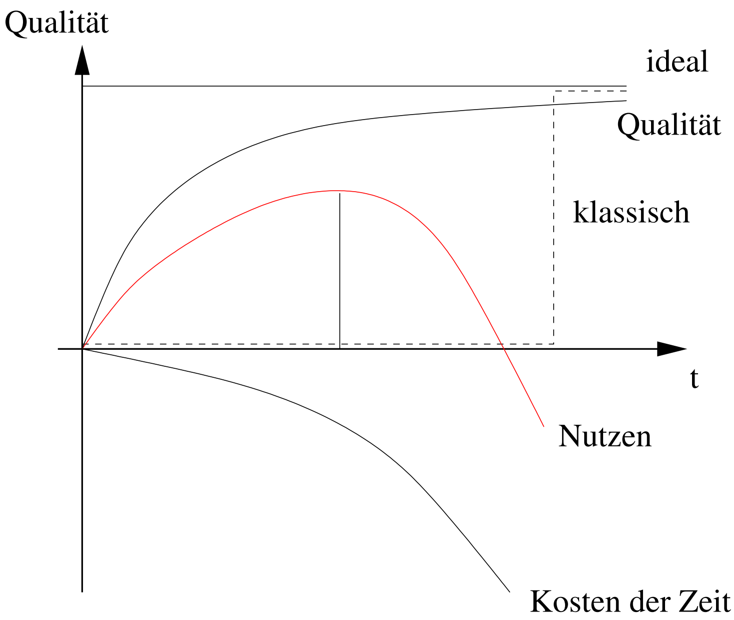

Abbildung 2.3: Einfluß der Zeit auf den Nutzen

Funktionen ergibt den optimalen Ressourcenaufwand.

2.3.2 Beschränkungen der Ressource “Zeit”

In diesem Zusammenhang werden im Spezialfall der Ressource “Zeit” drei verschiedene Arten von Ressourcenbeschränkungen unterschieden:

- **Feste Zeitschranke:** Nach Ablauf der Schranke sind die erzielten Ergebnisse nutzlos.
- **Feste Kostenfunktion:** Ziel ist die Minimierung der Kosten bei möglichst großem Nutzen.
- **Stochastische Zeitschranke:** Es besteht eine Unsicherheit bezüglich der Zeitschranke, ab der das erzielte Ergebnis nutzlos ist.

Welche dieser drei Möglichkeiten vorliegt, ist von der betrachteten Anwendung abhängig.

2.3.3 Meta-Kontrolle durch “Value of Computation”

Russell und Wefald (Russell & Wefald, 1991) zeigen mit ihrem Ansatz “Value of Computation” eine Art der Metakontrolle auf, die es einem Agenten ermöglicht, die Eigenschaft der beschränkten Optimalität zu erfüllen.

Zur Erläuterung diese Ansatzes werden folgende Bezeichnungen eingeführt: Der Agent besitzt eine *Nutzfunktion* $U : \Omega \rightarrow \mathbb{R}$, die die Zustände Ω, in denen sich der Agent befinden kann, auf eine reelle Zahl abbildet, die angibt, wie wertvoll der betrachtete Zustand ist. Der Agent kann entweder Berechnungen auf der Metaebene oder eigentliche Aktionen vornehmen. Die Schreibweise $[A_i, \omega]$ symbolisiert den

Folgezustand zu Zustand ω, wenn in ihm die Metaberechnung oder die Aktion A_i ausgeführt wird. Der Folgezustand zum aktuellen Zustand wird abkürzend mit $[A_i]$ bezeichnet.

Laut Russell und Wefald entscheidet der Agent zu jedem Zeitpunkt, ob er weitere Metaberechnungen $S_1, \ldots, S_k$ vornimmt, oder ob er die beste bislang bestimmte *Defaultaktion* α ausführt. Arbeitet er weiter auf der Metaebene, so bezeichnet α_T diejenige Aktion, die nach der Metaberechnungsfolge T ausgeführt wird. Die optimale Aktion, die der Agent auswählen sollte, ist diejenige, die den erwarteten Nutzen U maximiert.

Im folgenden wird erläutert, wie die beiden Autoren bei diesem Ansatz eine erwartete optimale Aktion bestimmen. Der *Nettowert* V einer Metaberechnung S_j stellt den Zuwachs an Nutzen im Vergleich zur Ausführung der Defaultaktion α dar:

$$V(S_j) = U([S_j]) - U([\alpha]). \tag{2.1}$$

Der Agent führt die Aktion mit dem größten Nettowert aus. Zu seiner Bestimmung müssen die folgenden Fälle unterschieden werden.

Wird nach der Metaberechnung S_j die Aktion α_{S_j} ausgeführt, dann gilt:

$$U([S_j]) = U([\alpha_{S_j}, [S_j]]) \tag{2.2}$$

und somit

$$V(S_j) = U([\alpha_{S_j}, [S_j]]) - U([\alpha]). \tag{2.3}$$

Wird nach der Metaberechnung S_j eine weitere durchgeführt, das heißt keine Aktion, dann will man mit ihrem Nutzen ausdrücken, in wie weit sich die zukünftigen Metaberechnungen auf die letztendliche Wahl der auszuführenden Aktion auswirken. Somit repräsentiert dieser Wert den erwarteten Nutzen dieser Aktion. Dieser ergibt sich aus der Summation über die Nutzen aller möglichen Metaberechnungssequenzen T unter Berücksichtigung ihrer Wahrscheinlichkeit:

$$U([S_j]) = \sum_T P(T) U([\alpha_T, [S_j.T]]), \tag{2.4}$$

wobei $P(T)$ die Wahrscheinlichkeit ist, daß der Agent die Metaberechnungssequenz T ausführt und $S_j.T$ bedeutet, daß T nach S_j durchgeführt wird. Die beste Wahl ist die mit dem größten erwarteten Nutzen.

Die Nutzfunktion U umfaßt alle Aspekte der Umgebung des Agenten, so auch die Zeit. Russell und Wefald schlagen in einem weiteren Schritt die Aufspaltung dieser Funktion in einen zeitunabhängigen Anteil U_I und die Kosten der Zeit TC vor. Dann ergibt sich für den Nutzen der Aktion bzw. Metaberechnung A_i nach der Metaberechnungssequenz T:

$$U([A_i, [T]]) = U_I([A_i]) - TC(|T|). \tag{2.5}$$

Somit folgt für den Nettowert:

$$V([T]) = \Delta(T) - TC(|T|) \tag{2.6}$$

mit

$$\Delta(T) = U([\alpha_T]) - U([\alpha]). \tag{2.7}$$

Man trennt auf diese Weise die Zeit von der Nutzfunktion ab und kann diese im Rahmen der Metaentscheidungsfindung gesondert repräsentieren und behandeln. Dies ist vorteilhaft, da die Zeit in den meisten Systemen die entscheidende Ressource ist.

Wie dieses Vorgehen in praktische Ansätze einfließt, wird im folgenden Abschnitt untersucht.

2.4 Praktische Forschungsansätze

Anschließend werden einige praxisorientierte Ansätze zur Behandlung der bisher diskutierten Probleme erläutert, um einen Überblick über die bislang erreichten Ergebnisse im Forschungsfeld der ressourcenbeschränkten Berechnung zu erhalten.

2.4.1 Anytime-Algorithmen (Horwitz, Dean und Boddy)

Eine erster Ansatz, der sich mit diesem Thema beschäftigte, ist der Begriff der *flexiblen Berechnungen* (Horvitz, 1987). Darunter versteht man die Verwendung flexibler Strategien zur Problemlösung in Abhängigkeit von den vorhandenen Ressourcen. Es wird immer die unter den gegebenen Voraussetzungen vielversprechendste angewendet. Dies entspricht der Maximierung des Nutzens bei Russell und Wefald.

Am Beispiel eines Agenten zum automatischen Schließen mit Unsicherheiten im Rahmen eines medizinischen Diagnosesystems (PROTOS) wird eine Vorgehensweise aufgezeigt, die durch Bestimmung von Teilergebnissen unterschiedlicher Qualitäten, approximative Lösungen bereitstellt, die von der zur Verfügung stehenden Zeit abhängig sind. Unter Betrachtung des Kosten-Nutzen-Verhältnisses der zur Berechnung benötigten Zeit werden Ergebnisse produziert, deren Qualitäten sich monoton steigend bezüglich der aufgewendeten Zeit verhalten. Horvitz spricht hier von einer Verfeinerung der Lösung. Im einzelnen stellt er folgende Anforderungen an sie:

- **Kontinuität des Nutzens und der Qualität:** Der Nutzen sowie die Qualität der Ergebnisse, die im Verlauf der Verfeinerung berechnet werden, stellen stetige Funktionen dar.

- **Monotonie der Qualität:** Die Qualitätswerte müssen sich wie oben erwähnt monoton steigend bezüglich des Ressourcenaufwands verhalten.

- **Konvergenz der Qualität:** Die Qualität konvergiert gegen die des optimalen Ergebnisses, das berechnet würde, wenn genügend Ressourcen zur Verfügung stehen würden.

- **Dominanz des Nutzens:** Die entsprechenden Algorithmen müssen Zeitintervalle besitzen, in denen sich die Nutzfunktion monoton steigend verhält, je mehr Ressourcen zur Verfügung stehen.

Darauf aufbauend prägen Dean und Boddy in (Dean & Boddy, 1988; Boddy, 1991a, 1991b) den Begriff des *Anytime-Algorithmus*. Er stellt eine Formalisierung der von Horvitz eingeführten flexiblen Berechnungen dar.

Die Autoren erläutern dieses Konzept des Anytime-Algorithmus am Beispiel eines stationären Roboters, der Gegenstände auf einem Fließband erkennen muß und sie entsprechend ihres Typs an andere Bänder weiterleiten soll. Die Gegenstände erscheinen in unregelmäßigen Abständen in seinem Blickfeld. Ihm bleibt also genau die Zeitspanne zur Erkennung und Zuordnung der Objekte, die vergeht, bis sie das Ende des Fließbandes erreicht haben. Außerdem variiert die zur Bearbeitung eines Objekts zur Verfügung stehende Zeit, je nachdem wieviele Gegenstände sich im Augenblick im Bereich seiner visuellen Wahrnehmung befinden.

Der Roboter hat also mehrere Aufgaben, die er unter Berücksichtigung verschiedener Anforderungen, in einer bestimmten Zeitspanne bearbeiten muß. Er muß entscheiden, wieviel Zeit er dafür im einzelnen aufwendet. Dies geschieht nach den von Russell und Wefald (s. Abschnitt 2.3.3) beschriebenen Vorgehensweisen. Zur Erledigung der einzelnen Teilaufgaben (Erkennung, Zuordnung) werden Anytime-Algorithmen verwendet, die die folgenden Eigenschaften besitzen:

- Anytime-Algorithmen können jederzeit mit minimalem administrativem Aufwand unterbrochen und wieder fortgesetzt werden.
- Sie stellen zu jedem Zeitpunkt ein (approximatives) Ergebnis bereit, dessen Qualität als Funktion der investierten Arbeitszeit monoton ansteigt.
- Jedem Anytime-Algorithmus sind Informationen über seinen Tradeoff bezüglich aufgewendeter Zeit und erzielter Qualität in monoton steigenden *Performanzprofilen* zugeordnet. Diese können genutzt werden, um eine (optimale) Zeitverteilung an mehrere Anytime-Algorithmen vorzunehmen.

Dieser Abschnitt beschäftigte sich mit den Grundlagen der ressourcenbeschränkten Berechnung im Rahmen einzelner Algorithmen. Anschließend werden diesbezügliche Konzepte zur Erstellung von Systemen beschrieben.

2.4.2 Compilierung von Anytime-Algorithmen (Zilberstein)

Weiterführend löst Zilberstein (Zilberstein, 1993) die Komposition eines Systems aus mehreren Anytime-Algorithmen mittels einer *Offline-Compilierung*, in deren Verlauf, unter Verwendung ähnlicher Techniken wie bei Russell und Wefald, eine Verteilung der Ressourcen auf die einzelnen Komponenten vorgenommen wird.

Dazu erweitert er den Ansatz von Dean und Boddy um den Begriff des *bedingten Performanzprofils*, welches benötigt wird, um das Verhalten der Ergebnisqualität unter Berücksichtigung einer bestimmten Eingabequalität zu beschreiben. Dies ist notwendig, da bei der Compilierung verschiedene Anytime-Algorithmen sequentiell ausgeführt werden, insbesondere kann einer das Ergebnis eines anderen als Eingabe benutzen. Zilberstein compiliert einzelne Anytime-Komponenten zusammen mit zusätzlichem Kontrollcode und Informationen über die Ressourcenverteilung zu einem ausführbaren Gesamtsystem. Es werden zwar zur Laufzeit durch *Monitoring* noch Anpassungen der Ressourcenzuteilung vorgenommen, aber die zur Verfügung stehende Gesamtzeit muß vor der Compilierung, also insbesondere vor dem Start des Systems bekannt sein. Eine flexible Reaktion auf Änderungen der Ressourcenbeschränkungen ist somit nicht möglich.

Wegen der Verwendung von Anytime-Algorithmen im Rahmen des in der vorliegenden Arbeit entwickelten Systems werden die Konzepte der beiden letzten Abschnitte im Kapitel 3 ausführlich erläutert.

2.4.3 Design-to-Time-Scheduling (Garvey und Lesser)

Mit dem *Design-to-Time-Scheduling* (Garvey & Lesser, 1993, 1994; Garvey, Humphrey & Lesser, 1993) stellen Garvey und Lesser eine Metakontrolle vor, die nicht auf der Verwendung von Anytime-Algorithmen aufbaut. Die Autoren gehen von der Voraussetzung aus, daß es zur Lösung des Gesamtproblems verschiedene Verfahren zur Bearbeitung von Teilaspekten gibt, die unterschiedlichen Ressourcenverbrauch besitzen und die abhängig von den zur Verfügung stehenden Ressourcen unterschiedliche Ergebnisqualitäten liefern können. Die Aufgabe lautet dann, diejenigen Algorithmen auszuwählen, die unter Beachtung der vorgegebenen Schranken das bestmögliche Ergebnis erzielen. Dabei werden eventuell vorhandene Abarbeitungsreihenfolgen eingehalten. Zur Darstellung der Struktur des Systems werden Bäume benutzt.

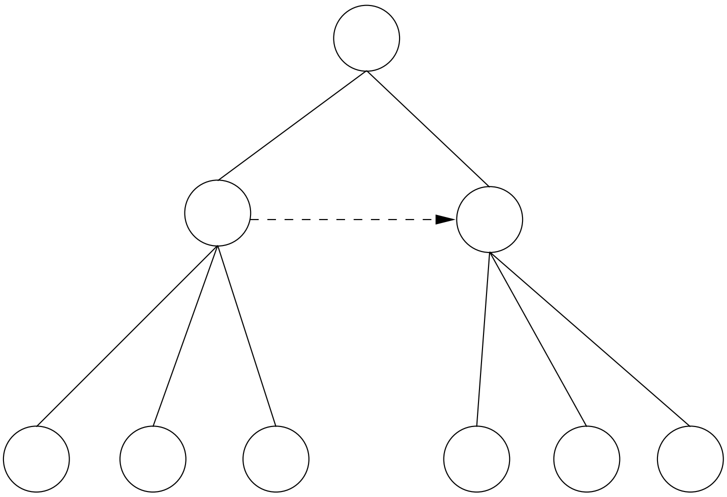

Abbildung 2.4: Baumstruktur der Berechnungen beim Design-to-time-Scheduling

Abbildung 2.4 zeigt ein Beispiel hierfür. Die Blätter repräsentieren die eigentlichen Berechnungen. Innere Knoten des Baumes symbolisieren Teilaspekte des Gesamtproblems als Kombination der Berechnungen ihrer Nachfolgeknoten. Ein gestrichelter Pfeil bedeutet, daß die Berechnung von deren Knoten er ausgeht, erledigt sein muß, bevor die Berechnung, auf deren Knoten der Pfeil gerichtet ist, ausgeführt werden kann. Voneinander unabhängige Probleme werden in getrennten (Teil-)Bäumen modelliert.

Eine Weiterentwicklung besteht in der expliziten Anpassung des Konzepts an die Verwendung von Anytime-Algorithmen (Garvey & Lesser, 1996). Dabei werden deren besonderen Eigenschaften genutzt. So können aus einem Anytime-Algorithmus mehrere Alternativen zur Berechnung generiert werden, indem er mit unterschiedlichen Laufzeiten benutzt wird. Um möglichst optimale Ergebnisse zu erzielen, sind sehr gute Heuristiken zur Bestimmung dieser Laufzeiten notwendig.

Ebenso wie bei Zilberstein wird Monitoring angewendet, um unvorhergesehene Abweichungen des realen Laufs von den Vorgaben festzustellen und auszugleichen. Solche Abweichungen können beispielsweise durch Ungenauigkeiten bei der Zeitmessung auftreten.

2.4.4 Controlled Concurrency (Wang)

Ein weiterer Vorschlag zur Vorgehensweise bei ressourcenbeschränkter Berechnung ist der der *Controlled Concurrency* (Wang, 1996). Am Beispiel des Interferenzsystems NARS (Non-Axiomatic Reasoning System) wird ein weiterer Ansatz zur internen Strukturierung eines unter Ressourcenbeschränkungen arbeitenden Systems aufgezeigt.

Die Bearbeitung anfallender Aufgaben, wie das Einbringen von neuem Wissen und den daraus resultierenden Schlußfolgerungen oder Anfragen, die an das System gestellt werden, wird in einzelne Einheiten (*Tasks*) gegliedert. Diese können dann nebenläufig abgehandelt werden. Außerdem kann jede von ihnen für sich parametrisiert werden, insbesondere können die Anzahl der Interferenzschritte und der zu untersuchenden Wissenseinheiten festgelegt werden.

Durch die Gesamtheit aller Tasks und der im Rahmen ihrer nebenläufigen Ausführung sukzessive ermittelten Ergebnisse erhält man ein System, das sich wie ein Anytime-Algorithmus verhält: Es ist jederzeit unterbrechbar und es besitzt eine monoton steigende Ergebnisqualität.

Um unterschiedlichen Eigenschaften der verschiedenen Aufgaben Rechnung zu tragen, werden jedem Task zwei Werte zugeordnet. Die *Dringlichkeit* charakterisiert die Priorisierung des Tasks bezüglich anderer. Betrachtet man zwei Tasks A und B mit Dringlichkeit d_A und d_B, so werden die vorhandenen Ressourcen im Verhältnis d_A/d_B unter ihnen verteilt. Um nun noch eine längerfristige Modellierung des Verhaltens zu ermöglichen, wird der zweite Wert, der *Verfall*, genutzt, der einen Alterungsprozeß des Tasks bewirkt. Sein Wert gibt an, wie sich die Dringlichkeit des Task nach einer gewissen Zeitspanne verhält.

Zur Bearbeitung der Tasks können auch in diesem Ansatz Anytime-Algorithmen verwendet werden, wobei die Performanzprofile Hinweise auf die Größe der Dringlichkeit und des Verfalls liefern. Ein Algorithmus dessen Ergebnisqualität sehr schnell auf ein hohes Niveau ansteigt und dann nur noch geringe Verbesserungen über große Zeiträume bietet, wird eher mit einer relativ großen Dringlichkeit und einem relativ großen Verfall behandelt, denn nach einer bestimmten Zeit lohnt sich ein starker Ressourcenaufwand für ihn nicht mehr. Dann sollten die verbleibenden Ressourcen auf andere Tasks verteilt werden.

2.4.5 Vergleich

In Tabelle 2.1 werden die Eigenschaften der verschiedenen Forschungsansätze zur ressourcenbeschränkten Berechnung gegenübergestellt. Dabei wurde auf die Aufführung der beiden Ansätze "flexible Berechnungen" und "Anytime-Algorithmen" verzichtet, da diese sich auf die Entwicklung für sich arbeitender Algorithmen beziehen und im Zusammenhang mit der Konstruktion von komplexen Systemen als Grundlage dienen können.

Die drei verbleibenden Konzepte "Compilierung von Anytime-Algorithmen", "Design-to-time-Scheduling" und "Controlled Concurrency" werden anhand folgender Charakteristika zusammen mit einer wünschenswerten Kombination verglichen:

- **Möglichkeit der Verwendung von Standard-Algorithmen[1]:** Standard-

[1]Unter *Standard-Algorithmen* werden in diesem Zusammenhang solche Algorithmen verstanden,

Algorithmen können zur Konstruktion des Gesamtsystems genutzt werden. Dies hat den Vorteil, bekannte Verfahren anwenden zu können. Es ist nicht notwendig neue Anytime-Algorithmen zu entwickeln.

- **Möglichkeit der Verwendung von Anytime-Algorithmen:** Es besteht die Möglichkeit im Rahmen der Entwicklung des Gesamtsystems, Anytime-Algorithmen zu benutzen. Dies führt meist zu einer feineren Auflösung der Qualität der Ergebnisse. Werden nur Standard-Algorithmen verwendet, so entwickelt sich der Graph des Performanzprofils des Systems in größeren Stufen (s. hierzu Abschnitt 3.1.4).

- **Unterstützung unterschiedlicher Lösungsstrategien:** Existieren für die Lösung eines Problems unterschiedliche Lösungsstrategien, so können diese auch genutzt werden. Dies wirkt sich dann positiv aus, wenn diese Strategien unter unterschiedlichen Voraussetzungen unterschiedlich gute Ergebnisse produzieren. Durch eine geeignete Metakontrolle können so abhängig von den zur Verfügung stehenden Ressourcen die am besten geeigneten Methoden ausgewählt werden. Weiterhin führt die Verwendung unterschiedlicher Lösungsstrategien zu einer feineren Auflösung der Qualitätsfunktion.

- **Bearbeitung abhängiger Teilaufgaben:** Das Konzept sieht eine Unterstützung der Bearbeitung abhängiger Teilaufgaben vor. Damit wird eine flexible Entwicklung der Systemstruktur gewährleistet. Insbesondere wird ein modularer Systemaufbau ermöglicht.

- **Unterbrechbarkeit des konstruierten Systems:** Das Gesamtsystem ist jederzeit unterbrechbar. Die Gesamtzeit muß nicht vor dem Start bekannt sein. Eine solche Festlegung vor der Bearbeitung der Aufgabe stellt eine Einschränkung der Anwendbarkeit des Konzepts in vielen Einsatzbereichen dar.

- **Monitoring:** Zur Laufzeit wird Monitoring zur Anpassung eventuell auftretender Abweichungen von den vorher bestimmten Vorgaben angewendet. Dies führt zu einer verbesserten Performanz des Systems bezüglich der optimalen Verwendung der Ressourcen.

- **Vorgabe der Gesamtressourcen nicht notwendig:** Die zur Verfügung stehenden Gesamtressourcen müssen nicht vor dem Start des Systems bekannt sein.

In Tabelle 2.1 wird in der sich am weitesten rechts befindenden Spalte "wünschenswert" aufgeführt, wie die beste Kombination der Eigenschaften eines Systems aussieht.

Es sollte nicht unbedingt notwendig sein, nur Anytime-Algorithmen als elementare Komponenten des Gesamtsystems verwenden zu können, sondern es sollte auch die Möglichkeit bestehen, auf bekannte Lösungsverfahren zurückzugreifen. Weiterhin sollten wegen der Flexibilität unterschiedliche Lösungsstrategien, Monitoring, die Bearbeitung abhängiger Teilaufgaben und jederzeitige Unterbrechbarkeit unterstützt werden. Damit sollte auch die Vorgabe der Gesamtressourcen entfallen.

die einer *Turing-Maschine* entsprechen. Bei ihnen kann die Berechnung nicht unterbrochen werden, um Zwischenergebnisse abzufragen (s. Abschnitt 3.1.1).

	Forschungsansatz			
Eigenschaft	Compilierung	Design-to-time-Scheduling	Controlled Concurrency	wünschenswert
Möglichkeit der Verwendung von Standard-Algorithmen	-	+	-	+
Möglichkeit der Verwendung von Anytime-Algorithmen	+	+	+	+
Unterstützung unterschiedlicher Lösungsstrategien	-	+	-	+
Bearbeitung abhängiger Teilaufgaben	+	+	-	+
Unterbrechbarkeit des konstruierten Systems	-	-	+	+
Monitoring	+	+	+	+
Vorgabe der Gesamtressourcen nicht notwendig	-	-	+	+

Tabelle 2.1: Vergleich der verschiedenen Forschungsansätze zur ressourcenbeschränkten Berechnung

Eine Metakontrolle für ein beliebiges System, die diese Eigenschaften ermöglicht, wird in Kapitel 4 beschrieben.

2.5 Zusammenfassung

In diesem Kapitel wurde in das Gebiet der ressourcenbeschränkten Berechnung eingeführt. Dazu wurde der Begriff der "Ressource" in dieser Arbeit festgelegt. Anschließend folgte eine Erläuterung unterschiedlicher Arten der Ressourcensensitivität, insbesondere wurde die Ressourcenadaptivität betrachtet. Es schlossen sich theoretische Grundlagen der Vorgehensweise zum Umgang mit Ressourcenbeschränkungen eines Agenten an, deren Verwendung innerhalb einiger Beispiele praktischer Forschungsansätze erläutert wurde. Die wichtigsten in diesem Zusammenhang zu nennenden Begriffe sind der Anytime-Algorithmus und die Compilierung von Anytime-Algorithmen.

Kapitel 3

Anytime-Berechnung

In diesem Kapitel werden der Begriff des Anytime-Algorithmus und die Compilierung von Anytime-Algorithmen genauer vorgestellt. Sie besitzen grundlegenden Charakter für das in Kapitel 4 beschriebene System zum Ressourcenmanagement in Anytime-Systemen und sind deshalb im Rahmen der vorliegenden Arbeit von besonderem Interesse.

Nachdem bislang allgemein von "Ressourcen" gesprochen wurde, wird in diesem Kapitel nur die spezielle Ressource "Zeit" betrachtet.

3.1 Anytime-Algorithmen

Die Anforderungen und Aufgaben, die an Computer gestellt werden, werden immer komplexer und fordern somit auch immer größere Mengen an zeitlichen Ressourcen, die nicht alleine durch die Weiterentwicklung der Hardware (beispielsweise durch Verwendung von schnelleren Prozessoren) bereitgestellt werden können. Deshalb müssen auch Möglichkeiten im Bereich der Entwicklung von Algorithmen gefunden werden, die die Lösung dieses Problems unterstützen. Ein solcher Ansatz ist das von Dean und Boddy entwickelte Konzept der *Anytime-Algorithmen* (Dean & Boddy, 1988).

3.1.1 Begriffsbildung

Die allgemein verwendeten Algorithmen, die nach der in Abbildung 3.1(a) dargestellten sequentiellen Vorgehensweise *Eingabe* $\rightarrow$ *Berechnung* $\rightarrow$ *Ergebnis* arbeiten, liefern das korrekte Ergebnis erst nach Abarbeitung aller Arbeitsschritte, wozu eine bestimmte Zeitspanne benötigt wird. Vor Beendigung dieser Zeitspanne wird ein Zugriff auf das Ergebnis oder gar auf eventuell generierte Zwischenergebnisse nicht unterstützt. Dies entspricht dem Konzept der *Turing-Maschine* (s. beispielsweise (Lewis & Papdimitriou, 1981)). Eine Berechnung wird dort definiert als der Prozeß, von einem Anfangszustand zu einem Endzustand zu gelangen, mit einer Problemstellung als Eingabe und dem Ergebnis als Ausgabe. Ein Algorithmus ist die Implementierung einer *berechenbaren Funktion*. Das bedeutet, die sie repräsentierende Turing-Maschine hält bei jeder Eingabe nach einer bestimmten endlichen Anzahl von Schritten an und liefert ein Ergebnis.

Man kann einen solchen Algorithmus als eine Funktion

$$f_A : I \longrightarrow R \tag{3.1}$$

auffassen, die ein Element aus der Menge der möglichen Eingaben I auf das zugehörige Element der Menge der möglichen Ergebnisse R abbildet.

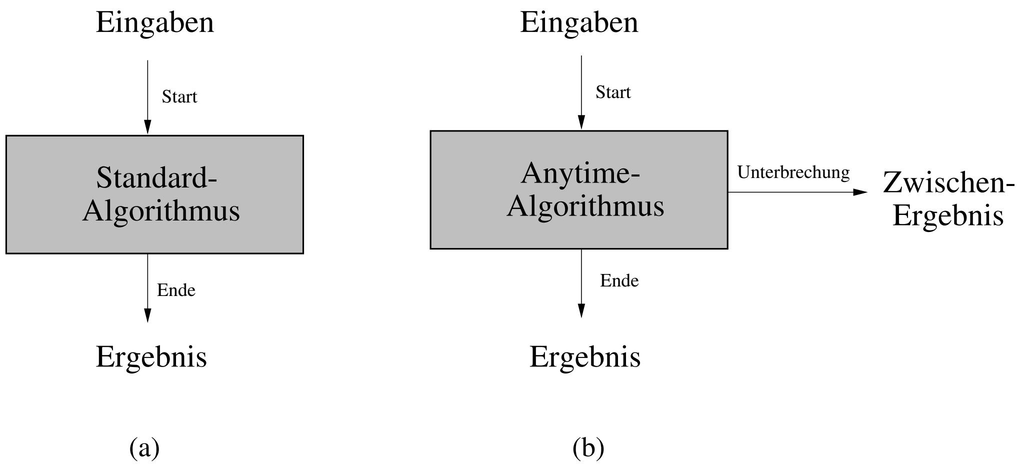

Abbildung 3.1: Arbeitsweisen verschiedener Algorithmentypen

Im Zusammenhang mit dieser Art von Algorithmen kann man von einer Korrektheit eines Ergebnisses im booleschen Sinne sprechen. Das Ergebnis ist immer korrekt. In der Praxis können durch fehlerhafte Implementierungen von Algorithmen aber auch falsche Ergebnisse auftreten.

Anytime-Algorithmen können im Gegensatz zu den bisher betrachteten Algorithmen jederzeit unterbrochen werden und das bis zu diesem Zeitpunkt ermittelte (Zwischen-)Ergebnis bereitstellen (vgl. Abbildung 3.1(b)). Hier kann man allerdings nicht mehr von der “Korrektheit” eines Ergebnisses im booleschen Sinne sprechen, sondern muß zusätzlich die *Qualität* $q_{AA}(t)$ des Ergebnisses des Anytime-Algorithmus AA zum Zeitpunkt t angeben. Interpretiert man einen Anytime-Algorithmus als Abbildung, so erhält man eine Funktion von Elementen der Eingabemenge I auf Paare, bestehend aus einem Ergebnis aus der Ergebnismenge R und der dazugehörigen Qualität (üblicherweise aus dem Intervall $[0.0, 1.0]$):

$$f_{AA} : I \longrightarrow R \times [0.0, 1.0] \tag{3.2}$$

Ein Wert von 0.0 kennzeichnet ein unbrauchbares Ergebnis und ein Wert von 1.0 bedeutet, daß es sich um ein bestmögliches Ergebnis handelt.

Wegen diesen Eigenschaften des Algorithmus ist es möglich, jederzeit eine an ihn gestellte Anfrage zu beantworten, und eine Näherungslösung zur Verfügung zu stellen. Ein einfaches Beispiel ist das Newton-Verfahren zur Bestimmung der Nullstelle einer differenzierbaren Funktion (s. beispielsweise (Königsberger, 1990)).

Eine weitere Möglichkeit das Konzept des Anytime-Algorithmus anzuwenden, besteht darin, ihm vor Beginn eine feste Zeitspanne zuzuteilen, nach deren Ablauf das bis dahin berechnete Ergebnis abgefragt wird. Je mehr Zeit zur Verfügung gestellt wird, desto besser ist die Qualität des Ergebnisses. Dies ist eine der charakteristischen Eigenschaften eines Anytime-Algorithmus, die im nächsten Abschnitt genauer betrachtet werden.

3.1.2 Eigenschaften

Aus den bisherigen Ausführungen wird deutlich, daß sich das Konzept des Anytime-Algorithmus auf mehrere Arten von dem der üblicherweise verwendeten Algorithmen unterscheidet. Die folgenden Eigenschaften charakterisieren nach Dean und Boddy sowie (Grass, 1996) einen Anytime-Algorithmus:

- **Qualitätsmaß:** Wie im vorigen Abschnitt erläutert, liefert ein Anytime-Algorithmus anstatt eines korrekten oder eines falschen Ergebnisses (im booleschen Sinne), ein Ergebnis zusammen mit einer Qualität zurück. Diese gibt an, wie brauchbar dieses für den Benutzer[1] ist und/oder wie nahe es am Endergebnis liegt, das errechnet würde, wenn der Algorithmus komplett durchgearbeitet wird. Eine genauere Betrachtung des Qualitätsmaßes wird in Abschnitt 3.1.4 vorgenommen.

- **Prognostizierbarkeit:** Zu einem Anytime-Algorithmus gehören Informationen über die Qualität seiner (Zwischen-)Ergebnisse. Es kann zumindest prognostiziert werden, in welchem Bereich die Qualität seines Ergebnisses nach einer bestimmten Zeitspanne liegt. Diese Werte können auf verschieden Arten erhalten werden, wie z.B. durch die Analyse früherer Durchläufe des Algorithmus. Hierauf wird ebenfalls im Abschnitt 3.1.4 näher eingegangen.

- **Unterbrechbarkeit und Fortsetzbarkeit:** Ein Anytime-Algorithmus kann jederzeit mit vernachlässigbarem administrativen Aufwand unterbrochen werden und das zu diesem Zeitpunkt aktuelle Zwischenergebnis abgefragt werden. Danach besteht die Möglichkeit, daß der Algorithmus aus dem Unterbrechungszustand heraus weiter fortgesetzt wird bis die nächste Anfrage gestellt wird oder bis er terminiert.

- **Monotonie:** Die Qualität des jeweils aktuellen Ergebnisses eines Anytime-Algorithmus verhält sich monoton steigend, d.h. sie steigt mit wachsender Zeit an oder bleibt zumindest gleich.

Bei der Entwicklung von Anytime-Algorithmen müssen alle diese Eigenschaften berücksichtigt werden, sonst werden eventuell unerwünschte oder sogar falsche Ergebnisse geliefert. Wäre beispielsweise die Eigenschaft der Monotonie verletzt, erhielte man bei einer längeren Zeitspanne unter Umständen schlechtere Resultate als bereits ermittelt. Eine Vorgehensweise zur Konstruktion von Anytime-Algorithmen wird in Abschnitt 3.1.5 beschrieben.

3.1.3 Contract-Anytime-Algorithmen und unterbrechbare Anytime-Algorithmen

Als Weiterentwicklung des Konzepts von Dean und Boddy wird von Zilberstein (Zilberstein, 1993) die Unterscheidung zwischen *Contract-Anytime-Algorithmen*[2] und

[1] Der Begriff 'Benutzer' steht hier sowohl für einen menschlichen Bediener als auch für ein übergeordnetes Programm, das den Anytime-Algorithmus als Unterprogramm benutzt.

[2] engl. contract: Vertrag. Mit dieser Bezeichnung wird ausgedrückt, daß dem Algorithmus bevor er gestartet wird, eine gewisse Zeit 'vertraglich' zugesichert wird, die er auf jeden Fall zur Verfügung

unterbrechbaren Anytime-Algorithmen[3] eingeführt. Letztere liefern immer ein Ergebnis mit der zum Zeitpunkt der Unterbrechung zu erwartenden Qualität, wohingegen ein Contract-Anytime-Algorithmus eine bestimmte Zeitspanne benötigt bis ein Ergebnis verfügbar ist. Werden sie vorher unterbrochen, so kann dies zu nichtbrauchbaren Ergebnissen führen. Somit entspricht der Typ der unterbrechbaren Anytime-Algorithmen dem Ansatz von Dean und Boddy und repräsentiert den allgemeineren Fall. Jeder unterbrechbare Anytime-Algorithmus ist zugleich ein Contract-Anytime-Algorithmus, der keine Einschränkungen bezüglich festgelegter Unterbrechungszeitpunkte besitzt.

In der Praxis ist es allerdings meist einfacher Contract-Anytime-Algorithmen zu entwickeln, da diese nicht jederzeit auf die Möglichkeit einer Unterbrechung vorbereitet sein müssen. Allgemein versucht man aber genau diese Eigenschaft der Unterbrechbarkeit zu jedem beliebigen Zeitpunkt zu erreichen. In diesem Fall liefert der *Reduktionssatz von Zilberstein* eine Lösung, um aus einem Contract- einen unterbrechbaren Anytime-Algorithmus zu entwickeln:

Satz 3.1.1 (Reduktionssatz von Zilberstein) *Für jeden Contract-Anytime-Algorithmus A kann ein unterbrechbarer Anytime-Algorithmus B konstruiert werden, so daß für jede feste Eingabe gilt:*

$$q_B(4t) \geq q_A(t), \tag{3.3}$$

das heißt die Qualität des Ergebnisses von B zum Zeitpunkt $4t$ ist besser oder gleich der Qualität des Ergebnisses von A zum Zeitpunkt t.

Die Idee dieses Satzes basiert auf der Möglichkeit den unterbrechbaren Anytime-Algorithmus A durch nacheinander stattfindenden Läufe des Contract-Anytime-Algorithmus B mit exponentiell ansteigender Laufzeit zu konstruieren. Für den ausführlichen Beweis wird auf (Zilberstein, 1993, S. 38) verwiesen.

3.1.4 Qualitätsmaß und Performanzprofile

Die in diesem Abschnitt eingeführten und erläuterten Begriffe stellen einen wichtigen Aspekt innerhalb des Konzepts der Anytime-Algorithmen dar. Sie sind eine Voraussetzung der Prognostizierbarkeit der Qualität der approximativen Zwischenergebnisse.

Unter dem *Qualitätsmaß* versteht man in diesem Zusammenhang eine Zuordnung des Ergebnisses zu einem Wert aus dem Intervall $[0.0, 1.0]$, der angibt, wie "gut" das Ergebnis ist. Für die Wahl der *Metrik* bei dieser Zuordnung gibt es verschiedene Möglichkeiten, die im folgenden erläutert werden.

---

hat, um seine Berechnungen durchzuführen. Dies stellt in gewissem Sinne einen Widerspruch zum allgemeinen Konzept des Anytime-Algorithmus dar.

[3] Die Nomenklatur ist an dieser Stelle irreführend, denn ein Anytime-Algorithmus ist immer unterbrechbar. Der Zusatz "unterbrechbar" dient lediglich zur Abgrenzung gegenüber der Contract-Anytime-Algorithmen, die nur zu vorher bestimmten ausgewählten Zeitpunkten unterbrochen werden können.

3.1.4.1 Qualitätsmetriken

Die Wahl der Metrik zur Bewertung der Qualität der Ergebnisse eines Anytime-Algorithmus ist abhängig von der Anwendung, in der der Anytime-Algorithmus arbeitet. Nach (Zilberstein & Russell, 1996) unterscheidet man im wesentlichen die drei folgenden Typen:

- **Sicherheit:** Diese Metrik bestimmt, in wie weit es zutrifft, daß das berechnete Resultat korrekt ist. Üblicherweise werden dazu Wahrscheinlichkeitswerte verwendet. Als Beispiel hierfür kann die Metrik eines medizinischen Diagnosesystems angeführt werden. Jeder von ihm produzierten Diagnose kann eine Wahrscheinlichkeit bezüglich der Korrektheit zugeordnet werden.

- **Genauigkeit:** Hiermit wird angegeben, wie stark das aktuelle Ergebnis vom exakten abweicht, z.B. durch Angabe einer oberen Schranke für die Abweichung. Solche Schranken treten häufig in numerischen Domänen auf, beispielsweise die Angabe der Schranke bei der Taylorreihenentwicklung mittels des Lagrangen Restgliedes (s. beispielsweise (Bronstein & Semendjajew, 1991)). Dieses Beispiel verdeutlicht die Tatsache, daß schon vor der Formalisierung des Konzepts des Anytime-Algorithmus in der Künstlichen Intelligenz solche approximativen Algorithmen in anderen Gebieten der Wissenschaft genutzt wurden.

- **Spezifität:** Diese Metrik reflektiert den Detaillierungsgrad des Ergebnisses, z.B. innerhalb eines hierarchischen Planers. Je tiefer die Ebene liegt, auf deren Betrachtung das Resultat basiert, desto besser ist die Qualität.

3.1.4.2 Performanzprofile

Die Repräsentation der mit Hilfe einer Qualitätsmetrik bestimmten Werte für die Zwischenergebnisse geschieht mit Hilfe eines *Performanzprofils* (PP) (Dean & Boddy, 1988). Es charakterisiert die Qualität der Ergebnisse des zugehörigen Anytime-Algorithmus in Abhängigkeit von der zur Berechnung benötigten Zeit.

Formal gilt für die Qualität q:

$$q : \mathbb{R}_{\geq 0} \longrightarrow [0.0, 1.0]. \tag{3.4}$$

Abbildung 3.2 zeigt drei typische Performanzprofile für verschiedene Arten von Algorithmen. Graph (a) symbolisiert das Verhalten eines Standard-Algorithmus. Der sprunghafte Anstieg der Qualität von 0.0 auf 1.0 ist charakteristisch. Bevor nicht alle Schritte des Algorithmus abgearbeitet sind, liegt kein brauchbares Ergebnis vor, danach ändert sich die Qualität nicht mehr. Graph (b) stellt das Verhalten eines idealisierten Anytime-Algorithmus dar: Ein stetiger Anstieg der Qualität des Ergebnisses mit fortschreitender Zeit. Der letzte Graph (c) zeigt ein Performanzprofil, wie es in der Praxis häufig auftritt: Der Graph steigt stufenweise an, das bedeutet die Qualität ändert sich immer dann, wenn in sich geschlossene Teilaufgaben des Algorithmus gelöst worden sind.

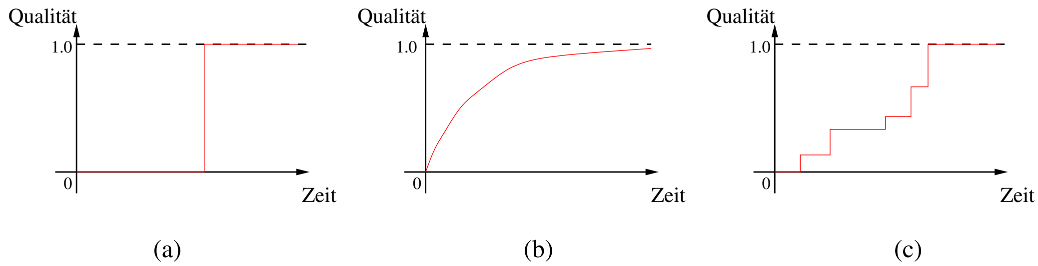

Abbildung 3.2: Performanzprofile für verschiedene Algorithmentypen

3.1.4.3 Bestimmung von Performanzprofilen

Es gibt verschiedene Arten, die Performanzprofile zu generieren (vgl. (Grass, 1996)):

- **statistische Schätzungen:** Auf der Basis früherer Läufe des Algorithmus werden statistische Werte über den zeitlichen Verlauf der Qualität ermittelt, wobei untersucht werden muß, welche Anzahl an Durchläufen benötigt wird, um verläßliche Ergebnisse zu erhalten.

- **Projektionen:** Aufgrund des bisherigen Verlaufs der Qualitätswerte wird eine Voraussage über das zukünftige Verhalten gemacht. Man kann am aktuellen Graph die Tangente mit der augenblicklichen Steigung anlegen und somit versuchen, diese Vorhersagen zu treffen (s. Abbildung 3.3). Um zum Zeitpunkt t_1 die erwartete Qualität zur Zeit t_2 zu bestimmen, wird der Graph mit der aktuellen Steigung fortgesetzt (gestrichelte Linie). Der reale Verlauf ist durchgezogen dargestellt. Die durch Projektion bestimmte Qualität liegt in diesem Beispiel also etwas über der in der Realität auftretenden.

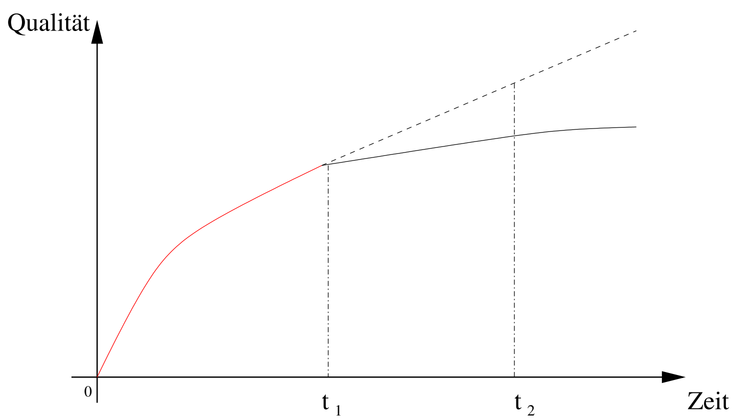

Abbildung 3.3: Projektion eines Performanzprofils

- **algorithmische Schätzungen:** In diesem Fall muß es möglich sein, eine geschlossene mathematische Formel anzugeben, die relativ genaue Näherungen für die Qualität bei Angabe der zur Verfügung stehenden Zeit liefert. Als Beispiel kann eine Intervallschachtelung dienen.

In der Praxis ist die Kombination dieser verschiedenen Methoden sinnvoll. Liegen keine Erfahrungen früherer Durchläufe vor, so kann man mit dem Verfahren der Projektion arbeiten und durch Aufzeichnen der tatsächlichen Ergebnisse statistische Werte sammeln, die mit größerer Anzahl der Programmläufe immer adäquater werden und in einer späteren Phase zur Voraussage der Qualität verwendet werden können.

3.1.4.4 Repräsentation von Performanzprofilen

Für die Speicherung von Performanzprofilen im Rechner gibt es mehrere Möglichkeiten. Ist die exakte Repräsentation eines Performanzprofiles durch eine mathematische Formel nicht möglich, kann es auf mehrere Arten angenähert werden, beispielsweise durch abschnittsweise lineare Approximation oder durch Approximation mittels einer Exponentialfunktion. Dean und Boddy schlagen hierfür Funktionen der Familie $f(x) = 1 - e^{-\lambda x}$ vor (s. Abbildung 3.4). Sie sind (streng) monoton wachsend und konvergieren gegen den Wert 1.0 und haben somit die geforderten Eigenschaften.

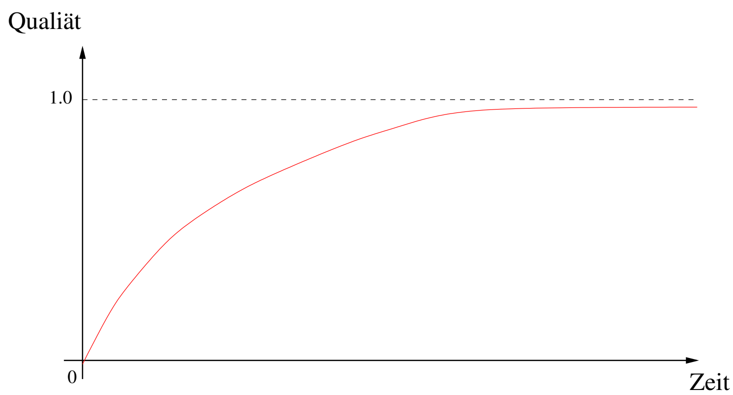

Abbildung 3.4: Exponentialfunktion vom Typ $f(x) = 1 - e^{-\lambda x}$

Eine weitere Möglichkeit ist das Ablegen des Performanzprofils in einer Tabelle wie Tabelle 3.1, einem *Performanzverteilungsprofil* (Zilberstein, 1993). Diese Art der Speicherung basiert auf einer diskreten Aufteilung der Qualitätsachse in Intervalle. Jedem Feld der entstehenden Tabelle wird ein Wert zugeordnet, der angibt, wie groß die Wahrscheinlichkeit ist, daß die Qualität im entsprechenden Intervall liegt, wenn die zugehörige Zeitspanne der Berechnung zur Verfügung steht. Es wird also die Zeit abgebildet auf die Wahrscheinlichkeitsverteilung der Qualität. Im angegebenen Beispiel gilt, daß die Wahrscheinlichkeit dafür, daß die Qualität nach 0.6 Zeiteinheiten im Bereich von 0.2 bis 0.4 liegt, einen Wert von 0.3 besitzt.

3.1.5 Entwicklung

Durch die Unterschiede eines Anytime-Algorithmus im Vergleich zu einem Standard-Algorithmus ergeben sich auch bei der Entwicklung verschiedene Vorgehensweisen. Zilberstein (Zilberstein, 1993) spricht von einer *indirekten Programmiertechnik der Anytime-Algorithmen*, wobei sichergestellt werden muß, daß der Algorithmus eine

	Qualität				
Zeit	0.2	0.4	0.6	0.8	1.0
0.0	1.0	0.0	0.0	0.0	0.0
0.2	0.6	0.3	0.1	0.0	0.0
0.4	0.3	0.5	0.2	0.0	0.0
0.6	0.0	0.3	0.5	0.2	0.0
0.8	0.0	0.0	0.2	0.6	0.2
1.0	0.0	0.0	0.1	0.2	0.7

Tabelle 3.1: Repräsentation eines Performanzprofils in Tabellenform

Reihe von Zwischenergebnissen generiert, anstatt nur die Berechnung des Endergebnisses als Ziel zu besitzen.

3.1.5.1 Implementierung des Verbesserungsschritts

Grass und Zilberstein (Grass & Zilberstein, 1995) haben eine Vorgehensweise und verschiedene Hilfsmittel zur Entwicklung von Anytime-Algorithmen entwickelt. Danach ist das Auszeichnen eines *Verbesserungsschritts* innerhalb der Berechnung des endgültigen Ergebnisses ein wichtiger Aspekt. Nach jedem solchen Schritt wird das aktuelle Ergebnis samt Qualität in einer dafür vorgesehenen Datenstruktur für eventuelle Zugriffe abgelegt. In Abbildung 3.5 ist der Pseudocode für einen prototypischen Anytime-Algorithmus dargestellt. Bevor der eigentliche iterative Abschnitt des Algorithmus in Zeile 3 beginnt, muß eine problemabhängige initiale Lösung generiert werden, um ein Zwischenergebnis im Falle einer Anfrage vor dem ersten Verbesserungsschritt liefern zu können. Danach wird die den Verbesserungsschritt beinhaltende Schleife so lange abgearbeitet, bis entweder ein Unterbrechungssignal erfolgt und das aktuelle Ergebnis angefordert wird oder keine Verbesserung mehr erzielt werden kann, das Abbruchkriterium somit erfüllt ist und der Algorithmus terminiert.

```
    Anytime-Prototyp(Eingaben)
(1) Ergebnis ← initiale_Lösung(Eingaben)
(2) lege_ab(Ergebnis)
(3) while not Unterbrechung and not Abbruchkriterium do
(4)     Ergebnis ← Verbesserungsschritt(Ergebnis)
(5)     lege_ab(Ergebnis)
(6) od
```

Abbildung 3.5: Prototyp eines Anytime-Algorithmus

Bei der Implementation des Verbesserungsschrittes können bekannte Verfahren verwendet werden, so kann beispielsweise bei Suchalgorithmen der Suchbereich iterativ vergrößert werden oder verschränkte Breiten- und Tiefensuche genutzt werden. Im Bereich der randomisierten Algorithmen werden Las-Vegas- oder Monte-Carlo-Algorithmen verwendet. Iterative Algorithmen wie das Newton-Verfahren zur Bestimmung von Nullstellen sind schon Anytime-Algorithmen und bedürfen keiner wesentlichen Änderung. In (Boddy, 1991b) wird ausführlich erläutert, wie der Verbesserungsschritt in bestimmten Fällen unter Verwendung von dynamischem Programmieren implementiert werden kann.

3.1.5.2 Bestimmung des Performanzprofils

Der nächste Schritt nach der Implementierung des Anytime-Algorithmus ist die Bestimmung des zugehörigen Performanzprofils. Hier schlagen Grass und Zilberstein vor, zuerst einige wenige zufällige Probleminstanzen zu kreieren bzw. von einem Tool kreieren zu lassen, und damit eine Schätzung für die durchschnittliche Laufzeit des entwickelten Algorithmus zu bestimmen. Diese wird benötigt, um den Definitionsbereich und gegebenenfalls die Granularität des Performanzprofils bzw. Performanzverteilungsprofils festzulegen. In einem nächsten Schritt werden dann große Mengen von Probleminstanzen durch den Algorithmus bearbeitet, um durch Aufzeichnen des Verhaltens der Qualität der Zwischenergebnisse eine möglichst genaue Approximation des Performanzprofils zu erhalten. Die so ermittelten Werte können dann wie in Abschnitt 3.1.4 beschrieben entweder in Tabellenform oder durch Approximation mittels geeigneter Funktionen festgehalten werden.

3.1.6 Beispiel

In diesem Abschnitt werden die bisher vorgestellten Konzepte und Begriffe anhand des Beispiels eines Anytime-Algorithmus zur approximativen Lösung des *Rucksack-Problems* verdeutlicht: Ein Dieb findet bei einem Einbruch n Gegenstände G, die durch ihr Gewicht und ihren Wert charakterisiert sind. Natürlich will er die Beute mit dem größtmöglichen Wert stehlen, hat aber leider nur einen Rucksack, der eine Kapazität von g Gewichtseinheiten besitzt. Deshalb stellt sich ihm die Frage, welche Gegenstände soll er mitnehmen? (Diese spezielle Formulierung des Problems ist (Cormen, Leiserson & Rivest, 1990) entnommen.)

Der in Abbildung 3.6 dargestellte Algorithmus entspricht in seinem Aufbau dem prototypischen Anytime-Algorithmus aus Abbildung 3.5. Er beginnt mit der Bestimmung der initialen Lösung (Zeile 1). In diesem Beispiel besteht diese in der Auswahl einer zufälligen Menge von Gegenständen, deren Gesamtgewicht die Kapazität des Rucksacks nicht überschreitet. Darauf wendet er dann iterativ den Verbesserungsschritt (Zeile 6 - 18) an.

Das Prinzip der Verbesserung basiert auf dem Austausch von Gegenständen aus dem Rucksack mit solchen, die aktuell nicht enthalten sind, aber unter Berücksichtigung des Gewichts hineinpassen und einen größeren Wert besitzen. Nacheinander werden alle Kombinationen von i Gegenständen, die aus dem Rucksack herausgenommen werden sollen, und j Gegenständen, die dafür hinzugenommen werden können, auf diese Eigenschaft überprüft. Die Funktion `Austausch` bestimmt hierzu das bestmögliche Ergebnis bei Ersetzen von i Elementen aus der aktuellen Lösung L durch j Elemente von $G - L$, die noch in den Rucksack aufgenommen werden können. Falls ein besseres Ergebnis gefunden wird, wird dieses als Zwischenergebnis übernommen und der Schleifendurchlauf beginnt aufs Neue. Dieser Vorgang dauert so lange an, bis ein Unterbrechungssignal gegeben wird oder alle Kombinationen der

```
Anytime-Rucksack(g, G)
L ← initiale_Lösung(g, G)
lege_ab(L)
n ← |G|
i ← 1
j ← 1
while i ≤ n and not Unterbrechung do
    while j ≤ i do
        L' ← Austausch(L, i, j, g, G)
        if Gewicht(L') > Gewicht(L) then
            L ← L'
            i ← 1
            j ← 0
            lege_ab(L)
        fi
        j ← j + 1
    od
    i ← i + 1
od
```

Abbildung 3.6: Anytime-Algorithmus zur Lösung des Rucksack-Problems

Gegenstände überprüft wurden.

Das in Abbildung 3.7 dargestellte idealisierte Performanzprofil für diesen Algorithmus hat die gewünschten Eigenschaften. Es steigt zu Beginn der Berechnung stark an und zeigt gegen Ende nur noch geringere Qualitätsverbesserungen. Es wird wie in Abschnitt 3.1.4 beschrieben durch Analyse mehrerer Durchläufe gewonnen. Als Qualitätsmaß wird in diesem Beispiel das Verhältnis der aktuellen zur einer oberen Schranke der Lösung verwendet.

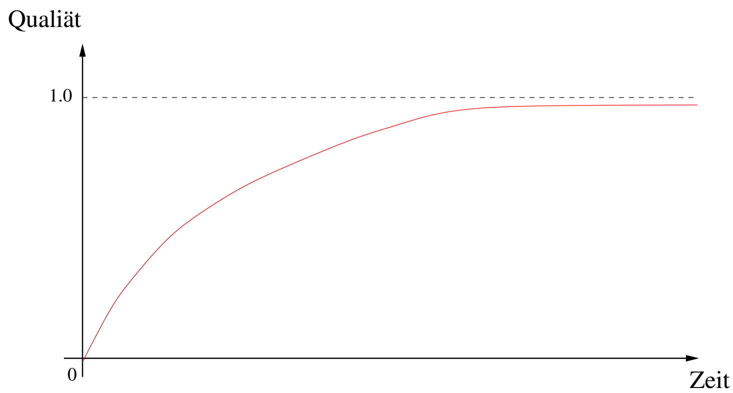

Abbildung 3.7: Idealisiertes Performanzprofil für die Lösung des Rucksack-Problems

Einige praxisnahe Einsatzmöglichkeiten von Anytime-Algorithmen werden im nächsten Abschnitt aufgezeigt.

3.1.7 Anwendungen

Anwendungsgebiete für Anytime-Algorithmen sind alle Bereiche, in denen Zeitrestriktionen auftreten. Ursprünglich entwickelten Dean und Boddy das Konzept anhand des Beispiels des stationären Roboters aus Abschnitt 2.4.1.

Ebenso gibt es im Bereich der *Bayes'schen Netze* Algorithmen, die die Eigenschaften eines Anytime-Algorithmus besitzen, wie die *Stochastische Simulation* bei mehrfach verbundenen Bayes'schen Netzen (s. (Pearl, 1988)). Ein ausführliches Beispiel dazu findet sich in (Brandherm, Wagner & Wittig, 1997).

Ein neu erschlossenes Gebiet der Anwendung von Anytime-Algorithmen ist das Verarbeiten *natürlicher Sprache* (Görz & Kesseler, 1994b, 1994a). Diese Arbeiten, die im Rahmen des VERBMOBIL-Projekts[4] (Wahlster, 1993) entstanden sind, führen auch das Konzept der *Transaktion* ein. Dabei handelt es sich um eine in sich abgeschlossene Einheit, die in ihrer Abarbeitung nicht unterbrochen werden darf, das heißt eine Unterbrechungsanforderung an den Anytime-Algorithmus wird so lange hinausgezögert, bis eine begonnene Transaktion beendet ist. Transaktionen sind ein wichtiges Konzept innerhalb des in dieser Arbeit vorgestellten Systems und werden später in Abschnitt 4.1.4 noch genauer betrachtet. Sie stehen in engem Zusammenhang mit Contract-Anytime-Algorithmen.

Die vorliegende Arbeit entstand im Rahmen der Entwicklung eines Systems zur *Objektlokalisation* und sollte die bereits im Vorgängersystem OLS (Gapp, 1997) integrierten Anytime-Eigenschaften erweitern und verbessern. Eine weitere Arbeit, die sich mit Anytime-Berechnung in diesem Anwendungsgebiet der *sprachlichen Raumbeschreibung* befaßt, ist (Blocher & Stopp, 1998).

Diese Beispiele verdeutlichen die Vielfalt der Anwendungsmöglichkeiten von Anytime-Algorithmen in unterschiedlichsten Gebieten der Künstlichen Intelligenz.

3.2 Compilierung nach Zilberstein

Bisher wurden nur einzelne isoliert arbeitende Anytime-Algorithmen betrachtet. Die Möglichkeit der Konstruktion von komplexen Systemen aus einzelnen Anytime-Komponenten bietet die *Compilierung von Anytime-Algorithmen nach Zilberstein* (Zilberstein, 1993). Gerade in der Künstlichen Intelligenz sind viele Probleme nicht durch die Verwendung eines einzelnen Anytime-Algorithmus lösbar. In vielen Anwendungen sind die Teilaufgaben zu unterschiedlich, um innerhalb eines einzelnen solchen Algorithmus bearbeitet zu werden. Betrachtet man z.B. einen Roboter, der mittels einer Kamera seine Umgebung wahrnimmt und dann einen Weg hindurch bestimmen muß, dann wird deutlich, daß er zwei klar gegeneinander abgegrenzte Komponenten besitzen muß.

3.2.1 Prinzip

Die Problemstellung, die sich hier bei der Kombination mehrerer Anytime-Algorithmen ergibt, lautet: Finde eine Verteilung der zur Verfügung stehenden Gesamt-

[4] "Verbmobil ist ein Sprachtechnologieprojekt im Bereich Maschinelle Übersetzung. Verbmobil erkennt gesprochene Sprache, analysiert die Eingabe, übersetzt sie ins Englische, erzeugt einen Satz und spricht ihn aus." (Zitat aus der Web-Seite des VERBMOBIL-Projektes (VERBMOBIL, 1991))

zeit t derart, daß das Ergebnis des Gesamtsystems nach Laufzeit t die zu diesem Zeitpunkt bestmögliche Qualität aufweist. Das bedeutet, es sollen die Anforderungen der beschränkten Optimalität erfüllt werden (s. Abschnitt 2.3).

Bei diesem Ansatz müssen die zu verteilenden Zeitressourcen vor dem Starten der Berechnung bekannt sein. Eine Unterbrechung des Gesamtsystems, die von dieser Vorgabe abweicht und bei diesem Verfahren auch nicht vorgesehen ist, liefert dann im allgemeinen ein suboptimales bzw. überhaupt kein Ergebnis.

Zilberstein löst das Problem der optimalen Ressourcenverteilung mit Hilfe einer *Offline-Compilierung* der einzelnen Komponenten zu einem Gesamtmodul, deren Ablauf in Abbildung 3.8 dargestellt ist.

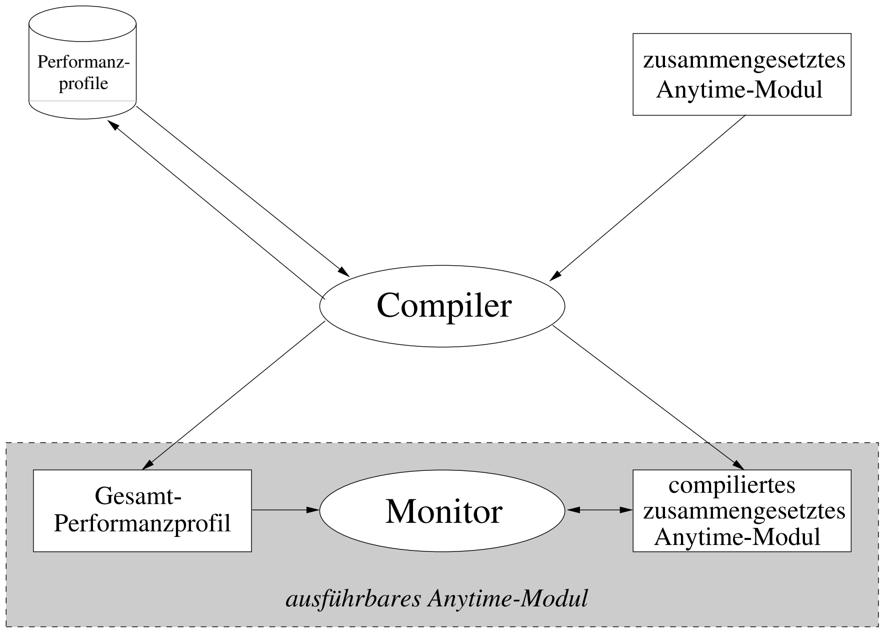

Abbildung 3.8: Compilierung nach Zilberstein (aus (Zilberstein, 1993, S. 56))

Der *Compiler* erhält zwei Eingaben: Einerseits ein aus den einzelnen Anytime-Algorithmen aufgebautes, noch *nicht lauffähiges Modul*, das noch keine Informationen zur Ressourcenverteilung beinhaltet, und andererseits die zugehörigen Performanzprofile. Diese stammen aus einer *Bibliothek*, in der die entsprechenden Performanzprofile, der zur Verfügung stehenden Anytime-Algorithmen, abgelegt sind. Daraus generiert der Compiler ein *ausführbares Anytime-Modul*, das aus drei Bestandteilen aufgebaut ist: Einem für dieses Anytime-Modul aus den Einzelperformanzprofilen bestimmtes *Gesamtperformanzprofil*, einem *Monitor* und dem *compilierten zusammengesetzten Anytime-Modul*. Diese drei Komponenten besitzen alle notwendigen Informationen und Mechanismen, um das gewünschte Verhalten des Gesamtsystems zu gewährleisten. Der Monitor kontrolliert unter Verwendung des Gesamtperformanzprofils die korrekte Aktivierung und Beendigung der verschiedenen Komponenten innerhalb des ausführbaren Anytime-Moduls.

In den nächsten Abschnitten wird zunächst eine in diesem Zusammenhang benötigte Erweiterung des Begriffs des Performanzprofils eingeführt. Danach werden die Faktoren betrachtet, die einen Einfluß auf die Komplexität des Compilierungsvorgangs haben. Damit sind dann die Grundlagen vorhanden, um die Vorgehensweise an sich und die dabei auftretenden Probleme zu erläutern.

3.2.2 Bedingte Performanzprofile

Durch die Konstruktion des Gesamtsystems aus einzelnen Anytime-Algorithmen besteht die Möglichkeit, daß einer das Ergebnis eines oder mehrerer anderen als Eingabe benutzen muß. Damit ist die Qualität seiner Berechnung nicht mehr alleine von der benötigten Zeit abhängig, sondern auch von der Qualität der Eingabe. Ein Beispiel hierfür ist ein medizinisches System zur Erkennung und Behandlung von Krankheiten. Es besteht aus zwei Anytime-Algorithmen, die erst anhand der Symptome eine Diagnose stellen und dann darauf basierend eine Behandlungsmethode vorschlagen. Je besser die Qualität der Diagnose ist, desto sicherer kann man sein, daß die richtige Behandlungsmethode vorgeschlagen wird. Das bedeutet, daß der Algorithmus zur Bestimmung der Behandlung bezüglich der Qualität seiner Ergebnisse abhängig ist von der Qualität der Ergebnisse des Diagnosealgorithmus. Dieses Problem führt zu einer Erweiterung des vorgestellten Konzepts des Performanzprofils, dem *bedingten Performanzprofil*.

Das bedingte Performanzprofil q_{kond} berücksichtigt die Qualität der Eingaben und bildet diese zusammen mit der Laufzeit und der betrachteten Qualität auf deren Wahrscheinlichkeit ab. Damit erhält man formal die Abbildung

$$q_{kond} : [0.0, 1.0] \times \mathbb{R}_{\geq 0} \times [0.0, 1.0] \longrightarrow [0.0, 1.0]. \tag{3.5}$$

Besitzt ein Anytime-Algorithmus n verschiedene Eingaben und will man deren unterschiedliche Qualitäten stärker berücksichtigen, so kann man in obiger Abbildung das erste Intervall durch das n-fache direkte Produkt des Intervalls $[0.0, 1.0]$ ersetzen und somit eine genauere Abhängigkeit der Qualität des Ergebnisses modellieren.

Für die rechnerinterne Repräsentation bedingter Performanzprofile muß man für alle Eingabequalitäten, die berücksichtigt werden sollen, ein entsprechendes Performanzverteilungsprofil speichern. Deren Gesamtheit bildet das bedingte Performanzprofil. In Abbildung 3.9 werden die Verläufe dieser Performanzverteilungsprofile für verschiedene Eingabequalitäten anhand ihrer wahrscheinlichsten Ergebnisqualitäten gezeigt.

Hier werden nur vier verschiedene Möglichkeiten für die Qualität der Eingabe betrachtet. Die Genauigkeit der Modellierung richtet sich nach der Anzahl der möglichen Eingabequalität. Je größer diese Zahl ist, desto besser wird das bedingte Performanzprofil approximiert. Der durchgezogene Graph steht für das Verhalten der Qualität bei Eingabequalität 1.0, der gestrichelte für das bei 0.8, der gepunktete für den Wert 0.4 und der letzte für 0.2. Damit kann dann bei gegebener Eingabequalität die erwartete Qualität des zu diesem bedingten Performanzprofil gehörenden Anytime-Algorithmus bestimmt werden.

Ebenso wie es hier verschiedene Genauigkeitsgrade gibt, existieren noch andere Faktoren, die die Komplexität des Compilierungsproblems beeinflußen.

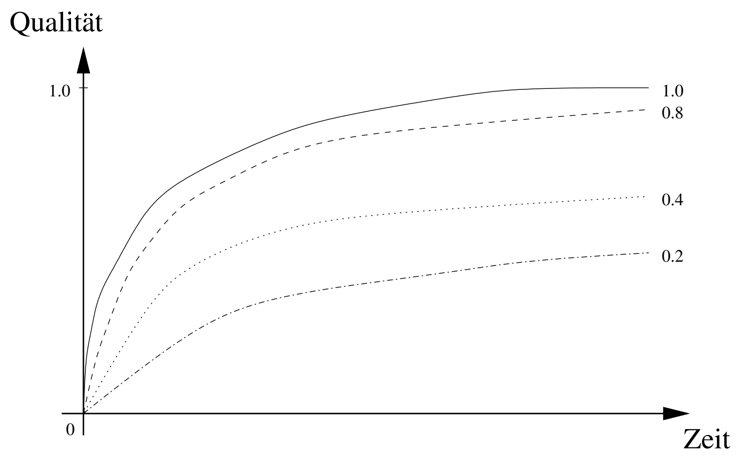

Abbildung 3.9: Beispiel eines bedingten Performanzprofils

3.2.3 Faktoren

Bevor die Vorgehensweise der Compilierung von Anytime-Algorithmen nach Zilberstein näher beschrieben wird, werden in diesem Abschnitt die Faktoren diskutiert, die Auswirkungen auf die Lösung dieses Problems und dessen Komplexität haben. Nach Zilberstein und Russell (Zilberstein, 1996; Zilberstein & Russell, 1996) sind dies folgende:

1. **Kompositionsstruktur:** Die Art der Verknüpfung der einzelnen Anytime-Algorithmen spielt eine sehr große Rolle bei der Komplexität der Compilierung. Einfach zu behandeln ist beispielsweise eine sequentielle Verknüpfung, wohingegen eine rekursive Struktur sehr aufwendig zu compilieren ist.

2. **Art der Performanzprofile:** Je feiner die Granularität bei der rechnerinternen Repräsentation eines Performanzprofils ist, desto aufwendiger wird es, das Gesamtperformanzprofil zu generieren[5].

3. **Art der Anytime-Algorithmen:** Sowohl bei der Auswahl der Algorithmen, die bei der Konstruktion des Systems gewählt werden, als auch bei der Entscheidung, welcher Art das Gesamtsystem sein soll, gibt es immer die Wahl zwischen Contract- und unterbrechbaren Anytime-Algorithmen. Der einfachere Fall ist immer der erstere.

4. **Art des Monitorings:** Es gibt zwei Möglichkeit des Monitorings: *passives* und *aktives* Monitoring. Bei ersterem werden alle Scheduling-Entscheidungen der Metaebene vor dem Start des Systems getroffen. Aktives Monitoring ermöglicht hingegen ein Anpassen der Vorgehensweise an die auftretenden Umstände zur Laufzeit. Wenn beispielsweise eine Komponente wesentlich bessere Ergebnisse liefert als erwartet, können verbleibende Ressourcen umverteilt werden. Eine genauere Betrachtung findet sich in Abschnitt 3.2.8.

[5]Dies ist nur der Fall, falls eine ähnlich feine Granularität im Gesamtsystem benötigt wird.

5. **Aufwand zur Bestimmung der Qualität der Zwischenergebnisse:** Unter diesem Punkt wird der Aufwand zur Bestimmung der aktuellen Qualität eines Zwischenergebnisses einer der Komponenten angesprochen. Der Monitor muß die Möglichkeit haben auf diese Werte zuzugreifen und anhand ihrer Entwicklung eventuelle Veränderungen an der Ressourcenverteilung vorzunehmen. Ist dieser Zugriff bzw. die Vorbereitungen dazu aufwendig, so beeinflußt dies auch die Komplexität des Compilierungsprozesses negativ.

Vor der Konstruktion und Compilierung einzelner Anytime-Algorithmen zu einem Gesamtsystem müssen diese Aspekte untersucht und entschieden werden, welche Vorgehensweisen im jeweiligen Fall sinnvoll sind.

3.2.4 Compilierungsvorgang

Der zentrale Compilierungsvorgang von n Anytime-Komponenten besteht aus mehreren Schritten. Zu Beginn steht die Festlegung der Kombinationsart der einzelnen Anytime-Algorithmen. Die beiden am häufigsten auftretenden Typen sind *sequentiell* (eine Nacheinanderausführung) und *rekursiv* (der Algorithmus ruft sich innerhalb seiner Abarbeitung selbst auf). Eine *parallele* Ausführung ist in diesem Zusammenhang ein Spezialfall der sequentiellen: Die beiden betrachteten Komponenten arbeiten in einer beliebigen Reihenfolge nacheinander, keiner der beiden erhält ein Ergebnis des anderen als Eingabe. Die Kombinationsart muß dem Compiler mitgeteilt werden, damit er das korrekt-compilierte ausführbare Anytime-Modul erzeugen kann. Werden zwei Algorithmen z.B. sequentiell verknüpft, so hängt er ihre Programmsequenzen entsprechend der Vorgabe aneinander.

Danach erfolgt die Bestimmung der Verknüpfungsart der n Einzelqualitäten $q_i, 1 \leq i \leq n$, zur Gesamtqualität q. Einfache Beispiele hierfür sind die Addition und die Multiplikation, je nach Anwendung können aber auch komplexere Operatoren auftreten. Um die optimale Ressourcenverteilung $t_1, \ldots, t_n$ auf die einzelnen Komponenten zu bestimmen, muß nun das globale Maximum der Gesamtqualität berechnet werden. Dazu muß die Ableitung dieser n-dimensionalen Funktion gleich Null gesetzt werden. Man erhält folgendes Gleichungssystem aus $n-1$ Differentialgleichungen:

$$\frac{\partial q}{\partial t_1} = 0 \tag{3.6}$$

$$\vdots \tag{3.7}$$

$$\frac{\partial q}{\partial t_{n-1}} = 0. \tag{3.8}$$

Damit hat man die gewünschte Ressourcenverteilung berechnet.

Im nächsten Abschnitt wird der Vorgang der Compilierung und dabei auftretende Probleme an einem einfachen Beispiel aus (Zilberstein, 1993) erläutert.

3.2.5 Beispiel

Es sollen zwei gegebene Anytime-Algorithmen A_1 und A_2 sequentiell verknüpft und zu einem Gesamtalgorithmus compiliert werden. Die beiden zugehörigen Performanzprofile seien durch die exponentiellen Funktionen $q_1(x) = 1 - e^{-\lambda_1 x}$ und

$q_2(x) = 1 - e^{-\lambda_2 x}$ determiniert ($\lambda_1, \lambda_2 \in \mathbb{R}_{>0}$). Es stehe die Gesamtzeit t zur Verfügung, die so auf die beiden Komponenten verteilt werden muß, daß eine optimale Gesamtqualität erreicht wird. Diese errechne sich im Beispiel als Summe der Einzelqualitäten der beteiligten Anytime-Algorithmen:

$$q(x) = q_1(x) + q_2(x) = 1 - e^{-\lambda_1 x} + 1 - e^{-\lambda_2 (t-x)}. \tag{3.9}$$

(Der Ausdruck $t - x$ im zweiten Teil der obigen Formel rührt von der Tatsache, daß wenn dem ersten Algorithmus x Zeiteinheiten zugeteilt werden nur noch $t - x$ für den zweiten vorhanden sind.)

Zur Bestimmung des Maximums dieser Funktion muß folgende Gleichung gelöst werden:

$$\frac{\partial q}{\partial x} = \lambda_1 e^{-\lambda_1 x} - \lambda_2 e^{-\lambda_2 (t-x)} = 0. \tag{3.10}$$

Dies liefert folgendes Ergebnis:

$$x_0 = \frac{\ln \lambda_1 - \ln \lambda_2 + \lambda_2 t}{\lambda_1 + \lambda_2}. \tag{3.11}$$

Damit hat man eine in Bezug auf die Gesamtqualität optimale Aufteilung der Gesamtzeit, wenn A_1 x_0 und A_2 $t - x_0$ Zeiteinheiten erhält.

Arbeitet man mit Tabellenrepräsentationen der Performanzprofile, so transformiert sich das Problem des Lösens der Gleichung in ein Suchproblem auf den Einträgen der Tabellen.

Betrachtet man den allgemeinen Fall von n verschiedenen Anytime-Algorithmen, so muß man wegen der höheren Dimension des Problems, wie im vorigen Abschnitt erläutert, ein Gleichungssystem aus $n - 1$ Differentialgleichungen lösen:

$$\frac{\partial q}{\partial t_1} = 0 \tag{3.12}$$

$$\vdots \tag{3.13}$$

$$\frac{\partial q}{\partial t_{n-1}} = 0. \tag{3.14}$$

Sind die Performanzprofile in Tabellen abgelegt, so erhöht sich der Suchraum dieses Optimierungsproblems im Vergleich zum Beispiel mit zwei Komponenten. Die Anzahl der verschiedenen Möglichkeiten, t Zeiteinheiten auf n Komponenten zu verteilen, ist nach (Zilberstein, 1993):

$$\frac{(t + n - 1)!}{t!(n - 1)!}. \tag{3.15}$$

Damit ist die Größe des Suchraums exponentiell in n und t, das heißt es ist im allgemeinen nicht mehr möglich, in einer angemessenen Zeit, eine Ressourcenverteilung zu bestimmen. Zilberstein zeigt, daß das zugehörige Entscheidungsproblem NP-vollständig ist. Die im nächsten Abschnitt vorgestellte Technik bietet eine Möglichkeit, akzeptable Lösungen bei geringerer Komplexität zu erreichen.

3.2.6 Lokale Compilierung

Die im letzten Abschnitt vorgestellte Berechungsvorschrift geht von einer globalen Sicht des Problems aus. Zilberstein bezeichnet sie deshalb als *globale Compilierung*. Der Compiler berechnet eine Aufteilung der Ressourcen für alle beteiligten Anytime-Komponenten in einem einzigen Arbeitsschritt. Diese Vorgehensweise ist als Optimierungsproblem wie beschrieben NP-vollständig. Einen Ansatz, sie zu reduzieren, stellt die *lokale Compilierung* dar. Sie betrachtet im Gegensatz zur globalen Compilierung nur die direkten Subkomponenten eines Konstruktes. Unter Subkomponenten versteht man in diesem Zusammenhang Anytime-Algorithmen, die als Subprogramme benutzt werden. Im Beispiel des medizinischen Systems, das in Abschnitt 3.2.2 erläutert wurde, repräsentieren die beiden Algorithmen zur Diagnose und zur Bestimmung einer Behandlung solche Subkomponenten des Gesamtsystems. Graphisch dargestellt ergibt sich der einfache Baum aus Abbildung 3.10, bestehend aus einem Knoten für das Gesamtsystem, einem für die Diagnosekomponente und einem für die Komponente zur Bestimmung der Behandlung. Es handelt sich hier um eine sequentielle Verknüpfung beider Teile.

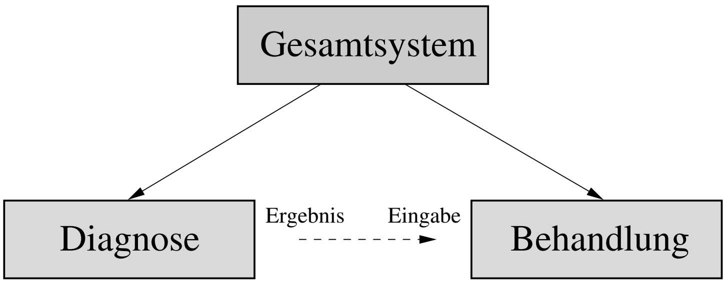

Abbildung 3.10: Beispiel des Aufbaus eines Anytime-Systems aus Subkomponenten

Die gesamte Aufgabe wird auf diese Weise in mehrere Teilaufgaben zerlegt, die mit weniger Aufwand zu bearbeiten sind. Allerdings kann nun nicht mehr in allen Fällen die Optimalität der so bestimmten Ressourcenverteilung gewährleistet werden.

Ein Beispiel für eine Situation, in der keine Optimalität erreicht wird, ist das wiederholte Auftreten von Subkomponenten, die die gleichen Aufgaben erledigen. Das bedeutet, daß bei einer Identifizierung dieser Komponenten, die Forderung nach einer Baumstruktur verletzt wird. Abbildung 3.11(a) zeigt ein solches Beispiel, Abbildung 3.11(b) zeigt das gleiche mit identifizierten Subkomponenten.

Bei der lokalen Compilierung werden allen Komponenten gewisse Ressourcen zugeteilt, wohingegen bei einer globalen Sichtweise nur einer Ressourcen zugewiesen werden und alle anderen auf das von ihr berechnete Ergebnis zurückgreifen (vgl. Abschnitt 3.2.7).

Zur effektiven Anwendung der lokalen Compilierung auf zusammengesetzte Module müssen drei Voraussetzungen erfüllt sein:

- **Baumstruktur:** Das zu compilierende Modul besitzt eine Baumstruktur, insbesondere dürfen keine Zyklen auftreten.

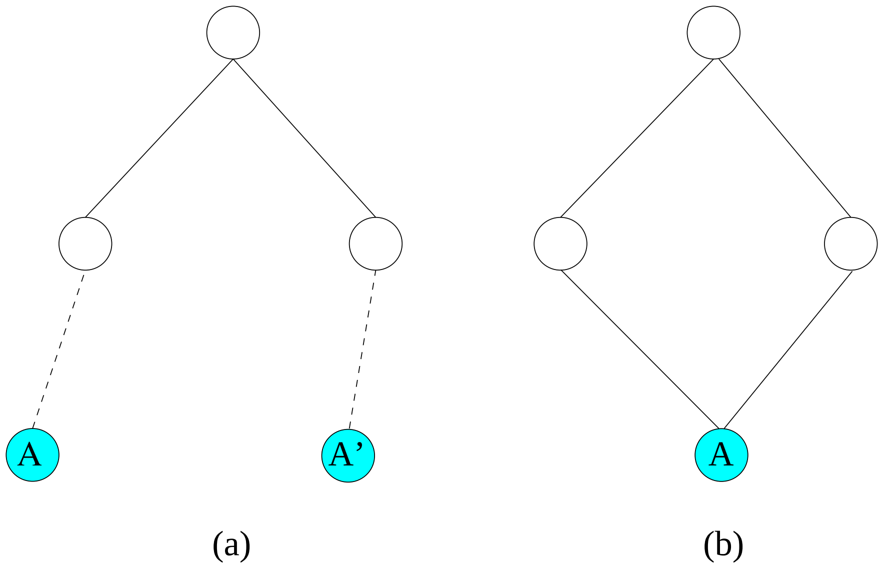

Abbildung 3.11: Mehrfach auftretende Subkomponenten

- **beschränkter Grad:** Die Anzahl der Eingaben in jede Komponente ist beschränkt durch eine (kleine) Konstante.

- **Monotonie der Eingabequalität:** Die Qualität der Ausgaben einer Anytime-Komponente erhöht sich, wenn die Qualität ihrer Eingaben sich erhöht.

Die erste Voraussetzung ermöglicht erst die oben beschriebene Vorgehensweise, denn in jedem Schritt werden die Nachfolgerknoten im Baum betrachtet. Existierte ein Zyklus, würde man in eine endlose Rekursion laufen. Die zweite garantiert die Effizienz der lokalen Compilierung. Die dritte beschreibt eine intuitive Annahme, die aber nicht in allen Fällen erfüllt ist. Mit diesen Einschränkungen läßt sich der folgende Satz formulieren:

Satz 3.2.1 (Optimalität der lokalen Compilierung) *Die lokale Compilierung eines beliebigen zusammengesetzten Moduls ist optimal, wenn das Modul eine Baumstruktur besitzt, der Grad beschränkt ist und die Monotonie der Eingabequalität erfüllt wird.*

Bezüglich der Komplexität der lokalen Compilierung zeigt Zilberstein folgenden Satz, der die praktische Anwendung rechtfertigt:

Satz 3.2.2 (Komplexität der lokalen Compilierung) *Besitzt das zusammengesetzte Modul eine Baumstruktur und gelten die Annahmen des beschränkten Grades und der Monotonie der Eingabequalität, dann läßt sich die lokale Compilierung in pseudo-polynomialer Zeit durchführen.*

Damit hat man mit der lokalen Compilierung eine approximative Vorgehensweise zur Lösung des NP-vollständigen Problems der Compilierung von Anytime-Algorithmen.

3.2.7 Mehrfaches Auftreten von Subkomponenten

Um das im vorigen Abschnitt angesprochene Problem der mehrfach auftretenden Subkomponenten beim Compilierungsprozeß zu berücksichtigen und eine unnötige Zuteilung von Ressourcen zu vermeiden, sind zusätzliche oder andere Vorgehensweisen zu den bisher vorgestellten notwendig. Zilberstein zeigt drei verschiedene Ansätze auf, die im folgenden erläutert werden. Voraussetzung bei allen Verfahren ist die Repräsentation von Performanzprofilen in diskreten Tabellen.

3.2.7.1 Hill-Climbing

Das erste Verfahren ist im Rahmen der vorliegenden Arbeit besonders interessant, weil es eine Art der Berechnung der Ressourcenverteilung ist, die auch losgelöst von der Compilierung nach Zilberstein anwendbar ist (vgl. Kapitel 4). Dies ist der Fall, weil es ein globaler Ansatz ist, also nicht die lokale Compilierung benutzt, und es somit die Ressourcen auf alle Komponenten verteilt ohne rekursive Berechnungen der einzelnen Konstrukte durchzuführen. Damit kann es jederzeit zur Laufzeit eines Systems angewendet werden, nicht nur bei der Konstruktion bzw. Compilierung, wie es bei (Zilberstein, 1993) vorgesehen ist.

`Hill-Climbing-Ressourcenverteilung`(t)
for each $q_{in} \in [q_u, \ldots, q_o]$ **do**
  $s \leftarrow$ `initiale_Größe`(t)
  $t_i \leftarrow t/n \; \forall i : 1 \leq i \leq n$
  repeat
    while $\exists i, j :$
    $E(q_{out}(q_{in}, t_1, \ldots, ti - s, \ldots, t_j + s, \ldots, t_n)) >$
    $E(q_{out}(q_{in}, t_1, \ldots, q_n))$ **do**
      seien i, j die Indizes, die die erwartete Qualität maximieren
      $t_i \leftarrow ti - s$
      $t_j \leftarrow tj + s$
    od
    $s \leftarrow s/2$
  until $s < \epsilon$
  $T[q_{in}] \leftarrow (t_1, \ldots, t_n)$
od

Abbildung 3.12: Hill-Climbing-Algorithmus zur Berechnung der Ressourcenverteilung

Der in Abbildung 3.12 dargestellte *Hill-Climbing-Algorithmus* berechnet für vorgegebene Gesamtressourcen t und für alle möglichen Eingabequalitäten q_{in} die zugehörige Verteilung. Er geht zu Beginn von einer Gleichverteilung der Ressourcen auf alle Komponenten aus. Danach wird eine Zeitspanne s zwischen zwei Komponenten derart umverteilt, daß ein möglichst großer Anstieg in der Gesamtqualität erzielt wird. Solange solche Umverteilungen möglich sind, werden sie durchgeführt. Ist dies nicht mehr der Fall, so findet eine Halbierung der Zeitspanne s statt und der

Vorgang wiederholt sich. Der Abbruch erfolgt, wenn eine untere Grenze ϵ erreicht wird. Damit ist ein bezüglich der Parameter lokales Maximum bestimmt, welches als Ergebnis zurückgeliefert wird. Wie bei allen Hill-Climbing-Algorithmen kann das Erzielen des bestmöglichen Ergebnisses nicht garantiert werden, da der Algorithmus in einem lokalen Extremum 'hängenbleiben' kann.

Die beiden folgenden Verfahren stellen eine Modifikation bzw. Erweiterung der lokalen Compilierung dar und sind somit im weiteren Verlauf dieser Arbeit nicht von Bedeutung. Deshalb werden nur die Ideen, die dahinter stehen, erläutert.

3.2.7.2 Konditionierung

Die *Konditionierung* basiert darauf, daß vor Beginn der lokalen Compilierung die Ressourcen für die mehrfach auftretenden Subkomponenten festgelegt werden. Die Ressourcen der anderen Komponenten werden wie üblich unter Festhalten der Zuteilung an die Subkomponenten durch die lokale Compilierung berechnet. Nacheinander werden alle möglichen Kombinationen dieser Zuteilungen bearbeitet. Die Lösung mit der besten Gesamtqualität liefert die gesuchte Ressourcenverteilung, bei der den mehrfach auftretenden Subkomponenten jeweils nur einmal Ressourcen zugeteilt werden. Das Vorgehen ist analog wie beim Verfahren der Konditionierung zur Berechnung der Lösung in mehrfach verbundenen Bayes'schen Netzen (Pearl, 1988).

3.2.7.3 Lernen

Das letzte von Zilberstein vorgeschlagene Verfahren ist ein Algorithmus, der die Zuteilung von Ressourcen an mehrfach auftretende Subkomponenten durch lokale Compilierung "lernt". Zu Beginn wird das Vorhandensein mehrerer Kopien von Subkomponenten ignoriert und die lokale Compilierung ohne Berücksichtigung dieser Problematik angewendet. Danach erfolgen Anpassung der Performanzprofile mit anschließender lokaler Compilierung. Diese Vorgehensweise endet mit einer einmaligen Zuweisung von Ressourcen an alle Subkomponenten.

Für nähere Erläuterungen und die expliziten Algorithmen der beiden letzten Verfahren wird auf (Zilberstein, 1993) und (Zilberstein & Russell, 1996) verwiesen.

Nach der Beschreibung der Compilierung nach Zilberstein folgt nun eine Betrachtung des Monitorings innerhalb des erzeugten ausführbaren Anytime-Moduls zur Laufzeit des konstruierten Systems.

3.2.8 Monitoring

Das Monitoring ist ein zentraler Aspekt im Zusammenhang mit der Compilierung von Anytime-Algorithmen. Es kontrolliert während der Laufzeit die Ressourcenzuteilung an die einzelnen Komponenten und ermöglicht somit das effektive Ablaufen des Gesamtsystems. Man unterscheidet zwei Arten des Monitorings:

Definition 3.2.1 (passives Monitoring) *Von* passivem Monitoring *spricht man, wenn alle Ressourcenzuteilungen komplett vor Start des Systems vorgenommen werden.*

Man kann das passive Monitoring als eine Form der vorgeplanten Kontrolle auffassen, deren Kontrollaktionen vor dem Ablauf der Berechnung bezüglich dem Zeitpunkt ihres Auftretens festgelegt werden.

Definition 3.2.2 (aktives Monitoring) *Von* aktivem Monitoring *spricht man, wenn es nicht passiv ist, das heißt wenn zur Laufzeit des Systems Veränderungen an der Ressourcenzuteilung vorgenommen werden.*

Der Hauptgrund, aktives Monitoring anzuwenden, ist das Problem der Unsicherheit. Wäre alles in Bezug auf das System absolut deterministisch (Performanzprofile, Umgebung, etc.), so würde eine Ressourcenverteilung vor Start des Systems genügen und es wären keine Anpassungen während des Laufs mehr nötig. In realen Anwendungen ist dies nicht der Fall. Es gibt zwei Arten der Unsicherheit in solchen Systemen:

1. **intern:** Im allgemeinen Fall kann die Qualität des Ergebnisses eines Anytime-Algorithmus bei gleicher Laufzeit unterschiedlich sein. Der Grund dafür liegt in der hohen Komplexität der Rechnerumgebung (andere Prozesse, Benutzer, etc.). Dies wird auch dadurch deutlich, daß Performanzprofile nur erwartete(!) Qualitäten repräsentieren. Deshalb kann unmöglich vor Start das genaue Verhalten des Systems vorhergesagt werden.

2. **extern:** Änderungen in der Umgebung des Systems sind ein weiterer Faktor. Es können Ereignisse eintreten, die bei weiterer Berechnung das zu erwartende Ergebnis unbrauchbar machen. Ein Beispiel hierfür ist der Planer eines Roboters, der immer noch am aktuellen Plan arbeitet, obwohl sich ein bewegliches Hindernis unerwartet in seinen Weg gestellt hat. Weiterhin ist das Modell, welches das System von seiner Umgebung besitzt, normalerweise nicht absolut identisch mit der Realität. Diese Approximation ist ein weiterer Unsicherheitsfaktor, der zum Tragen kommen kann.

Aus diesen Gründen arbeitet der Monitor zur Laufzeit des Systems mit einer *zeitabhängigen Nutzfunktion* $U_A(S, t, q)$ (vgl. (Russell & Wefald, 1991)), die für den zugehörigen Anytime-Algorithmus A die Werte der aktuellen Qualität q und Zeit t auf einen Wert abbilden, der den Nutzen für das System im vorliegenden Zustand S repräsentiert:

$$U_A : \Omega \times \mathbb{R}_{\geq 0} \times [0.0, 1.0] \longrightarrow \mathbb{R}. \tag{3.16}$$

(Ω ist die Menge der internen Zustände, die das System annehmen kann.)

Da diese Funktion sehr stark anwendungsabhängig ist, muß sie dem Monitor explizit vorgegeben werden.

Abbildung 3.13 zeigt einen typischen Verlauf einer zeitabhängigen Nutzfunktion im zweidimensionalen Fall, bei der die Qualität konstant bleibt (s. auch Abbildung 2.3). Ab einem gewissen Zeitpunkt gilt: Je mehr Zeit verstreicht, desto geringeren Nutzen liefert das zugehörige Ergebnis dem Gesamtsystem bis er auf dem Wert 0 angelangt ist und das Ergebnis damit nutzlos wird.

Im folgenden werden verschiedene Strategien des Monitorings vorgestellt, die je nach Eigenschaften des zu betreuenden Systems zu bevorzugen sind.

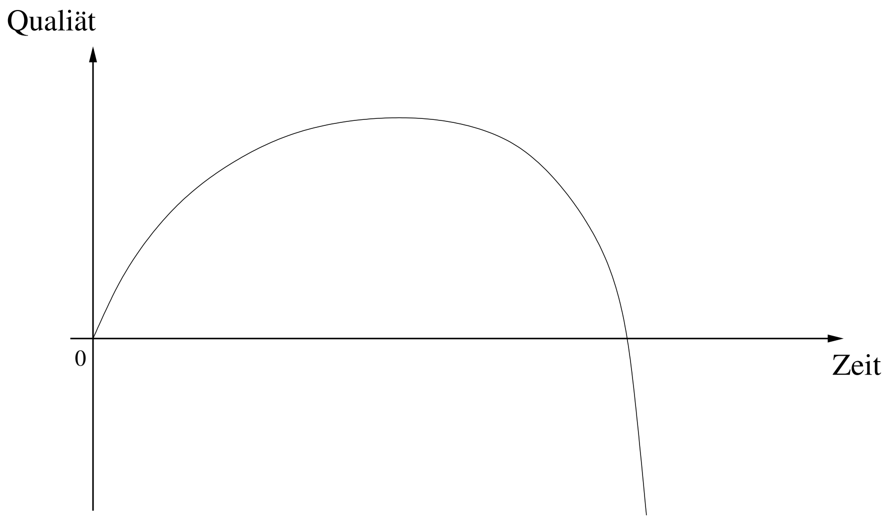

Abbildung 3.13: Beispiel einer zeitabhängigen Nutzfunktion

Der *fixed-contract Ansatz* basiert auf dem Prinzip, die bestmögliche Ressourcenzuteilung vor Aktivierung des Systems zu berechnen. Er stellt damit eine Art des passiven Monitorings dar. Diese Vorgehensweise eignet sich für Contract-Anytime-Systeme, deren Varianz der Performanzprofile relativ gering ist, d.h. die sich tatsächlich ergebenden Qualitäten unterscheiden sich im allgemeinen nur sehr gering von den erwarteten Werten.

Ist dies nicht der Fall, können zwei erweiterte Varianten dieses Ansatzes verwendet werden, die dann dem aktiven Monitoring zugerechnet werden müssen.

Die erste weist nicht genutzte Ressourcen, den noch nicht beendeten Komponenten neu zu. Im Falle von Contract-Anytime-Algorithmen ist der einzige Zeitpunkt, an dem die Qualität abgefragt werden kann, das Ende der zugesicherten Zeitspanne. Anhand dieser Qualitäten können mit Hilfe der bekannten Performanzprofile der anderen Komponenten die verbleibenden Ressourcen so neu verteilt werden, daß das bestmögliche Ergebnis unter der Berücksichtigung der bis zu diesem Zeitpunkt bekannten Zwischenergebnisse erreicht wird.

Die zweite Methode funktioniert ähnlich, der einzige Unterschied zur ersten besteht darin, daß nicht die verbleibende Restzeit neu verteilt wird, sondern Anpassungen an den ursprünglichen Ressourcenzuteilungen vorgenommen werden.

Ein Ansatz, der besonders für die Verwendung unterbrechbarer Anytime-Algorithmen geeignet ist, also für Systeme, die in Domänen arbeiten, die schnellen und großen Veränderungen unterliegen können, untersucht den erwarteten Zuwachs der zeitabhängigen Nutzfunktion nach einer (kleinen) Zeitspanne. Sobald dieser den Wert 0 erreicht, erfolgt keine weitere Zuteilung von Ressourcen an diese Komponente.

3.3 Zusammenfassung

Dieses Kapitel diente der ausführlichen Darstellung der für die vorliegende Arbeit wichtigen Konzepte und Begriffe. Dazu wurden die Arbeiten von Dean und Boddy bezüglich der Anytime-Algorithmen und von Zilberstein mit der Compilierung von Anytime-Algorithmen beschrieben. Erstere stellt die grundlegenden Werkzeuge und Vorgehensweisen zur Verfügung, um Probleme unter Beachtung zeitlicher Restriktionen lösen zu können. Aufbauend zeigt Zilberstein eine Technik der Konstruktion komplexer Anytime-Systeme, die allerdings den Nachteil hat, daß schon vor dem Start so entwickelter Systeme die zur Verfügung stehende Laufzeit bekannt sein muß. Eine Einflußnahme während des Laufs ist nur sehr eingeschränkt möglich.

Kapitel 4

Eine Anytime-System-Shell

Inhalt dieser Arbeit ist die Entwicklung eines Systems zum Ressourcenmanagement, das die Möglichkeit bereitstellt, beschränkt-optimale Realzeitsysteme zu konstruieren, die die Eigenschaften von Anytime-Algorithmen (s. Abschnitt 3.1.2) besitzen. Die Beschreibung dieses Systems ist Thema des vorliegenden Kapitels.

In den ersten Abschnitten wird die Konzeption dargestellt, um danach die Implementation ausführlich zu beschreiben.

4.1 Eigenschaften

Im folgenden werden die charakteristischen Merkmale und Besonderheiten des in dieser Arbeit entwickelten Systems vorgestellt. Insbesondere werden die Aspekte betrachtet, die sich von den in Kapitel 2 beschriebenen Ansätzen zur Konstruktion von Realzeitsystemen unterscheiden.

4.1.1 Unterbrechbarkeit

Die wichtigste Eigenschaft des in diesem Kapitel vorgestellten Systems ist die *jederzeitige Unterbrechbarkeit* der mit seiner Hilfe konstruierbaren Gesamtsysteme. Es werden keinerlei Einschränkungen bezüglich ihrer Laufzeit zum Startzeitpunkt gemacht. Hierin besteht der große Vorteil gegenüber den Ansätzen des Design-to-time-Schedulings (s. Abschnitt 2.4.3) und der Compilierung von Anytime-Algorithmen (s. Abschnitt 3.2), die voraussetzen, daß die dem Gesamtsystem zur Berechnung zur Verfügung stehende Zeit vor Beginn mitgeteilt wird, um vorher eine Ressourcenverteilung zu ermöglichen[1]. Da in vielen Anwendungen keine Informationen über den Zeitpunkt von Anfragen an das Gesamtsystem vorliegen, eignet sich für diese das vorgestellte Konzept besonders gut.

4.1.2 Unabhängigkeit vom Gastsystem

Um eine *universelle Einsetzbarkeit* des vorgestellten Systems zu gewährleisten, muß es vollständig von dem jeweils betrachteten *Gastsystem* abgetrennt sein. Unter dem

[1] abgesehen von zur Laufzeit durchgeführtem Monitoring

Gastsystem versteht man ein System, das von dem entwickelten System zum Ressourcenmanagement "betreut" wird, das bedeutet dessen zur Verfügung stehende Ressourcen so verwaltet werden, daß als Gesamtsystem ein Realzeitsystem entsteht. Interaktionen zwischen diesen beiden Komponenten werden über eine definierte Schnittstelle durchgeführt. Wegen dieser und der Anytime-Eigenschaften kann das entwickelte System als eine *Anytime-System-Shell* bezeichnet werden. Mit ihr ist es möglich, beliebige Anytime-Systeme zu entwickeln, solange bei der Konstruktion des Gastsystems bestimmte Anforderungen erfüllt werden, die in Abschnitt 4.3.1 erläutert werden.

4.1.3 Flexible Architektur

Durch die Möglichkeit, unterschiedliche Lösungsstrategien für Teilprobleme verwenden zu können, wird eine *flexible Architektur* der Gesamtsysteme bereitgestellt, die sehr einfach um zusätzliche Verfahren erweitert werden kann. Damit ist ein sukzessiver Ausbau des Gesamtsystems möglich, was eine große Erleichterung bei der Konstruktion solcher Realzeitsysteme darstellt.

Ein weiterer Aspekt dieser flexiblen Architektur ist die Möglichkeit, Subsysteme zu verwenden. Es können mehrere mit Hilfe der Anytime-System-Shell erstellte Realzeitkomponenten in der Architektur des Gesamtsystems zur Lösung von Teilproblemen benutzt werden (s. Abbildung 4.1).

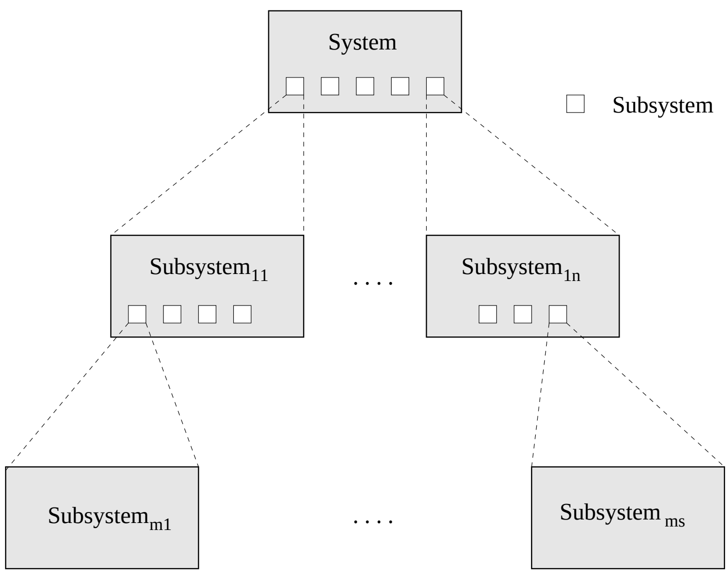

Abbildung 4.1: Flexible Architektur des Gesamtsystems

Somit kann eine Bibliothek von (Sub-)Systemen erstellt werden, auf die bei der Entwicklung neuer Anwendungen zurückgegriffen werden kann. So könnte beispielsweise ein zu einem früheren Zeitpunkt entwickeltes Objektlokalisationssystem innerhalb eines Navigationssystems als Subkomponente eingesetzt werden.

4.1.4 Transaktionen

Die Anytime-System-Shell ermöglicht die Verwendung von *Transaktionen* innerhalb des Gastsystems.

Definition 4.1.1 (Transaktion) *Unter einer* Transaktion *versteht man eine Folge von Berechnungsschritten, die, sobald der erste begonnen wurde, in ihrer Gesamtheit ohne Unterbrechung durchgeführt werden müssen.*

Dieses Konzept, das erstmals von Görz und Kesseler (Görz & Kesseler, 1994b, 1994a) im Zusammenhang mit dem Anytime-Begriff eingeführt wurde, stammt ursprünglich aus der Theorie der Datenbank- und Betriebssysteme (s. beispielsweise (Silberschatz & Galvin, 1994)).

Bei der Verwendung nebenläufiger Prozesse im Rahmen der Berechnung können auftretende Synchronisationsprobleme gelöst werden, indem der Übergang zwischen zwei stabilen internen Zuständen des Systems durch die Verwendung von Transaktionen realisiert wird (s. auch Abschnitt 4.3.3.4). Unter einem stabilen Zustand wird in diesem Zusammenhang ein Zustand verstanden, in dem die anderen Prozesse (weiter-)arbeiten können, ohne auf eventuell fehlerhafte (Zwischen-)Ergebnisse zurückgreifen zu müssen oder falsche Ergebnisse zu produzieren.

Als einfaches Beispiel zum Transaktionskonzept und der damit lösbaren Probleme kann eine Überweisung eines Geldbetrags von einem Konto auf ein anderes dienen. Der Betrag wird vom Kontostand des ersten abgezogen und zu dem des zweiten hinzuaddiert. Bevor nicht beide Aktionen komplett ausgeführt sind, darf der Vorgang nicht unterbrochen werden, denn sonst könnten Abfragen über den Kontostand Inkonsistenzen liefern. Tritt eine Unterbrechung zum Beispiel nach dem Abziehen des Betrags vom ersten Konto auf und werden dann die beiden Kontostände betrachtet, so "verschwindet" das zu überweisende Geld. Da aber die triviale Nebenbedingung gilt, daß die Gesamtgeldmenge der beiden an dem Vorgang beteiligten Konten sich zu keinem Zeitpunkt verändern darf, muß die Überweisung als Transaktion im obigen Sinne aufgefaßt werden und darf nur in ihrer Gesamtheit und ohne Unterbrechung durchgeführt werden.

4.1.5 Ressourcenadaptierende Metakontrolle

Eine weitere charakteristische Eigenschaft von mit Hilfe der Anytime-System-Shell konstruierten Gesamtsystemen ist ein *ressourcenadaptierendes Verhalten* wie in Abschnitt 2.2 erläutert. Durch Aufzeichnen der Performanz werden Informationen gesammelt, die in den weiteren Durchläufen benutzt werden können, um das Verhalten des Systems zu optimieren. Die so angesammelten Daten kann das System im Rahmen der Vorgehensweise bezüglich der Ressourcenverteilung verwenden. Dies geschieht innerhalb einer Komponente der Anytime-System-Shell, die eine Metakontrolle der Zuteilung von Ressourcen an einzelne Teilaufgaben darstellt. Da mit einer Zuteilung von unterschiedlichen Mengen an Ressourcen eine Priorisierung gewisser Lösungsstrategien erzielt wird, erhält man ein ressourcenadaptierendes Gesamtsystem. Zur Realisierung werden Techniken genutzt, die im Rahmen der Compilierung von Anytime-Algorithmen von Zilberstein vorgestellt wurden (s. Abschnitt 3.2.7). In Abschnitt 4.3.3 wird die Vorgehensweise hierzu näher beschrieben.

4.1.6 Plattformunabhängige Java-Werkbank

Als High-End-Benutzerschnittstelle für die Ein- und Ausgabe der Aufgabenstellung bzw. der erzielten Ergebnisse des Gastsystems dient die *plattformunabhängige Java-Werkbank*. Weiterhin können über sie die Parameter und interne Verwaltungsstrukturen der Anytime-System-Shell visualisiert und modifiziert werden. Somit steht mit der Java-Werkbank ein Tool als Benutzerinterface zur Verfügung, um die (internen) Abläufe des Gesamtsystems verfolgen und, falls noch keine oder nur sehr wenige Erkenntnisse über sein Verhalten vorliegen, gegebenenfalls optimieren zu können[2].

Den Abschluß dieses Abschnitts über die Eigenschaften der unter Verwendung der Anytime-System-Shell konstruierten Gesamtsysteme bildet eine Festlegung und Abgrenzung verschiedener Begriffe für die weiteren Abschnitte und Kapitel der vorliegenden Arbeit.

Definition 4.1.2 (Gastsystem) *Ein* Gastsystem *umfaßt die Lösungsstrategien und zugehörige Daten des betreuten Systems.*

Definition 4.1.3 (Subsystem) *Ein* Subsystem *bildet eine Teilmenge eines Gastsystems. Es umfaßt die Lösungsstrategien und Daten, die zur Bearbeitung einer bestimmten Teilaufgabe des Gastsystems benötigt werden.*

Definition 4.1.4 (Anytime-System-Shell) *Die* Anytime-System-Shell *stellt die zum Ressourcenmanagement notwendigen Werkzeuge zur Verfügung. Sie ist vollkommen unabhängig von irgendwelchen Gastsystemen.*

Definition 4.1.5 (Gesamtsystem) *Ein* Gesamtsystem *ergibt sich aus der Einbettung eines Gastsystems in die Anytime-System-Shell. Ein solches Gesamtsystem besitzt (durch die Werkzeuge der Anytime-System-Shell) die Fähigkeiten mit Ressourcenbeschränkungen umzugehen.*

Nach diesen einführenden Erörterungen wird in den nachfolgenden Abschnitten der Aufbau und die prinzipielle Arbeitsweise des Systems JAMES (Java Anytime Management & Editor System) zum Ressourcenmanagement in Anytime-Systemen vorgestellt.

4.2 Architektur eines Gesamtsystems

Die in Abbildung 4.2 dargestellte Architektur des Gesamtsystems ist geprägt von seiner rekursiven Struktur und der Abtrennung der vorgestellten Metakontrolle (Anytime-Kontrolle) vom Gastsystem.

Das Gesamtsystem besteht aus *Ressourcen-adaptierenden Modul-Instanzen (RAMI)*, die eine rekursive Baumstruktur bilden. Eine solche RAMI gliedert sich in zwei

[2]Später kann das System dann Informationen aus früheren Durchläufen nutzten und damit seine Performanz selbstständig optimieren.

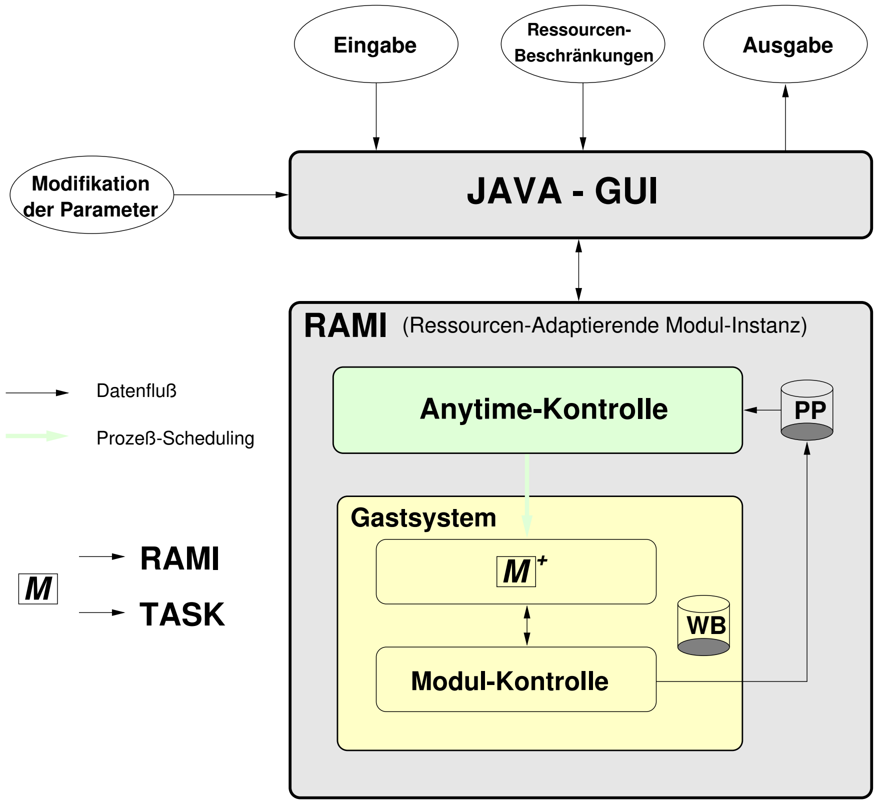

Abbildung 4.2: Architektur des Gesamtsystems mit Java-Werkbank

Komponenten: Die *Anytime-Kontrolle*, die für das Scheduling einzelner Berechnungen, im weiteren Verlauf als *Tasks* bezeichnet, zuständig ist, und das Gastsystem, das die eigentlichen Berechnungsstrategien bereitstellt.

Das Gastsystem teilt sich wiederum auf in eine Menge von *Modulen*, den zugehörige Datenbestand (WB) und die zugeordnete *Modulkontrolle*. Die Module repräsentieren entweder auf unterster Ebene einen Task oder eine RAMI als Subsystem zur Lösung eines bestimmten Teilproblems. Damit stellen die Tasks die Blätter der Baumstruktur dar. Ein Beispiel eines solchen Baumes zeigt Abbildung 4.3.

Die Modulkontrolle dient sowohl dem Austausch von Daten zwischen den einzelnen Modulen untereinander als auch zwischen der Gesamtheit der Module und der Anytime-Kontrolle. Letzteres bezieht sich in der Initialisierungsphase des Systems auf Konfigurationsdaten den internen Aufbau des Gastsystems betreffend und während der Berechnung auf Daten bezüglich der bisher erzielten Performanz unter Verwendung von Performanzprofilen. Anhand dieser Daten kann dann mittels der erlernten Informationen aus vorhergehenden Durchläufen die zukünftige Ressourcenzuteilung an die einzelnen Module bestimmt werden.

Auf oberster Ebene kommuniziert die Java-Werkbank, die der Ein- und Ausgabe, der Angabe von Ressourcenbeschränkungen, sowie der Anzeige und der Modifikation der Parameter dient, mit der obersten RAMI des Systems. Über diese Werkbank können insbesondere externe Start-, Unterbrechungs- und Stop-Signale an das Gesamtsystem abgesetzt werden.

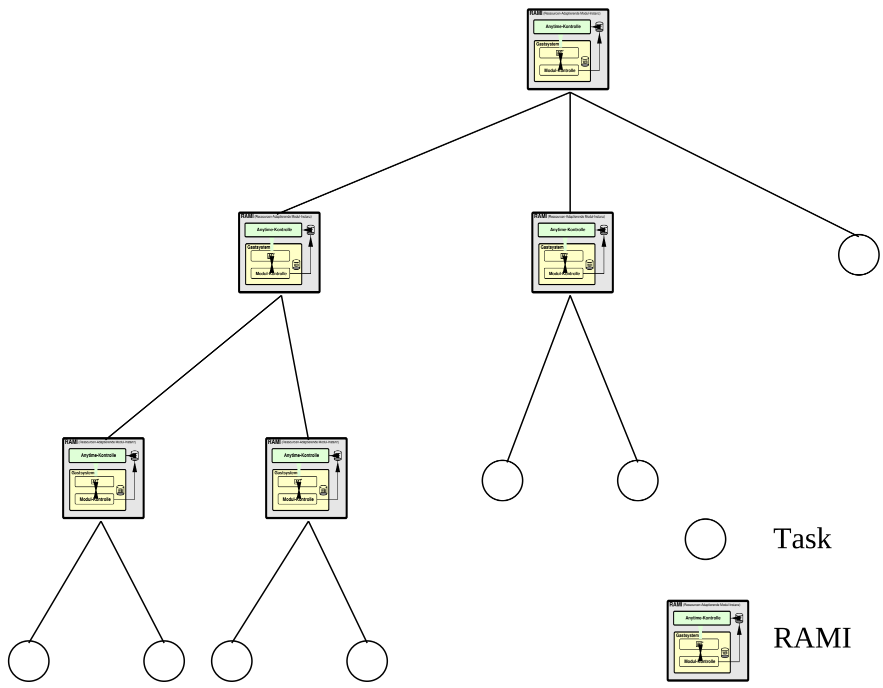

Abbildung 4.3: Baumstruktur des Gesamtsystems

4.3 RAMI

Den zentralen Begriff im Zusammenhang mit der Anytime-System-Shell stellt die RAMI dar. Ihr Aufbau und ihre Funktionsweise werden im folgenden ausführlich anhand der einzelnen Komponenten erläutert. Zunächst werden bestimmte Voraussetzungen formuliert, die das Gastsystem erfüllen muß.

4.3.1 Anforderungen an das Gastsystem

Das betreute Gastsystem muß für die Einbettung in die Anytime-System-Shell vorbereitet werden. Es muß die folgenden Anforderungen erfüllen, damit die zugehörigen Ressourcen verwaltet werden können.

4.3.1.1 Unterteilung in Module

Das Gastsystem muß in Teilaufgaben gegliedert werden. Die so entstehenden *Module* umfassen in ihrer Gesamtheit alle Berechnungsvorschriften, die zur Lösung der gestellten Probleme zur Verfügung stehen. Der restliche Programmcode dient lediglich administrativen Aufgaben, die im weiteren Verlauf dieses Kapitels erläutert werden.

Ein solches Modul kann und sollte mehrere unterschiedliche (approximative) Lösungsstrategien für eine Problemstellung beinhalten, um durch ihre nebenläufige Ausführung möglichst feine Verbesserungsschritte des Ergebnisses der Gesamtberechung zu bewirken. Es können sowohl Standard-Algorithmen, als auch Anytime-Algorithmen verwendet werden. Letztere optimieren das Anytime-Verhalten des Gesamtsystems. Soll zum Beispiel ein Modul der Lösung des Travelling-Salesman-Problems (s. beispielsweise (Cormen et al., 1990)) dienen, könnten eine approximative Anytime-Strategie und ein Algorithmus zur Bestimmung der exakten Lösung innerhalb des Moduls zur Verfügung gestellt werden. Solange die exakte Berechnung noch nicht beendet ist, würde das System auf die Näherungsergebnisse der approximativen Alternative zurückgreifen.

Eine weitere Möglichkeit ist, wie in Abschnitt 4.2 erwähnt, die Verwendung von eigenständigen Subsystemen zur Lösung von Teilaufgaben innerhalb eines Moduls.

4.3.1.2 Homogene hierarchische Systemstruktur

Die beschriebene Aufteilung des Gastsystems in Module und die zugehörigen Lösungsstrategien muß eine *homogene hierarchische Struktur* aufweisen: Es werden Teilprobleme des Gesamtsystems identifiziert und solange in weitere Subaufgaben untergliedert, bis eine Ebene erreicht wird, die elementare Lösungsstrategien oder Subsysteme bereitstellt. Dadurch wird ein Baum aufgespannt, bei dem sich alle Blätter auf der gleichen Ebene befinden. Eine einschränkende Forderung besteht darin, daß zwei Knoten einer Ebene sich bezüglich ihrer Nachfolgerknoten konzeptuell entsprechen müssen. Abbildung 4.4 zeigt hierfür ein Beispiel.

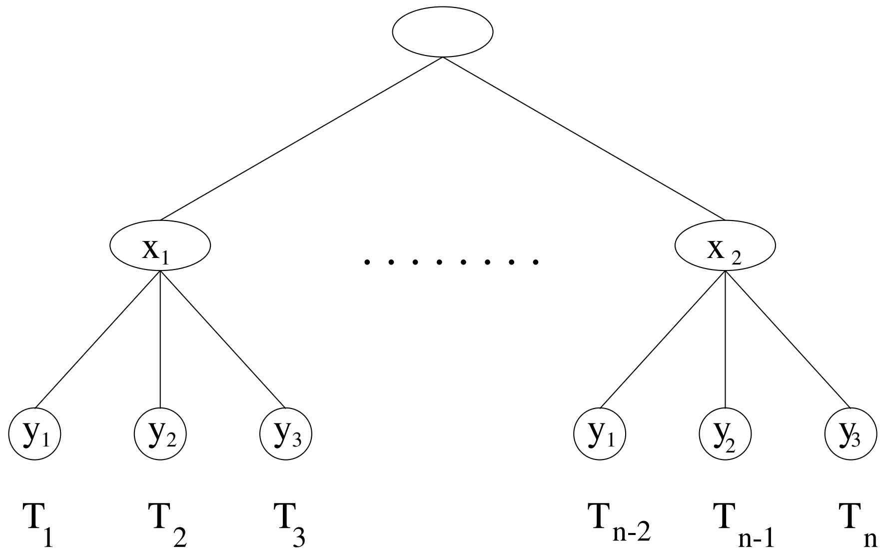

Abbildung 4.4: Homogene hierarchische Struktur des Gastsystems

Die Idee dabei ist, daß die Knoten des Pfades von der Wurzel zu einem Blatt die *Diskriminatoren* repräsentieren, die die vom Blatt dargestellte Lösungsstrategie determinieren. In der Abbildung besitzt der Tasks T_2 die Diskriminatoren x_1 und y_2 und führt das damit verbundene Verfahren durch. Im oben angeführten Beispiel des Travelling-Salesman-Problems könnte x_1 für die Entscheidung für das Anytime-Verfahrens stehen und y_2 könnten die explizite Auswahl einer Alternative bezüglich der dem Anytime-Algorithmus zur Verfügung stehenden unterschiedlichen Laufzeiten zur Verfeinerung des Performanzverhaltens repräsentieren. Insgesamt stellt die Menge der Blätter alle Kombinationen der möglichen Diskriminatoren und die dadurch festgelegten Lösungsverfahren dar.

4.3.1.3 Datenaustausch über Modulkontrolle

Der Austausch sämtlicher Ergebnisse und Hilfsdaten der einzelnen Module erfolgt über die Modulkontrolle. Dies bedeutet, daß immer wenn eine Teilaufgabe der zum Modul gehörenden Berechnung beendet wird, die erzielten Resultate dort registriert werden müssen. Auf diese Art können auch Abarbeitungsreihenfolgen festgelegt werden. Kann ein benötigtes Ergebnis eines anderen Moduls nicht von der Modulkontrolle erhalten werden, so wird die Berechnung des entsprechenden Moduls angestoßen.

Eine genauere Beschreibung der Modulkontrolle und ihrer Aufgaben folgt im nächsten Abschnitt.

4.3.2 Modulkontrolle

Die Modulkontrolle einer RAMI ist eine Datenstruktur, die den kompletten Datenfluß zwischen den einzelnen Teilkomponenten (Gastsystem, Anytime-Kontrolle) und, wie im vorigen Abschnitt erläutert, zwischen den verschiedenen Modulen des Gastsystems koordiniert. Ihre Aufgaben werden im folgenden im einzelnen untersucht. Dabei wird das Augenmerk auf die Aspekte gelenkt, die nicht im vorigen Abschnitt beschrieben wurden.

4.3.2.1 Übergabe der Konfigurationsdaten

In der Initialisierungsphase nach dem Starten des Systems besteht die Hauptaufgabe der Modulkontrolle in der Übergabe von Konfigurationsdaten an die Anytime-Kontrolle. Diese Daten beinhalten Informationen über die innere Struktur des Gastsystems (s. Abschnitt 4.3.1), insbesondere wird die in Abschnitt 4.3.1.2 beschriebene homogene hierarchische Systemstruktur zur Verfügung gestellt. Weiterhin werden zu diesem Zeitpunkt von der Modulkontrolle auch Angaben über eventuell vorhandene Subsysteme geliefert. Diese Daten benötigt die Anytime-Kontrolle, um ihre internen Strukturen zum Ressourcenmanagement aufzubauen. Wie diese aussehen und wozu sie im einzelnen dienen wird in Abschnitt 4.3.3 dargestellt.

4.3.2.2 Festlegung der Transaktionen

Die Festlegung der Transaktionen gehört zwar zu den Konfigurationsdaten, wird aber wegen der wichtigen Rolle der Transaktionen im Konzept der Anytime-System-Shell gesondert betrachtet.

Durch Registrieren der Transaktionen bei der Modulkontrolle kann das Gastsystem der Anytime-Kontrolle mitteilen, welche Programmsegmente ohne Unterbrechung der Berechnung durchgeführt werden müssen. Damit ist es dem Gastsystem bzw. dessen Entwickler möglich, flexibel zu entscheiden, was als Transaktion zu betrachten ist. Es gibt lediglich eine Einschränkung der Anwendung dieses Konzepts: Es wird gefordert, keine geschachtelten Transaktionen zu definieren.

Definition 4.3.1 (geschachtelte Transaktion) *Eine Transaktion heißt* geschachtelt*, wenn sie innerhalb einer anderen Transaktion beginnt.*

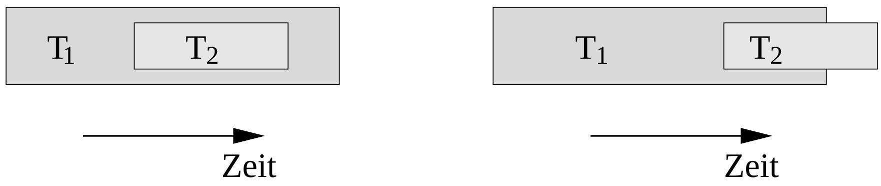

Abbildung 4.5: Geschachtelte Transaktionen

Abbildung 4.5 zeigt zwei Beispiele geschachtelter Transaktionen.

Diese Einschränkung ist notwendig, um in der Lage zu sein, die Beendigung einer Transaktion eindeutig festzustellen. Folgen zwei Transaktionen direkt nacheinander, so muß eine (eventuell bereits verzögerte) Unterbrechung durchführbar sein (s. Abbildung 4.6).

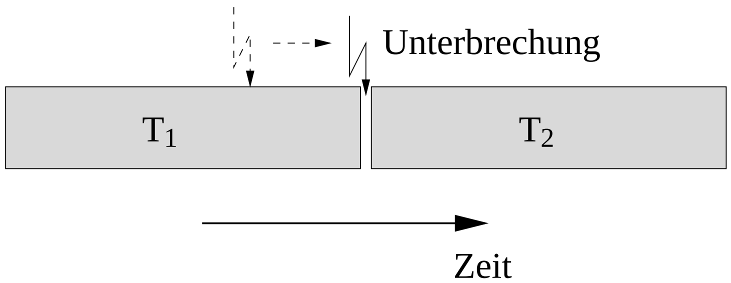

Abbildung 4.6: Direkt aufeinanderfolgende Transaktionen

Würde nicht unterbrochen, so müßte zunächst die anschließende Transaktion in Gänze abgearbeitet werden. Deshalb ist es essentiell, das Ende von Transaktionen eindeutig feststellen zu können. Im Fall von geschachtelten Transaktionen ist dies nicht immer möglich. Es kann zwar eine Transaktion beendet werden, aber die umschließende könnte noch nicht komplett durchgeführt sein. Im wesentlichen handelt es sich hierbei um ein Problem der Implementation. In der Anytime-System-Shell wurden deshalb geschachtelte Transaktionen ausgeschlossen, um diese Probleme zu vermeiden (siehe auch 4.5.2.7).

4.3.2.3 Bereitstellen des aktuellen Ergebnisses

Eine weitere wichtige Aufgabe der Modulkontrolle ist die Bereitstellung des jeweils aktuellen Zwischenergebnisses des Gastsystems zur Beantwortung von Anfragen bei Unterbrechungen des Gesamtsystems. Die Fähigkeit, jederzeit ein approximatives Zwischenergebnis liefern zu können, ist eine der Eigenschaften, die den Anytime-Charakter des Systems ausmachen (vgl. Abschnitt 3.1.2). Üblicherweise sollte nicht nur das Ergebnis zur Verfügung stehen, sondern zusätzlich auch die zugehörige aktuelle Qualität und die zu seiner Berechnung benötigten Ressourcen. Bei rekursiven Systemstrukturen (mit Subsystemen) interessiert den Benutzer im allgemeinen nur das aktuelle Ergebnis der Modulkontrolle der obersten RAMI. Untergeordnete Zwischenergebnisse werden häufig nur im Rahmen der Fehlersuche in der Entwicklungsphase des Systems genutzt (abgesehen von Zugriffen übergeordneter RAMIs innerhalb deren Berechnungen).

4.3.2.4 Aufzeichnen der Performanzdaten

Das Mitprotokollieren der Performanzdaten des zur betrachteten Modulkontrolle gehörigen Gastsystems dient der Erstellung von Performanzprofilen. Dazu werden die Zeitpunkte einer Qualitätsänderung zusammen mit dem zugehörigen Qualitätswert abgespeichert. Diese Wertepaare dienen als Stützpunkte des Performanzprofils.

Die so ermittelten Performanzprofile werden dann mit denjenigen von eventuell früher stattgefunden Läufen des Systems bei ähnlicher Aufgabenstellung kombiniert und in einer Datenbasis abgelegt. Danach können sie zur Vorhersage der Performanz und den damit verbundenen Metaentscheidungen der Anytime-Kontrolle verwendet werden (s. Abschnitt 4.3.3.2).

Eine zweite Verwendung ist die Möglichkeit der Anzeige der Performanzprofile mittels der Java-Werkbank. Damit können dann Rückschlüsse über das Verhalten des Systems gezogen und gegebenenfalls Tuning-Maßnahmen in den entsprechenden Modulen vorgenommen werden.

Die vorausgehenden Ausführungen zeigen, daß die Modulkontrolle die zentrale Koordinierungskomponente ist. Sie bildet die Schnittstelle zwischen dem betreuten Gastsystem und der Anytime-Kontrolle. Zusätzlich dient sie dem Protokollieren wichtiger statistischer Daten.

Im nächsten Abschnitt wird mit der für das Ressourcenmanagement zuständigen Komponente ein weiterer wichtiger Teil der Anytime-System-Shell beschrieben.

4.3.3 Anytime-Kontrolle

Dieser Abschnitt dient der Erläuterung der Systemkomponente, die für die Verwaltung der Ressourcen und die Kontrolle der Tasks zuständig ist. Als verwendbare Ressourcen stehen dem System Zeit und eine vom Benutzer oder Entwickler festgelegte Anzahl von Prozessen, die unter den Tasks verteilt und nebenläufig genutzt werden können, zur Verfügung.

Zunächst wird die prinzipielle Vorgehensweise der Anytime-Kontrolle beschrieben, um danach näher auf die einzelnen Aspekte einzugehen. Abschließend wird eine alternative Verfahrensweise vorgestellt, die bezüglich des Laufzeitverhaltens besser ist, aber ein schlechteres Anytime-Verhalten des Gesamtsystems bewirkt.

4.3.3.1 Prinzipielle Vorgehensweise

Mit Hilfe der vom Gastsystem bei der Modulkontrolle abgelegten Konfigurationsdaten werden zu Beginn eines Systemlaufs in einer Initialisierungsphase interne Strukturen zur Verwaltung der Ressourcen erzeugt. Es wird eine Datenstruktur angelegt, die den Aufbau des Gastsystems repräsentiert, und mit deren Hilfe die Zeit und die Prozesse unter den einzelnen Tasks verteilt werden. Dazu werden Vorgehensweisen von Zilberstein verwendet. Das Scheduling der Tasks geschieht in einer dem *Round-Robin-Verfahren* aus dem Bereich der Betriebssysteme (s. (Silberschatz & Galvin, 1994)) ähnlichen Strategie, die einzelne zeitliche *Zyklen* unter den Tasks aufteilt. Der Unterschied besteht darin, daß nicht alle Tasks eine gleich große Zeitspanne zugeteilt bekommen. Nach jedem Zyklus werden eventuelle Veränderungen der Datenstruktur vorgenommen, die z.B. durch die Beendigung von Tasks bedingt sein können. Danach erfolgt eine Neuverteilung der zeitlichen Ressourcen des nächsten Zyklus und eine neue Zuteilung von frei gewordenen Prozessen. Dieser auch in Abbildung 4.7 dargestellte Ablauf wiederholt sich solange bis entweder alle Berechnungsschritte abgearbeitet sind oder ein externes Stop-Signal erteilt wird. Ein Unterbrechungssignal fordert das bis zu diesem Zeitpunkt ermittelte Zwischergebnis an. Die einzelnen Schritte diese Vorgangs werden im folgenden ausführlich beschrieben.

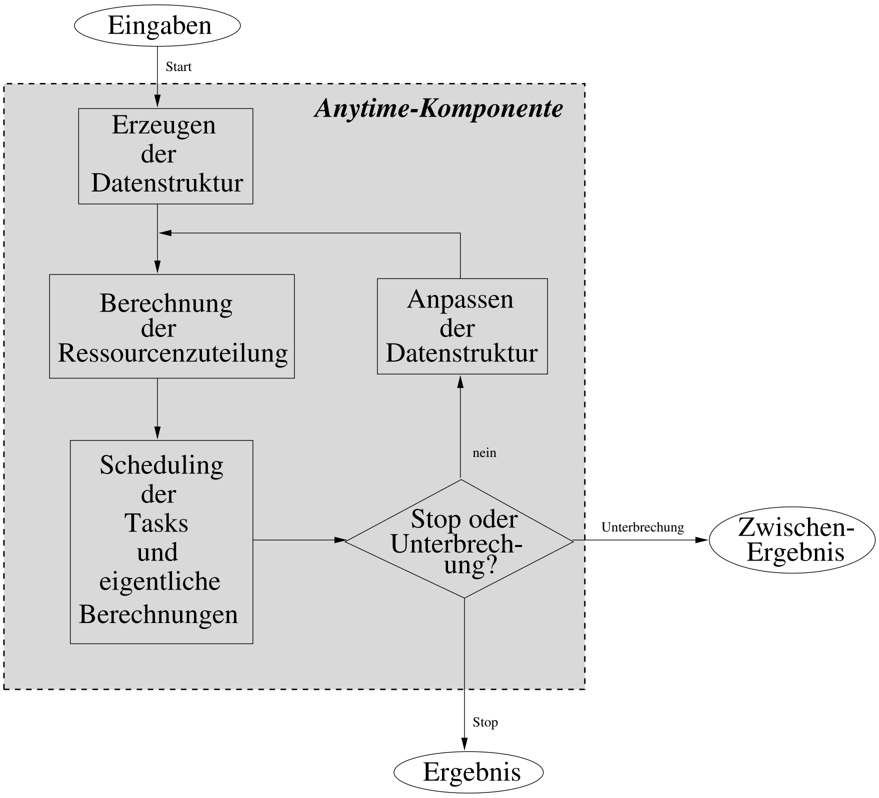

Abbildung 4.7: Prinzipieller Ablauf der Anytime-Kontrolle

4.3.3.2 Lokaler Ressourcenbaum zur Verwaltung der Tasks

4.3.3.2.1 Tasks

Jede Anytime-Kontrolle einer RAMI verwaltet einen *lokalen Ressourcenbaum*, der die in Abschnitt 4.3.1.2 eingeführte homogene hierarchische Systemstruktur des zugehörigen Gastsystems repräsentiert. Er wird zu Beginn eines Systemlaufs anhand der Daten der Modulkontrolle aufgebaut und besteht aus *virtuellen Tasks* als inneren Knoten und *realen Tasks* als Blätter. Die virtuellen Tasks symbolisieren abstrahierte Teilaufgaben des Gastsystems, die jedoch noch weiter untergliedert werden können. Die realen Tasks stehen für echte Berechnungen, das heißt für solche, die auch durchgeführt werden und somit die zur Verfügung stehenden Prozesse nutzen. Reale Tasks können, müssen aber nicht, eine oder mehrere Transaktionen beinhalten. Im weiteren Verlauf wird auf den Zusatz 'real' in diesem Zusammenhang verzichtet und nur von Tasks gesprochen. Weiterhin muß man unterscheiden zwischen Tasks, denen ein Prozeß zugeordnet ist, und solchen, die keinen besitzen.

Definition 4.3.2 (lauffähiger Task) *Ist einem realen Task ein Prozeß zugeordnet, so spricht man von einem* lauffähigen Task.

Definition 4.3.3 (nichtlauffähiger Task) *Ist einem realen Task kein Prozeß zugeordnet, so spricht man von einem* nichtlauffähigen Task.

Lauffähige Tasks sind diejenigen, die vom Scheduler kontrolliert werden und somit ihre Berechnungen durchführen können. Nichtlauffähige Tasks befinden sich im Wartezustand bis sie einen frei gewordenen Prozeß zugeteilt bekommen. Weitere Taskzustände werden durch die beiden folgenden Definitionen eingeführt:

Definition 4.3.4 (laufender Task) *Ist einem lauffähigen Task CPU-Zugriff zugeordnet, so spricht man von einem* laufenden Task.

und

Definition 4.3.5 (beendeter Task) *Hat ein Task seine ihm zugehörigen Berechnungen zur Gänze durchgeführt, dann spricht man von einem* beendeten Task.

Die verschiedenen Zustände, die ein Task annehmen kann, werden in Abbildung 4.8 aufgezeigt.

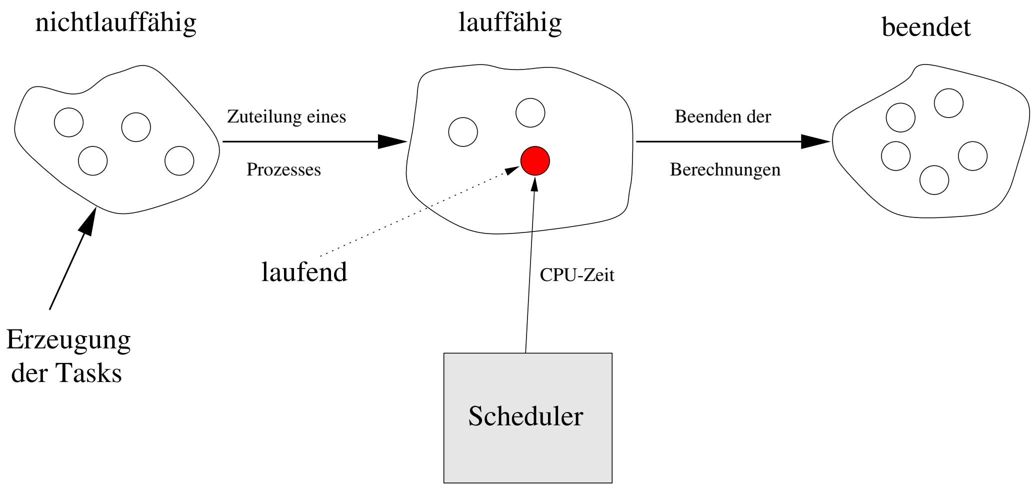

Abbildung 4.8: Zustände der Tasks

Nach seiner Erzeugung befindet sich ein Task im nichtlauffähigen Zustand. Bekommt er im Rahmen der Ressourcenverteilung einen Prozeß zugewiesen, geht er in den Zustand "lauffähig" über und wird in die Liste der Prozesse, die vom Scheduler der Anytime-Kontrolle bearbeitet werden, eingeordnet. Wird ihm dann vom Scheduler CPU-Zugriff zugewiesen, so wird er "laufend". Sind seine Berechnungen zur

Gänze durchgeführt, dann ist er "beendet".

Die Ressourcenverteilung geschieht in zwei Phasen. In der ersten findet eine Vorbereitung statt, die dann in der zweiten Phase die eigentliche Verteilung der Ressourcen ermöglicht. Zur Vorbereitung zwei Vorgehensweisen im System integriert, die in den nächsten beiden Abschnitten erläutert werden. Die erste basiert auf der Vorgabe bestimmter Parameter durch den Entwickler und kann in einer Lernphase des Systems eingesetzt werden, in deren Verlauf Erfahrungen in Form von Performanzprofilen gesammelt werden, die dann als Grundlage des zweiten Verfahrens dienen können. An dieser Stelle werden Techniken verwendet, die von Zilberstein vorgestellt wurden (s. Abschnitt 3.2.7). Diese Strategie verleiht dem Gesamtsystem dann das ressourcenadaptierende Verhalten.

4.3.3.2.2 Vorgabe von relativen Gewichten

Die erste Alternative zur Bestimmung der Ressourcenverteilung basiert auf der Zuordnung von *relativen Gewichten* zu jeder Kante des lokalen Ressourcenbaumes. Diese legen eine Priorisierung der (virtuellen) Tasks jeder Ebene des Baumes untereinander fest. Betrachtet man zwei Nachbarknoten K_1 und K_2 mit den relativen Gewichten g_1 und g_2 der in sie einlaufenden Kanten, so werden die für diese beiden Knoten verfügbaren Ressourcen im Verhältnis $g_1 : g_2$ auf sie verteilt. Da echte Berechnungen nur auf Blattebene stattfinden, muß die Ressourcenverteilung dort vorgenommen werden. Zeit und Prozesse können nur von diesen Tasks genutzt werden. Das bedeutet, daß die verfügbaren Prozesse und die vorhandene Zykluszeit unter Berücksichtigung der relativen Gewichte aller Vorgängerknoten auf sie verteilt werden müssen. Deshalb bekommen diese Tasks ein *globales Gewicht* zugewiesen, das sich folgendermaßen bestimmt: Seien $p_1, \ldots, p_{h-1}$ die Vorgängerknoten eines Tasks T im Pfad von der Wurzel innerhalb eines lokalen Ressourcenbaumes der Höhe h, dann ergibt sich das globale Gewicht g_{glob} von T als das Produkt der relativen Gewichte der Vorgängerknoten und dem relativen Gewicht von T:

$$g_{glob}(T) = g(T) * \prod_{i=1}^{h-1} g(p_i). \tag{4.1}$$

Ein Beispiel hierzu zeigt Abbildung 4.9. Die Werte unter den Blättern sind die zugehörigen globalen Gewichte.

4.3.3.2.3 Verwendung einer Datenbasis von Performanzprofilen

Die zweite Alternative der Ressourcenverteilung nutzt eine Datenbasis, die Performanzprofile vorhergehender Durchläufe des Systems beinhaltet. Nach jeder Beendigung einer Aufgabenstellung werden dazu die Werte aus den aufgezeichneten Performanzdaten der Modulkontrollen in die Performanzprofile der Datenbasis eingebracht.

Definition 4.3.6 (akquiriertes Performanzprofil) *Die Performanzprofile der Datenbasis eines unter Verwendung der Anytime-System-Shell konstruierten Gesamtsystems heißen* akquirierte Performanzprofile.

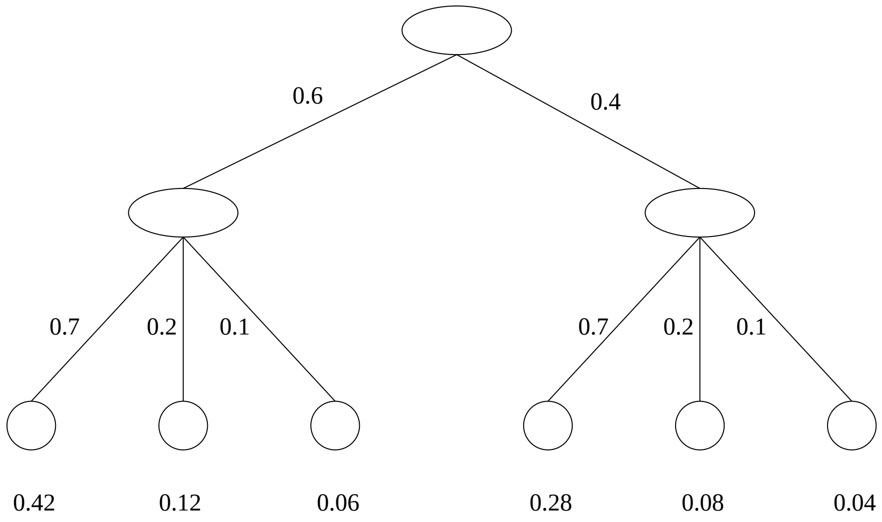

Abbildung 4.9: Beispiel zu globalen Gewichten von Tasks

Das konstruierte Gesamtsystem kann dann im Rahmen seiner Ressourcenzuteilung auf Erfahrungen früherer Bearbeitungen ähnlicher Problemstellungen zurückgreifen und sein (zukünftiges) Verhalten optimieren. Unter Ausnutzung der akquirierten Performanzprofile und der bisher erzielten Performanz, die aus der Modulkontrolle ermittelt wird, werden die globalen Gewichte der lauffähigen Tasks bestimmt.

Der in Abbildung 4.10 dargestellte Ablauf zur Bestimmung der Ressourcenverteilung findet vor jedem Zyklus statt und wird im folgenden erläutert.

	`Ressourcenverteilung()`
`(1)`	Überprüfung, ob alle benötigten Performanzprofile in der Datenbasis enthalten sind
`(2)`	Erstellen der projezierten Performanzprofile für die Tasks
`(3)`	Algorithmus zur Bestimmung der globalen Gewichte der lauffähigen Tasks

Abbildung 4.10: Algorithmus zur Bestimmung der globalen Gewichte

Zu Beginn wird überprüft, ob alle benötigten Performanzprofile in der Datenbasis vorhanden sind. Ist dies nicht der Fall, so kann diese Vorgehensweise der Ressourcenverteilung nicht angewendet werden und es wird auf die vorher erläuterte Verfahrensweise der Vergabe von relativen Gewichten zurückgegriffen.

Um die Ressourcenverteilung vornehmen zu können, muß ein *projeziertes Performanzprofil* bestimmt werden, das Informationen über das erwartete zukünftige Verhalten der einzelnen Tasks beinhaltet.

Definition 4.3.7 (projeziertes Performanzprofil) *Ein* projeziertes Performanzprofil *repräsentiert Informationen über das erwartete zukünftige Verhalten eines Task vom aktuellen Zeitpunkt an.*

Dabei kann es sich sowohl um virtuelle als auch um reale Tasks handeln. Im ersten Fall wird das beschriebene Verfahren auf einer festen inneren Ebene des lokalen Ressourcenbaumes durchgeführt und danach anhand der relativen Gewichte der Nachfolgeknoten, wie im alternativen Verfahren beschrieben, die globalen Gewichte der Blätter der einzelnen Teilbäume bestimmt. Im verbleibenden Teil dieses Abschnitts werden sämtliche mit Hilfe dieses Verfahrens ermittelten Werte eines Tasks als globale Gewichte bezeichnet. Dies gilt auch für den Fall, daß es sich bei den betrachteten Tasks um virtuelle handelt. Diese Festlegung ist sinnvoll, da diese Gewichte unabhängig von den übergeordneten Tasks sind.

Da in der Modulkontrolle nur die Performanz bis zum aktuellen Zeitpunkt aufgezeichnet ist, muß das Verhalten eines Tasks ab diesem Zeitpunkt geschätzt werden. Dies geschieht unter Verwendung dieser Werte aus der Modulkontrolle unter Zuhilfenahme der akquirierten Performanzprofile.

	`projeziertes_PP(`PP, t, q`)`
(1)	`neuer_Stützpunkt()`
(2)	`skaliere(`$PP, \frac{t}{x_1}$`)`
(3)	`verschiebe(`PP`)`
(4)	**return** (PP)

Abbildung 4.11: Algorithmus zur Bestimmung des projezierten Performanzprofils

Die Bestimmung der projezierten Performanzprofile ist in Abbildung 4.11 aufgezeigt. Als Eingaben erhält der Algorithmus das Performanzprofil aus der Datenbasis, die bisher benutzten Ressourcen t und die aktuelle Qualität q der Ergebnisses des Tasks, für den das projezierte Performanzprofil erstellt werden soll. Zuerst wird ein neuer Stützpunkt generiert, der als Ordinate die aktuelle Qualität besitzt (s. Abbildung 4.12). Danach wird das Performanzprofil so abgeschnitten, daß nur noch

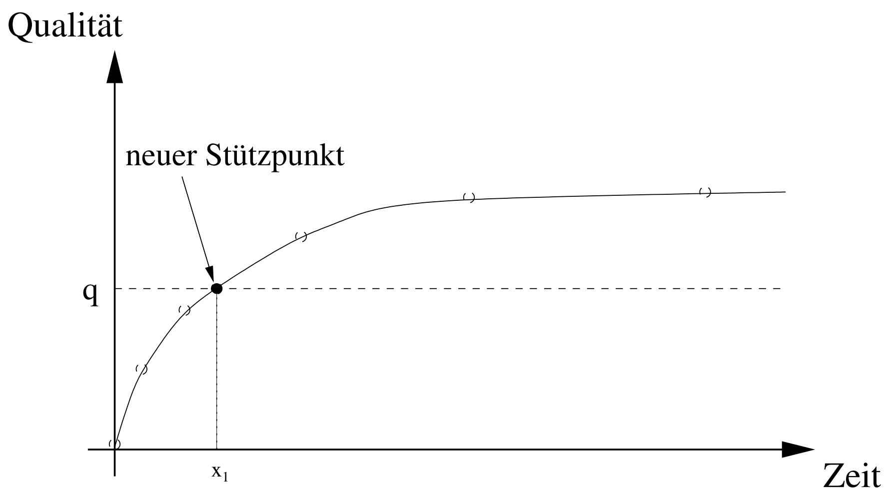

Abbildung 4.12: Bestimmung des neuen Stützpunktes des projezierten Performanzprofils

Qualitätswerte existieren, die über der aktuellen Qualität liegen. Anschließend wird das restliche Performanzprofil mit dem Faktor $\frac{t}{x_1}$ skaliert, wobei x_1 der Zeitwert des neuen Stützpunkts ist, und so verschoben, daß der neue Stützpunkt sich auf der Ordinate befindet. Diese Skalierung bringt das akquirierte zeitliche Verhalten der Performanz (aus der Datenbasis) unter Berücksichtigung des aktuellen (aus der Modulkontrolle) in das projezierte Performanzprofil ein. Die Verschiebung ist notwendig, um die verschiedenen projezierten Performanzprofile der einzelnen Tasks vergleichbar zu machen und die Anwendung im Algorithmus zur Bestimmung der globalen Gewichte zu ermöglichen. Auf diese Weise erhält man Performanzprofile, die Aussagen über das erwartete zukünftige Verhalten der Performanz des zugehörigen Tasks machen.

Anhand dieser projezierten Performanzprofile und, im Falle von virtuellen Tasks, der anschließenden Verteilung in den Teilbäumen werden die globalen Gewichte der lauffähigen Tasks errechnet, die zur Ressourcenzuteilung im anstehenden Zyklus verwendet werden.

Zu der Bestimmung dieser globalen Gewichte werden die Verfahren des Systems ORCAN (Operational Rationality through Compilation of ANytime-Algorithms) (Beckert & Baus, 1998) genutzt. Es stellt alternative Verfahren zur Verfügung, um anhand einer Menge von (projezierten) Performanzprofilen die Verteilung der vorgegebenen Gesamtressourcen auf die zugehörigen Anytime-Algorithmen zu bestimmen. Unter anderem existieren ein Verfahren der *linearen Regression* und ein *Hill-Climbing-Algorithmus*, der den in Abschnitt 3.2.7.1 vorgestellten Hill-Climbing-Algorithmus von Zilberstein abwandelt und anschließend vorgestellt wird.

Der Algorithmus beginnt mit einer Gleichverteilung der Ressourcen des nächsten Zyklus unter allen betrachteten Tasks. Anschließend wird das Performanzprofil gesucht, das bei einer Zuteilung einer als Parameter festzulegenden kleinen zusätzlichen Ressourcenmenge die erwartete Qualität maximiert. Der zugehörige Task bekommt soviele Ressourcen hinzu, daß gerade kein Abfallen der Qualität auftritt. Den anderen werden diese Ressourcen unter Berücksichtigung ihrer prozentualen Anteile am Gesamtressourcenvolumen abgezogen. Danach wird das nächste Performanzprofil mit der obigen Eigenschaft bestimmt. Dieser Vorgang wird solange wiederholt, bis ein lokales Maximum erreicht ist und somit kein Performanzprofil mehr existiert, das bei Zugabe von Ressourcen die erwartete Gesamtqualität verbessert. Die auf diese Weise bestimmten Werte stellen dann die globalen Gewichte der Tasks dar.

Um möglichst genaue Informationen aus der Datenbasis ermitteln zu können, werden nach jedem Systemlauf die neu erzielten Performanzdaten mit den abgespeicherten kombiniert und so in die Datenbasis eingebracht.

In einem ersten Schritt wird aus den in der Modulkontrolle mitprotokollierten Performanzdaten eines Tasks ein *lokales Performanzprofil* generiert.

Definition 4.3.8 (lokales Performanzprofil) *Ein* lokales Performanzprofil *repräsentiert des Verhalten eines Tasks der Anytime-System-Shell ohne die Einflüsse der übrigen.*

Die Zeitskala eines Profils aus der Modulkontrolle erstreckt sich über den kompletten Bereich von Beginn des Tasks bis zu seinem Ende inklusive der zwischenzeitlich stattfindenden Rechenphasen der anderen Tasks. Um eine "Bereinigung" der Profile durchzuführen, werden die Aktivierungs- und Deaktivierungszeitpunkte jedes Tasks mitprotokolliert. Damit ist es möglich, die Rechenphasen anderer Tasks zu eliminieren. Die so bestimmten lokalen Performanzprofile werden beim weiteren Vorgehen verwendet.

Jedem akquirierten Performanzprofil ist ein Zähler *count* zugeordnet, der die Anzahl der Durchläufe repräsentiert, deren Performanz eingeflossen sind. Dieser Wert wird benötigt, um die Auswirkungen eines Durchlaufs in Relation zu deren Gesamtanzahl zu setzen. Es darf nicht der Fall auftreten, daß ein einziger "Ausreißer" die Informationen aus vielen ähnlich verlaufenen Berechnungen übermäßig beeinflußt. Im folgenden wird dieses Verfahren zur Kombination des aktuellen und des akquirierten Performanzprofils der Datenbasis beschrieben. Es müssen zwei Aspekte berücksichtigt werden: Die eventuell variierende Dauer eines Tasks und das unterschiedliche Verhalten seiner Performanz. Das erste Problem wird durch eine Skalierung des Performanzprofils der Datenbasis um den Faktor

$$f = \frac{x_{DB,max} + (count + 1) * (x_{max} - x_{DB,max})}{x_{DB,max}}. \tag{4.2}$$

gelöst. Dabei bezeichnen $x_{DB,max}$ und x_{max} die maximalen Zeitwerte des Performanzprofils der Datenbasis bzw. des aktuellen lokalen Performanzprofils. Mit wachsender Anzahl von Durchläufen stabilisiert sich durch diese Vorgehensweise dieser Wert. Ein ähnliches Verfahren wird bei der Anpassung der Qualitätswerte angewendet. Sie werden mit dem analogen Faktor skaliert, die x-Werte werden in Formel 4.2 durch die entsprechenden Qualitätswerte ersetzt.

Abbildung 4.13 zeigt ein Beispiel der Bestimmung des neuen akquirierten Performanzprofils (aPP) aus dem alten und dem aktuellen lokalen Performanzprofil der Modulkontrolle. Hier sei der Wert von *count* 1, das bedeutet, daß alte akquirierte Performanzprofil auf den Werten eines Durchlaufs basiert. Nach der Bestimmung des neuen ist der Wert dieses Zählers 2.

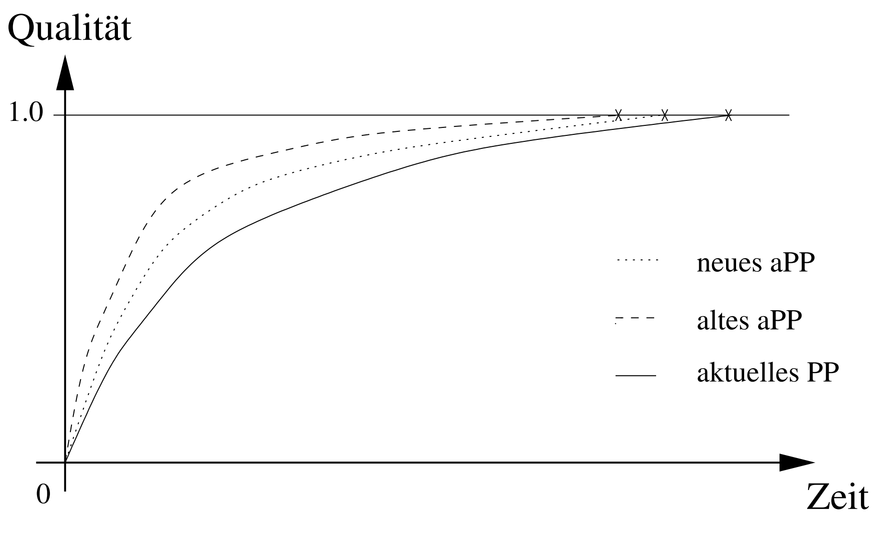

Abbildung 4.13: Bestimmung des neuen akquirierten Performanzprofils

Die zeitlichen Ressourcen werden dann unter Berücksichtigung der nach einem der Verfahren des Systems ORCAN bestimmten globalen Gewichten und der eventuell stattfindenden anschließenden Verteilung in den Teilbäumen unter den lauffähigen Tasks verteilt. Dies wird im Rahmen des Abschnitts 4.3.3.4 noch genauer betrachtet. Zunächst wird jedoch die Zuteilung der Prozesse an die Tasks anhand der lokalen

Ressourcenbäume aller RAMIs des Gesamtsystems beschrieben. Dies stellt eine Voraussetzung des Schedulings dar, denn die Ressourcen werden nur unter lauffähigen Tasks verteilt, das heißt unter solchen, die einen Prozeß zugewiesen bekommen haben.

4.3.3.3 Globaler Ressourcenbaum zur Verteilung der Prozesse

Durch die Kombination der lokalen Ressourcenbäume aller RAMIs ergibt sich ein *globaler Ressourcenbaum* des Gesamtsystems. Sein Aufbau unterliegt einigen Gesetzmäßigkeiten, die nun untersucht werden.

Es existieren wie beim lokalen Ressourcenbaum zwei Möglichkeiten für die Nachfolgeknoten eines virtuellen Tasks: ein realer Task oder ein weiterer virtueller Task. Im Unterschied zum lokalen Ressourcenbaum kann ein realer Task im globalen Ressourcenbaum Nachfolgeknoten besitzen. Dies ist dann der Fall, wenn die Berechnungen des betrachteten Tasks eine RAMI enthalten. Beim globalen Ressourcenbaum hat man es also im Gegensatz zum lokalen Ressourcenbaum nicht unbedingt mit einem Baum zu tun, bei dem alle Blätter die gleiche Tiefe besitzen, denn nicht alle Pfade zu einem Blatt müssen die gleiche Anzahl von Tasks, die eine RAMI repräsentieren, als Knoten besitzen.

Der globale Ressourcenbaum wird zur Bestimmung der Zuteilung der vorhandenen Prozesse an die einzelnen RAMIs benötigt. Dazu bekommt jede mitgeteilt, wieviele Prozesse ihr aus der Gesamtanzahl zur Verfügung stehen. Dies versteht sich inklusive ihrer untergeordneten RAMIs. Somit erhält die in der Hierarchie am höchsten angesiedelte RAMI alle Prozesse zugewiesen. In einer rekursiven Berechnung werden danach die Anzahlen für jede untergeordnete RAMI bestimmt. Seien wie in Abbildung 4.14 $K_1, \ldots, K_m$ alle Nachfolgeknoten des zum Subsystemknoten S gehörenden Vorgängerknotens V (inklusive S), bezeichnen $g_{glob}(K_1), \ldots, g_{glob}(K_m)$ ihre globalen Gewichte und sei $Tasks$ die Anzahl der lauffähigen Tasks in der aktuellen RAMI, dann bekommt die untergeordnete RAMI, die durch S repräsentiert wird, folgende Prozeßanzahl P_{RAMI} zugewiesen:

$$P_{RAMI}(S) = \max\{\lfloor \frac{g_{glob}(S)}{\sum_{j=1}^{m} g_{glob}(K_j)} * (P_{RAMI}(V) - Tasks(V)) \rfloor, 1\}. \tag{4.3}$$

Dieser Wert ergibt sich aus der Multiplikation des Verhältnisses des globalen Gewichts des RAMI-Knotens bezüglich der Summe aller globalen Gewichte seiner Geschwister inklusive ihm selbst mit der Differenz aus der Prozeßanzahl der übergeordneten RAMI und den dort vorhandenen lauffähigen Tasks. Dabei stellt die betrachtete RAMI selbst einen dieser Tasks dar. Jede RAMI erhält mindestens einen Prozeß. Sollte die Anzahl der RAMIs (Subsysteme) die Gesamtzahl der verfügbaren Prozesse übersteigen, werden zur Laufzeit des Gesamtsystems die fehlenden Prozesse erzeugt[3], so daß jederzeit eigentliche Berechnungen möglich sind und nicht die Situation entstehen kann, daß sämtliche Prozesse durch die Erzeugung von RAMIs belegt sind und die Berechnungen nicht stattfinden.

[3]Dies wirkt sich negativ auf die Laufzeit aus. Deshalb sollte die Anzahl der zur Verfügung stehenden Prozesse so gewählt werde, daß dieser Fall nicht auftritt.

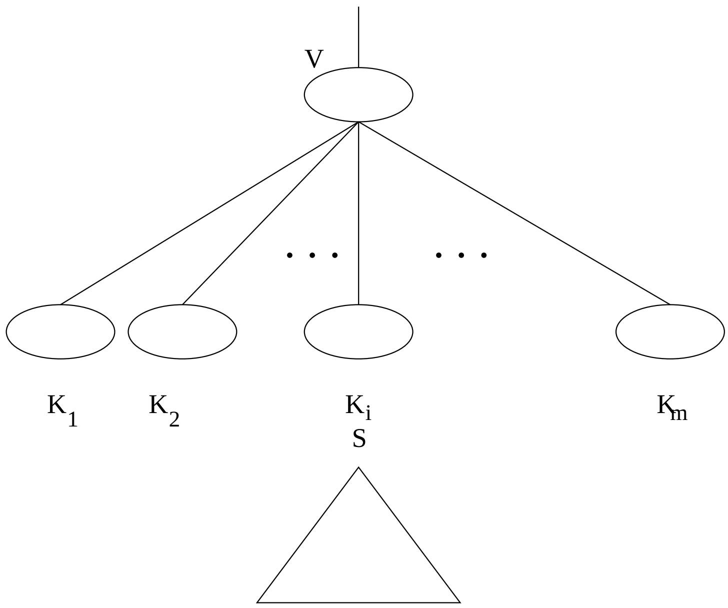

Abbildung 4.14: Verteilung der Prozesse an Subsysteme

Diese zugeteilten Prozesse werden dann innerhalb der einzelnen RAMIs mit Hilfe des lokalen Ressourcenbaums durch eine Propagierung, ausgehend vom Wurzelknoten, verteilt. Die Idee ist, daß die einem virtuellen Task zugeordneten Prozesse die Maximalanzahl darstellen, die von den Tasks seines Unterbaums benutzt werden können. Prozesse werden anhand der relativen Gewichte der Nachfolgeknoten unter diesen aufgeteilt. Dabei wird allerdings die maximale Anzahl M berücksichtigt, die in dem betreffenden Unterbaum überhaupt benötigt wird. Sei T ein virtueller Task, $K_1, \ldots, K_m$ alle Nachfolgeknoten des Vorgängerknotens V von T und $P_{Task}(V)$ die Anzahl der V zugeteilten Prozesse. Dann gilt für die T zugeordneten Prozesse $P_{Task}(T)$:

$$P_{Task}(T) = \min\{\lceil \frac{g(T)}{\sum_{j=1}^{m} g(K_j)} * P_{Task}(V) \rceil, M\} \tag{4.4}$$

Das Aufrunden verhindert, daß bei einer großen Zahl von Geschwisterknoten die Situation auftritt, daß alle nachfolgenden Teilbäume Null Prozesse zugewiesen bekommen und damit das System zu Stillstand kommt. Es können nicht mehr Prozesse als vorhanden in diesem Verfahren unter den Tasks verteilt werden, da, sobald keine freien Prozesse mehr verfügbar sind, der Zuteilungsvorgang beendet wird. Es werden unter keinen Umständen neue Prozesse erzeugt, im Gegensatz zur Vorgehensweise bei der Verteilung der Prozesse an die Subsysteme anhand des globalen Ressourcenbaumes.

Handelt es sich bei den Nachfolgeknoten eines virtuellen Tasks um reale Tasks, so werden die verfügbaren Prozesse in absteigender Reihenfolge der relativen Gewichte einzeln vergeben und die betreffenden Tasks somit in lauffähige Tasks transformiert.

Abbildung 4.15 zeigt hierzu ein Beispiel. Die Werte über den Knoten geben an, wieviele Prozesse ihnen nach Auswertung des ersten Arguments der Minimumbildung aus Formel 4.4 zugewiesen würden, die eingeklammerten Werte links dane-

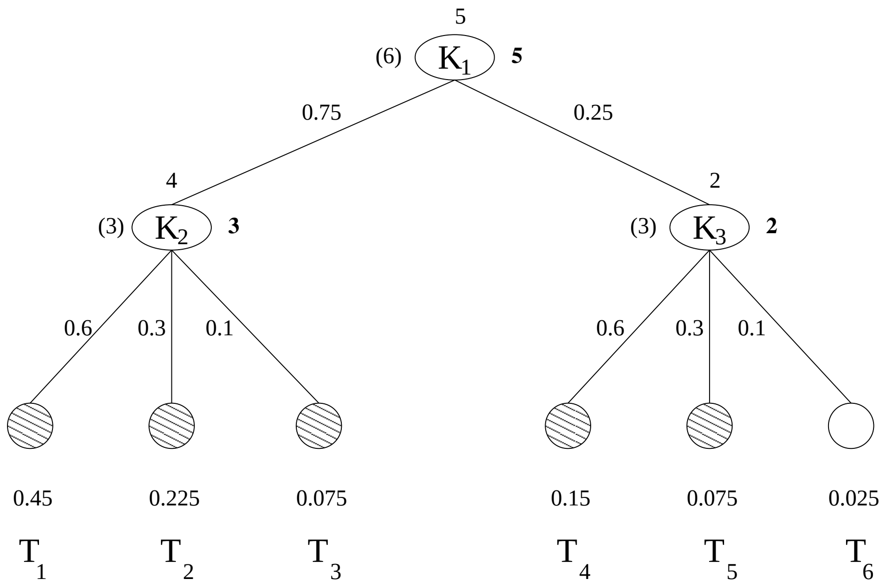

Abbildung 4.15: Beispiel zur Verteilung der Prozesse

ben stehen für die maximale Anzahl der Prozesse, die im betreffenden Unterbaum benötigt werden. Fett gedruckt rechts neben jedem Knoten findet sich die letztendlich zugewiesene Anzahl der Prozesse. Lauffähige Tasks sind schraffiert dargestellt. Im Beispiel stehen insgesamt fünf Prozesse zur Verteilung bereit. Diese werden zu Beginn dem Wurzelknoten K_1 zugewiesen. Danach werden anhand von Formel 4.4 die Werte für die Nachfolgerknoten K_2 und K_3 bestimmt:

$$P_{Task}(K_2) = \min\{\lceil \frac{0.75}{0.75 + 0.25} * 5 \rceil, 3\} = \min\{4, 3\} = 3 \tag{4.5}$$

$$P_{Task}(K_3) = \min\{\lceil \frac{0.25}{0.75 + 0.25} * 5 \rceil, 3\} = \min\{2, 3\} = 2 \tag{4.6}$$

Da K_2 drei reale Tasks als Nachfolger besitzt und drei Prozesse zur Verfügung stehen, bekommen sie alle einen Prozeß zugewiesen. K_3 hingegen kann nur zwei verteilen und da es sich bei den Nachfolgern um reale Tasks handelt, werden diese in der Reihenfolge ihrer relativen Gewichten berücksichtigt, das bedeutet, nur T_4 und T_5 werden lauffähig.

Das Scheduling der so erzeugten lauffähigen Tasks ist Inhalt des folgenden Abschnitts.

4.3.3.4 Scheduling der Tasks

Nachdem erläutert wurde, wie die Aufteilung des Gastsystems in Tasks erzeugt und die Verteilung der Prozesse durchgeführt wird, folgt die Vorgehensweise zum Scheduling der lauffähigen Tasks.

Die Bearbeitung der der betrachteten RAMI gestellten Aufgabe erfolgt in Zyklen, so daß einzelne Teilaufgaben, die durch Tasks bearbeitet werden, immer wieder eine gewisse Zeit zugeteilt bekommen, sie aber nicht andere Berechnungen blockieren. Nachdem die ihnen zugeteilte Zeit vergangen ist, werden sie unterbrochen und der nächste Task beginnt mit seiner Rechenphase. Die einen Zyklus ausmachende Zeitspanne stellt einen Parameter des Systems dar, der vom Entwickler (mittels der Java-Werkbank) an das vorliegende Gastsystem angepaßt werden kann. Treten relativ rechenintensive Tasks auf, dann ist es im allgemeinen vorteilhaft, eine etwas längere Zykluszeit zu wählen. Ebenso beeinflußt die Anzahl der Tasks die optimale Größe dieses Parameters, der die Verbesserung der Ergebnisqualität des Gesamtsystems stark beeinflußt. Er sollte so gewählt werden, daß er einerseits nicht zu klein ist und somit die einzelnen Tasks zuwenige Ressourcen erhalten, um eine Verbesserung des Ergebnisses zu bewirken, und daß er andererseits nicht zu groß ist, so daß wenige Tasks sehr viele Ressourcen bekommen und andere Aspekte der Berechnung des Gesamtsystems vernachlässigt werden.

Die Idee des Schedulingverfahrens besteht darin, die zur Verfügung stehende Zykluszeit gemäß den globalen Gewichten unter den lauffähigen Tasks zu verteilen. Die Ressourcen $t(T)$, die ein lauffähiger Task T innerhalb eines Zyklus zugewiesen bekommt, bestimmen sich als Produkt aus dem Verhältnis zwischen seinem globalen Gewicht $g_{glob}(T)$ und der Summe der globalen Gewichte $g_{glob}(T_1), \ldots, g_{glob}(T_m)$ aller noch nicht bearbeiteten lauffähigen Tasks $T_1, \ldots, T_m$ (inklusive T selbst) und der verbliebenen Zykluszeit t_{Rest}:

$$t(T) = \frac{g_{glob}(T)}{\sum_{j=1}^{m} g_{glob}(T_j)} * t_{Rest}. \tag{4.7}$$

Zusätzlich wird das *minimale Prozeßquantum*, das die minimale Ressourcenzuteilung an die lauffähigen Tasks festlegt, beachtet. Es ist ein Systemparameter, der abhängig ist von der Umgebung in der das System arbeitet und der zu Beginn global festgelegt wird. Würde eine geringere Zeitspanne zugeteilt werden, könnte die Situation entstehen, daß die Zeit für die Umgebung nicht ausreichend ist, um administrative Aufgaben beim Kontextwechsel zu erledigen und es zu keinen eigentlichen Berechnungen des Tasks kommt.

Die so bestimmte Zeitspanne steht der Berechnung des Tasks innerhalb des aktuellen Zyklus zur Verfügung. Es besteht allerdings die Möglichkeit, daß diese nicht komplett benötigt wird, falls der Task seine Aufgabe früher beendet, oder daß sie nicht genügt, falls der Tasks sich zum Ende der berechneten Zeitspanne in einer Transaktion befindet und diese noch abschließen muß. Deshalb wird zusätzlich zur vorberechneten Zeit noch die wirklich benötigte Zeit $t_{real}(T)$ gemessen. Dadurch ergibt sich nach der Bearbeitung des Tasks als neue verbleibende Zykluszeit $t_{Rest_{neu}}$:

$$t_{Rest_{neu}} = t_{Rest} - t_{real}(T). \tag{4.8}$$

Sie wird dann zur Berechnung der Ressourcen des in diesem Zyklus folgenden Tasks verwendet, womit gewährleistet ist, daß innerhalb eines Zyklus keine Ressourcen ungenutzt bleiben.

Diese Vorgehensweise wird solange wiederholt, bis entweder alle lauffähigen Tasks abgearbeitet sind oder die gesamte Zykluszeit verbraucht ist (abgesehen von eventuell auftretenden externen Stop-Signalen). In Abbildung 4.16 ist der Pseudocode dieses Verfahrens dargestellt. Dabei wird `pop` zum sukzessiven Abbau der Liste aller Tasks verwendet. Die Funktion `lasse_laufen` ist zuständig dafür, daß der betrachtete Task die vorher bestimmte Zeit zur Verfügung hat, um seine Berechnungen durchzuführen, und daß eventuell vorhandene Transaktionen beachtet werden. Dieser Vorgang wird im folgenden anhand eines Beispiels verdeutlicht.

`Scheduler`$(Tasks_{lauffähig},\ t_{Zyklus})$
$t_{Rest} \leftarrow t_{Zyklus}$
while not empty $Tasks_{lauffähig}$ **and** $t_{Rest} > 0.0$ **do**
  $T \leftarrow$ `pop`$(Tasks_{lauffähig})$
  $t(T) \leftarrow \frac{g_{glob}(T)}{\sum_{j=1}^{m} g_{glob}(T_j)} * t_{Rest}$
  `initialisiere_Uhr`
  `lasse_laufen`$(T, t(T))$
  $t_{real}(T) \leftarrow$ `stoppe_Uhr`
  $t_{Rest} \leftarrow t_{Rest} - t_{real}(T)$
od

Abbildung 4.16: Vorgehensweise des Schedulers

Betrachtet man die Situation aus Abbildung 4.15, so müssen die Tasks $T_1, \ldots, T_5$ vom Scheduler verwaltet werden. Die Dauer eines Zyklus betrage in diesem Beispiel zehn Zeiteinheiten. Es wird mit der Bearbeitung von T_1 begonnen. Zu Beginn umfaßt die verbliebene Zykluszeit t_{Rest} die komplette Zeitspanne von zehn Zeiteinheiten (ZE). Damit ergeben sich nach Formel 4.7 folgende Ressourcen für den Tasks T_1:

$$t(T_1) = \frac{0.45}{0.45 + 0.225 + 0.075 + 0.15 + 0.075} * 10ZE = 4.62ZE. \qquad (4.9)$$

T_1 erhält diese Zeitspanne von $4.62ZE$ zugewiesen, um seine Berechnungen durchzuführen. Nach deren Ablauf wird er vom Scheduler unterbrochen und die verbleibende Zykluszeit neu berechnet:

$$t_{Rest_{neu}} = 10ZE - 4.62ZE = 5.38ZE. \qquad (4.10)$$

In diesem Fall stimmen die vorberechnete Zeitspanne und die wirklich benötigte überein. Danach folgt die Bearbeitung des nächsten Tasks T_2. Für ihn ergeben sich die Ressourcen:

$$t(T_2) = \frac{0.225}{0.225 + 0.075 + 0.15 + 0.075} * 5.38ZE = 2.31ZE. \qquad (4.11)$$

Im Gegensatz zu T_1 befinde sich die Berechnung von T_2 zum Zeitpunkt der Unterbrechung durch den Scheduler nach $2.31ZE$ in einer Transaktion. Diese muß beendet werden und nimmt somit zusätzliche Zeit in Anspruch. In diesem Fall nehmen wir an, daß der Scheduler eine Gesamtlaufzeit von $3.11ZE$ für T_2 mißt. Es folgt:

$$t_{Rest_{neu}} = 5.38ZE - 3.11ZE = 2.27ZE. \qquad (4.12)$$

Diese $2.27ZE$ werden analog zu den bisher aufgeführten Berechnungen unter den restlichen Tasks T_3, T_4 und T_5 verteilt. Haben alle Tasks ihre Berechnungen durchgeführt, beginnt nach dem Anpassen des lokalen Ressourcenbaumes der nächste Zyklus und es werden die dann lauffähigen Tasks in dieser Weise geschedult.

```
Anpassen_des_Ressourcenbaumes(T)
while not Nachfolger(T) do
    Entferne_Task(T)
    T ← Vorgänger(T)
od
```

Abbildung 4.17: Anpassen des lokalen Ressourcenbaumes

Wird ein Task beendet, so muß der lokale Ressourcenbaum an die neue Situation angepaßt und der benutzte Prozeß freigegeben werden. Das Vorgehen hierzu zeigt der Algorithmus in Abbildung 4.17.

Es wird mit dem Entfernen des beendeten Tasks T aus dem lokalen Ressourcenbaum begonnen. Dann wird solange mit dem Löschen von Vorgängerknoten zur Wurzel hin fortgefahren bis ein virtueller Task erreicht ist, der noch andere Nachfolgeknoten, außer dem zuletzt gelöschten, besitzt.

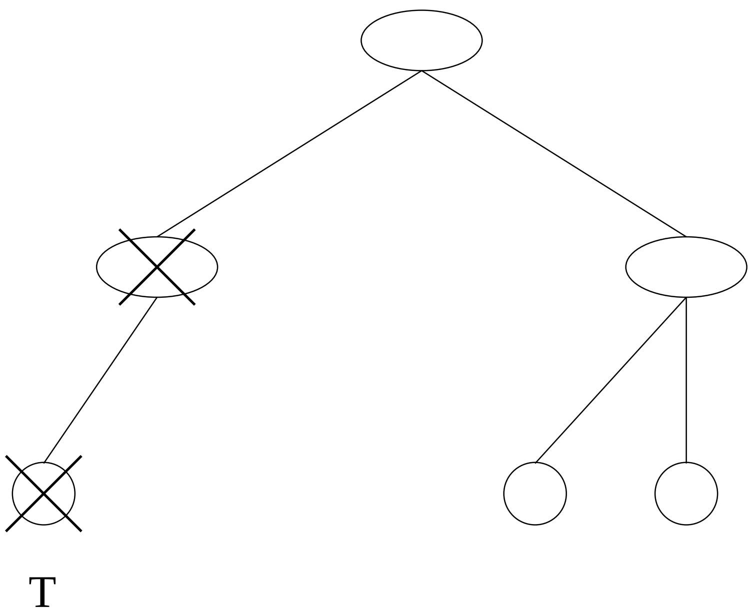

Abbildung 4.18: Beispiel zum Entfernen eines beendeten Tasks aus dem lokalen Ressourcenbaum

Abbildung 4.18 zeigt das Entfernen des beendeten Tasks T aus dem dargestellten lokalen Ressourcenbaum. Der übergeordnete Knoten zu T wird ebenfalls gelöscht, dann allerdings muß die Wurzel im nächsten Schritt beibehalten werden, da sie noch andere Nachfolgeknoten hat.

Die potentiell rekursive Struktur des Gesamtsystems durch die Verwendung von Subsystemen repräsentiert durch RAMIs muß beim Scheduling der lauffähigen Tasks besonders berücksichtigt werden, wie die folgende Situation veranschaulicht: Der aktuell arbeitende Task, dessen Berechnungen eine RAMI darstellen, hat seine ihm zugewiesenen Ressourcen aufgebraucht und soll unterbrochen werden damit der nächste lauffähige Task wiederum in seine Berechnungsphase eintreten kann. Es wird sichergestellt, daß der Scheduler der RAMI ebenfalls unterbrochen wird, insbesondere der gerade dort arbeitende Task, so daß nicht ein Task ohne Kontrolle durch einen Scheduler solange weiterarbeitet bis der zum Scheduler gehörende Task einer übergeordneten RAMI erst im nächsten Zyklus wieder zum Zuge kommt. Dieses Problem wird wie in Abbildung 4.19 dargestellt rekursiv behandelt, da mehrere Subsystemebenen existieren können. Zuerst wird in der Situation der Abbildung der sich im Moment in der Berechnungsphase befindliche Task in RAMI 2 unterbrochen, danach der übergeordnete Task in RAMI 1, der das zweite Subsystem beinhaltet.

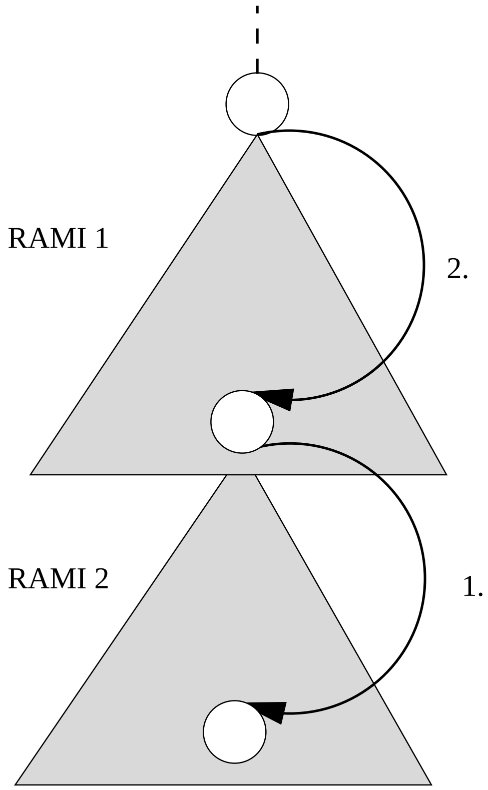

Abbildung 4.19: Rekursives Unterbrechen der Tasks

4.3.3.5 Alternative Vorgehensweise zur Verwaltung der Tasks: Sukzessiver Aufbau des lokalen Ressourcenbaums

In diesem Abschnitt wird eine Alternative zu der in den vorigen Abschnitten vorgestellten Vorgehensweise der Anytime-Kontrolle beschrieben. Der zentrale Unterschied besteht in der Verwaltung des lokalen Ressourcenbaumes. Er wird nicht zu Beginn eines Systemlaufs komplett aufgebaut, sondern sukzessive vor jedem Zyklus erweitert. Er wird immer nur soweit konstruiert, wie freie Prozesse zur Verfügung stehen. Bei diesem Verfahren existieren keine nichtlauffähigen realen Tasks. Damit entfällt die vor jedem Zyklus durchgeführte Verteilung der Prozesse im lokalen Ressourcenbaum. Die Tasks werden in absteigender Reihenfolge ihres relativen Gewichts in die Ebenen des Baums integriert, insbesondere wird als erster realer Task derjenige erzeugt, der auf allen Ebenen seines Pfades die größten relativen Gewichte besitzt. Dieses Vorgehen stellt eine Priorisierung der Tasks anhand ihrer relativen Gewichte dar. Deren genaue Größe spielt bei der Verteilung der Prozesse keine Rolle. Ihre Aufgabe besteht in diesem Zusammenhang lediglich darin, eine Ordnung auf den Tasks jeder Ebene des lokalen Ressourcenbaumes zu etablieren.

Der lokale Ressourcenbaum aus Abbildung 4.15 sähe bei Verwendung dieses Verfahrens wie in Abbildung 4.20 dargestellt aus. Der Ast, der den nichtlauffähigen Task T_6 repräsentiert, existiert in diesem Fall nicht, da lediglich fünf Prozesse zur Verfügung stehen.

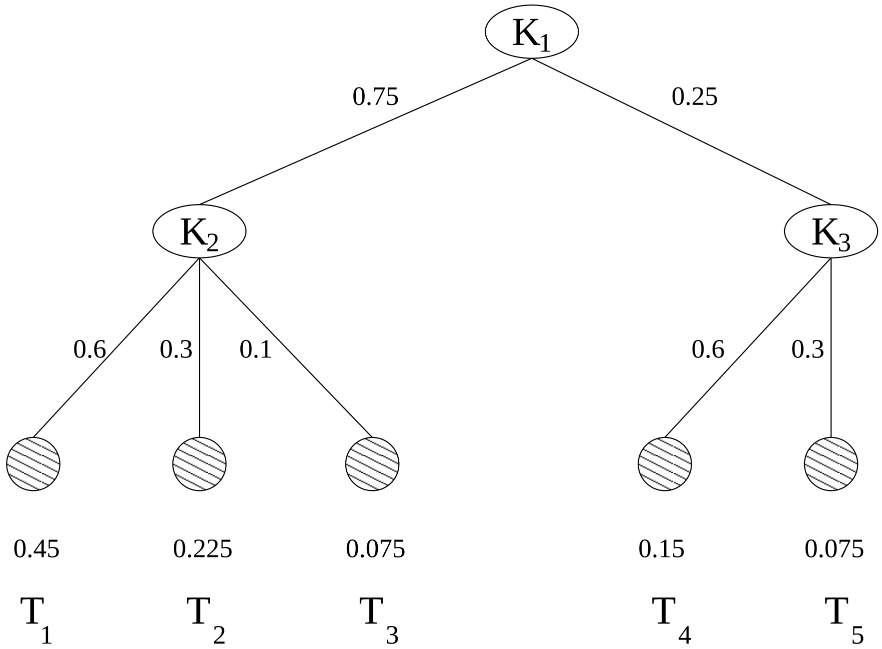

Abbildung 4.20: Beispiel eines lokalen Ressourcenbaums bei alternativer Vorgehensweise

Task T_6 wird in einer späteren Phase in den Baum integriert. Dies geschieht nach der Beendigung eines der lauffähigen Tasks. Wird beispielsweise T_2 beendet, dann sieht der vor Beginn des nächsten Zyklus um T_6 erweiterte lokale Ressourcenbaum wie in Abbildung 4.21 aufgezeigt aus.

Die Bestimmung der globalen Gewichte und das Scheduling der lauffähigen Tasks erfolgt wie in den Abschnitten 4.3.3.2.2, 4.3.3.2.3 und 4.3.3.4 vorgestellt. Insgesamt existieren also vier Möglichkeiten wie die Anytime-Kontrolle bezüglich der Ressourcenverteilung vorgehen kann: Zwei Arten der Verwaltung des lokalen Ressourcenbaumes (komplett oder sukzessiver Aufbau), die beliebig mit den zwei unterschiedlichen Vorgehensweisen zur Bestimmung der globalen Gewichte (mit oder ohne Datenbasis der Performanzprofile) kombiniert werden können.

4.3.3.6 Vergleich der beiden Alternativen zur Verwaltung der Tasks

Der wesentliche Unterschied der beiden Alternativen zur Verwaltung der Tasks besteht in der Art der Zuteilung der Prozesse an die realen Tasks. Beim kompletten Aufbau des lokalen Ressourcenbaumes werden die zur Verfügung stehenden Pro-

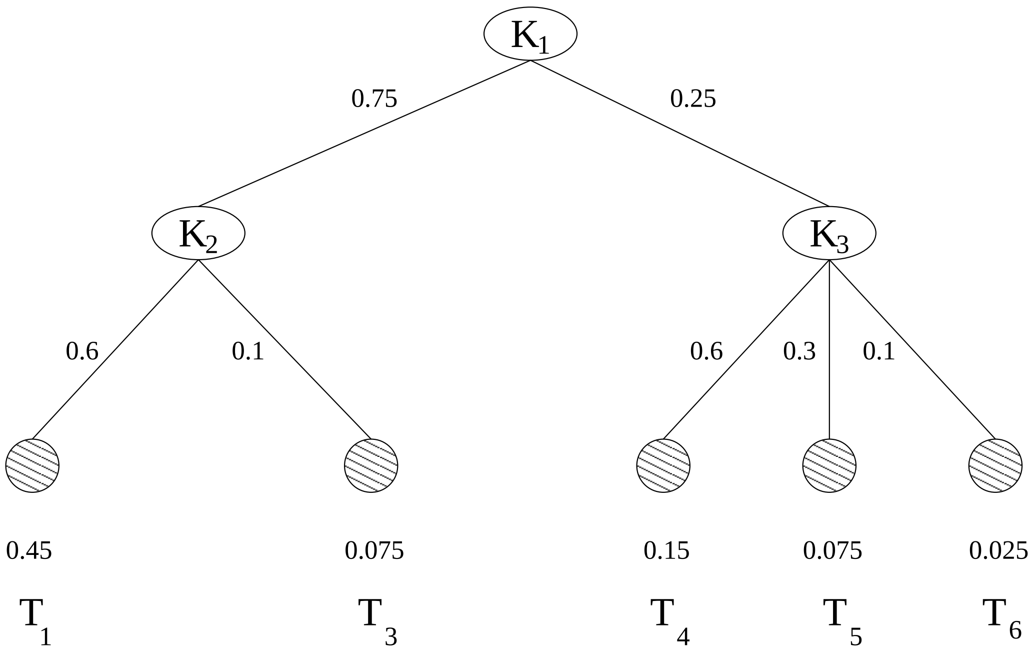

Abbildung 4.21: Beispiel der Erweiterung eines lokalen Ressourcenbaums bei alternativer Vorgehensweise

zesse jeweils proportional zu den relativen Gewichten an die Nachfolger im Baum verteilt. Somit bekommen schon sehr früh auch Teilaufgaben geringerer Priorität Prozesse zugewiesen, um ihre Berechnungen durchzuführen. Im Rahmen des sukzessiven Aufbaus werden hingegen die Teilaspekte der Gesamtberechung bevorzugt, die ein hohes relatives Gewicht besitzten. In diesem Fall kann man von einer Priorisierung dieser Tasks sprechen. Das Verfahren des kompletten Aufbaus des lokalen Ressourcenbaumes ist durch eine Betonung des Aspekts der Nebenläufigkeit der Berechnung charakterisiert, womit ein besseres Anytime-Verhalten des Gesamtsystems einhergeht, insbesondere wird sich im allgemeinen die Ergebnisqualität in feineren Schritten verbessern. Das alternative Verfahren kann hingegen bei entsprechender Wahl der relativen Gewichte die Bestimmung eines (guten) Ergebnisses zu einem früheren Zeitpunkt ermöglichen. Die vorher verfügbaren Zwischenergebnisse sind dann allerdings von minderer Qualität (falls diese existieren).

Betrachtet man den Overhead, der durch die Verwendung der Anytime-System-Shell im Vergleich zum Stand-alone-Betrieb des Gastsystems entsteht, dann tritt er bei den beiden Alternativen zu großen Teilen zu verschiedenen Zeitpunkten auf. Dabei handelt es sich um die Zeitspannen, die zur Verwaltung der lokalen Ressourcenbäume gebraucht werden. Beim kompletten Aufbau des lokalen Ressourcenbaumes findet seine Konstruktion zu Beginn statt. Die zweite Alternative verteilt den dazu benötigten Overhead über die gesamte Dauer der Berechnung. In gewissem Sinne wird der zu jedem realen Task gehörige Overhead an den Zeitpunkt seiner Ausführung gekoppelt. Dies führt dazu, daß beim kompletten Aufbau ein erstes Zwischenergebnis erst später vorliegt, da die Zeit zur Konstruktion des gesamten lokalen Ressourcenbaumes den eigentlichen Berechnungen nicht zur Verfügung steht. Der sukzessive Aufbau bewirkt hingegen in einer späteren Phase des Systemlaufs eine langsamere Verbessserung der Ergebnisqualität, da immer wieder Zeit zur Expansion des lokalen Ressourcenbaumes vor jedem Zyklus benötigt wird.

Nachdem die Verfahrensweisen der Anytime-System-Shell erläutert wurden folgt die Beschreibung der zugehörigen Java-Werkbank mit deren Hilfe die konstruierten Gesamtsysteme vom Benutzer kontrolliert und manipuliert werden können. Die mit Hilfe der Anytime-System-Shell konstruierten Gesamtsysteme können allerdings in der Entwicklungs- und Test-Phase auch ohne diese Java-Oberfläche betrieben werden.

4.4 Java-Werkbank

Die plattformunabhängige Java-Werkbank der Anytime-System-Shell stellt eine Benutzerschnittstelle dar, die der Anfragestellung, der Ausgabe von Ergebnissen und der Modifikation von Systemparametern dient. Es können der interne Ablauf des Gesamtsystems und die dazu benötigten Datenstrukturen beobachtet und beeinflußt werden. Die Vorstellung dieser Funktionen und Eigenschaften ist Inhalt dieses Abschnitts. Einen Eindruck des Erscheinungsbildes der Werkbank liefert Abbildung 4.22.

4.4.1 Plattformunabhängigkeit

Durch die Abkoppelung der Werkbank von der eigentlichen Funktionalität der Anytime-System-Shell und die Verwendung der Programmiersprache Java (Gosling, Joy & Steele, 1996) zur Implementation ergibt sich eine flexible Ansteuerungsmöglichkeit des Gesamtsystems. Außer der üblichen Nutzungsweise des Systems auf einem lokalen Rechner ist auch eine Ansteuerung über das World Wide Web (WWW) möglich. Dazu muß vom Entwickler eine zum System gehörende Web-Seite bereitgestellt werden. Die Vorgehensweise hierzu wird in Abschnitt 4.5.4.1 erläutert. Somit ist es prinzipiell möglich, das System ortsunabhängig zu nutzen[4].

4.4.2 Leistungen

Im folgenden werden die Funktionen und Leistungen der Werkbank zur Kontrolle der mit Hilfe der Anytime-System-Shell konstruierten Systeme vorgestellt.

4.4.2.1 Ein-/Ausgabe

Eine der wichtigsten Aufgaben der Werkbank ist die Eingabe von Aufgabenstellungen für das entwickelte System und die Ausgabe von (Zwischen-)Ergebnissen bei Unterbrechung bzw. Beendigung der vorgenommenen Berechnungen. Hierbei handelt es sich um reine Ein- und Ausgabefunktionen, das heißt die Daten werden von der Werkbank unmanipuliert ans System geschickt bzw. von ihr entgegengenommen und angezeigt. Ihre Bearbeitung erfolgt auf der Seite der Anytime-System-Shell.

[4] Voraussetzung ist lediglich ein Rechner mit WWW-Zugang.

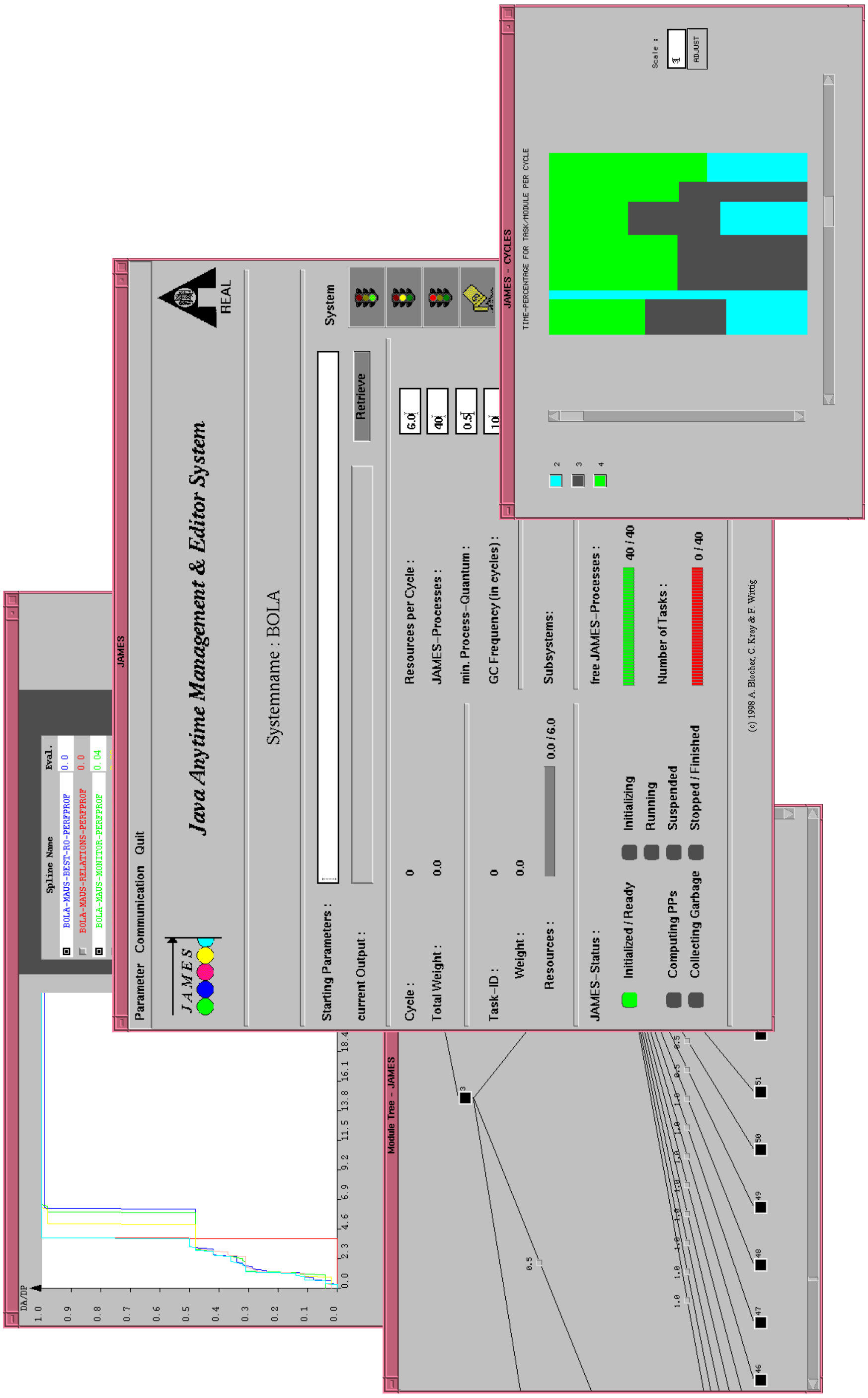

Abbildung 4.22: Die Java-Werkbank

4.4.2.2 Systemsteuerung

Es existieren fünf Steuerungselemente, mit deren Hilfe Signale an das System gesendet werden können, die dessen Kontrolle dienen:

1. **Start:** Mit dieser Funktion wird das Gesamtsystem nach Eingabe der Aufgabenstellung mit den aktuell eingestellten Parametern gestartet. Außerdem kann es hiermit aus einem unterbrochenen Zustand heraus wieder fortgesetzt werden.

2. **Unterbrechung:** Wird ein Unterbrechungssignal an das System gesendet, so wird es direkt in einen Wartezustand versetzt und das aktuelle Zwischenergebnis der obersten Modulkontrolle (also die des eigentlichen Gastsystems) wird angezeigt. Befindet sich die Berechnung zum Zeitpunkt des Eintreffens des Unterbrechungssignals in einer Transaktion, so wird diese noch beendet bevor das System unterbrochen wird.

3. **Stop:** Das System wird im aktuellen Zustand angehalten und die Berechnungen abgebrochen. Das zu diesem Zeitpunkt bestimmte Ergebnis wird wie bei einer Unterbrechung ausgegeben. Nach einem Stop kann das System im Gegensatz zu einer Unterbrechung nicht mehr fortgesetzt werden. Dies ist dadurch bedingt, daß verschiedene Datenstrukturen verändert werden und somit zur Fortsetzung benötigte Daten verloren gehen.

4. **Retrieve:** Ein Retrieve-Signal fordert das aktuelle Zwischenergebnis an und gibt es aus, ohne die Berechnungen des Gesamtsystems zu unterbrechen. Somit ist es möglich, die Entwicklung des Ergebnisses zu verfolgen, ohne das Gesamtsystem anzuhalten und den Overhead zu vergrößern.

5. **Reinitialisierung:** Eine Reinitialisierung versetzt die benutzten Datenstrukturen in einen neutralen Zustand und bereit so das System auf eine neue Aufgabenstellung vor.

Zu Beginn eines Systemlaufs ist nur ein Start-Signal möglich. Danach kann entweder ein Unterbrechungssignal oder ein Stop-Signal erfolgen. Im ersten Fall besteht die Möglichkeit, durch das Senden eines erneuten Start-Signals die unterbrochene Berechnung weiterzuführen. Reinitialisierungssignale sind nur im unterbrochenen, im gestoppten und im beendeten Zustand sinnvoll. Wird sofort ein neues Start-Signal gegeben, dann erfolgt eine automatische Reinitialisierung. Ein Retrieve-Signal kann jederzeit abgesetzt werden.

4.4.2.3 Anzeige der Performanzprofile und statistischer Daten

Eine weitere wichtige Leistung der Werkbank besteht in der Bereitstellung einer Vielzahl von statistischen Werten anhand derer das Verhalten sowohl des Gesamtsystems als auch gegebenenfalls der einzelnen Subsysteme nachvollzogen werden kann. Es sind dies folgende:

- **Performanzprofile:** Die Performanzprofile sind die wichtigsten Hilfsmittel beim Untersuchen des Verhaltens des Gesamtsystems und seiner Subsysteme.

Ihre Darstellung basiert auf den mitprotokollierten Werten in den verschiedenen Modulkontrollen der zu den (Sub-)Systemen gehörenden RAMIs (s. Abschnitt 4.3.2.4).

- **Ressourcenverteilung innerhalb der Zyklen:** Es wird eine Visualisierung der Aufteilung der zeitlichen Ressourcen jedes Zyklus in Form zweidimensionaler Balkendiagramme bereitgestellt. Die horizontale Achse entspricht hierbei der Zeitachse. Auf ihr werden anhand der Breite der Balken die Länge der Zyklen aufgetragen. In vertikaler Richtung werden die prozentualen Anteile der im entsprechenden Zyklus bearbeiteten Tasks bezüglich der Gesamtzykluszeit dargestellt. Abbildung 4.23 zeigt ein Beispiel eines solchen zweidimensionalen Balkendiagramms. Die verschiedenen gleichschraffierten Bereiche symbolisieren die benutzten Ressourcen der verschiedenen Tasks. Die unterschiedliche Länge der Zyklen resultiert aus Transaktionen die zum eigentlichen Ende des Zyklus noch fertiggestellt werden müssen und aus Tasks, die früher als erwartet ihre Berechnungen beenden. Zusätzlich besteht die Möglichkeit solche

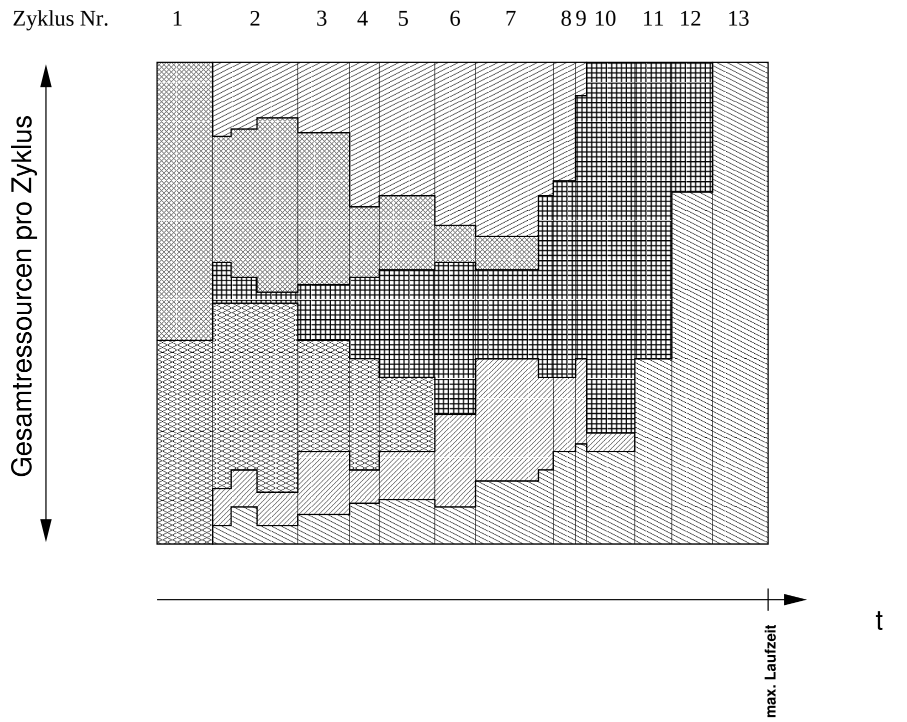

Abbildung 4.23: Beispiel eines zweidimensionalen Balkendiagramms zur Darstellung der Ressourcenverteilung

Diagramme für virtuelle Tasks erstellen zu lassen. Dazu muß vor dem Systemlauf die betreffende Ebene im lokalen Ressourcenbaum festgelegt werden, von deren virtuellen Tasks die Ressourcenverteilung mitprotokolliert werden soll.

- **Informationen über die Tasks:** Nach jedem Durchlauf des Systems können Informationen zu den einzelnen Tasks abgefragt werden. Die wichtigsten sind der Gesamtverbrauch an Ressourcen, die Lebensdauer in Zyklen und der Name. Die zusätzlich angezeigten Daten sind rein technischer Natur und können im Rahmen der Fehlersuche bei der Konstruktion eines Gesamtsystems nützlich sein.

4.4.2.4 Anzeige systeminterner Strukturen

Die Anzeigemöglichkeit der lokalen Ressourcenbäume versetzt den Benutzer in die Lage, jederzeit den Stand der Berechnungen des Gesamtsystems nachvollziehen zu können. Er kann erkennen, welche Tasks schon abgearbeitet sind, welche sich gerade im lauffähigen Zustand befinden und welche noch bearbeitet werden müssen. Anhand der relativen und globalen Gewichte der virtuellen und realen Tasks kann der Verlauf der Ressourcenverteilung bis in alle Details überwacht werden. Besonders in der Entwicklungsphase eines Gesamtsystems wird mit der Darstellung der lokalen Ressourcenbäume ein großer Beitrag zur Vereinfachung der Konstruktion geleistet.

4.4.2.5 Parametermodifikation

Die den Werkbankcharakter unterstreichende Leistung ist die Möglichkeit der Modifikation von Systemparametern. Damit kann experimentell das Verhalten des konstruierten Systems unter verschiedensten Ressourcenbeschränkungen untersucht werden. Nachfolgend werden die veränderbaren Parameter aufgelistet und ihre Bedeutung erläutert.

- **Anzahl der zur Verfügung stehenden Prozesse:** Die Anzahl der der Berechnung zur Verfügung stehenden Prozesse ist ein entscheidender Faktor für das Verhalten des Gesamtsystems. Sind zu wenige vorhanden, so führt dies zu einer praktisch sequentiellen Abarbeitung der Tasks wie man sich anhand des Extremfalles eines einzigen Prozesses verdeutlichen kann. Der nebenläufige Charakter geht verloren und der Overhead durch die administrativen Tätigkeiten des Anytime-System-Shell wächst stark an. Steht nur ein Prozeß zur Verfügung, so würde man das gleiche Ergebnis wesentlich schneller ohne die Verwendung der Anytime-System-Shell erzielen. Andererseits ist auch nicht wünschenswert, soviele Prozesse wie möglich zu haben, denn je mehr Prozesse von den Tasks benutzt werden können, desto geringer werden die Zeitspannen, die den einzelnen Tasks innerhalb eines Zyklus zugeteilt werden und der Overhead durch das Durchführen vieler Kontextwechsel steigt an. Das Problem der richtigen Wahl der Anzahl der Prozesse ist deshalb eine schwierige Entscheidung. Es wird die Möglichkeit geboten, diese Anzahl während einer Unterbrechung der Berechnung zu modifizieren, das bedeutet, es werden dann, zur Laufzeit, gegebenenfalls neue Prozesse erzeugt oder überzählige entfernt.

- **Länge der Zyklen:** Ebenso wie die Anzahl der Prozesse ist die Zykluslänge ein Aspekt, der starken Einfluß auf das Verhalten des Systems besitzt. Eine relativ lange Zyklusdauer bewirkt größere Ressourcenzuteilungen an die einzelnen Tasks pro Zyklus, wohingegen kürzere Zyklen die Auswirkungen des Overheads durch die Kontextwechsel verstärken. Die Größe dieses Parameters stellt lediglich einen Anhaltspunkt dar, denn Transaktionen und frühzeitig beendete Tasks können zu einer etwas längeren oder kürzeren Zyklusdauer führen. Auch dieser Parameter kann im Verlaufe der Berechnung modifiziert werden.

- **Minimales Prozeßquantum:** Die Größe des minimalen Prozeßquantums ist abhängig von der Umgebung (Hardware, Betriebssystem) in der das System arbeiten soll. Diese Wahl muß nur einmal getroffen werden, der Parameter kann aber zur Simulation verschiedener Szenarien verändert werden.

- **Garbage-Collection-Frequenz:** Ein weiterer Parameter ist die *Garbage-Collection-Frequenz*. Sie wird in Abschnitt 4.5.2.9 im Rahmen der Beschreibung der Implementierung der Anytime-System-Shell erläutert.

- **Relative Gewichte der Tasks:** Die Java-Werkbank bietet die Möglichkeit, die relativen Gewichte aller Tasks, sowohl der virtuellen als auch der realen, zu manipulieren. Wird eine solche Änderung vorgenommen, dann werden die davon abhängigen Werte der anderen Tasks des zugehörigen lokalen Ressourcenbaumes angepaßt. Damit können Varianten zu der aus den Konfigurationsdaten resultierenden Ressourcenverteilung des Gesamtsystems betrachtet werden.

4.4.2.6 Anzeige des Systemstatus

Zu jedem Zeitpunkt stellt die Werkbank Informationen über den Status des Gesamtsystems bereit. Es existieren sieben Zustände:

1. **Initialisierungsphase:** In der Initialisierungsphase bereitet das Gesamtsystem die Prozesse für ihre Verwendung vor und baut die internen Datenstrukturen anhand der Konfigurationsdaten aus der Modulkontrolle auf.

2. **Ready-Zustand:** Das System wartet auf eine Anfrage des Benutzers.

3. **Berechnungsphase:** In dieser Phase finden die eigentlichen Berechnungen zur Lösung der Problemstellung des Systems statt. Hierzu gehören in diesem Fall auch die administrativen Aufgaben der Initialisierungsphasen von Gastsystemen und Subsystemen, da diese eine Lösungsstrategie eines Teilproblems des übergeordneten Systems darstellen und damit ein Teil seiner Berechnungsphase sind.

4. **Unterbrechung:** Es wurde ein Unterbrechungssignal erteilt und noch nicht durch ein Start-Signal aufgehoben. In diesem Zustand können die Systemparameter modifiziert werden.

5. **Berechnen der Performanzprofile:** Dieser Zustand wird direkt nach einer Unterbrechung oder dem Stoppen des Systems angenommen. Es werden die Performanzprofile aus den mitprotokollierten Daten der Modulkontrolle ermittelt und für die Weiterverwendung gesichert.

6. **Garbage-Collection:** Hier wird wie im vorhergehenden Abschnitt auf Abschnitt 4.5.2.9 verwiesen.

7. **Beendet:** Das System hat entweder alle Berechnungen durchgeführt und sein Endergebnis ausgegeben oder es wurde ein Stop-Signal gesendet und das bis zu diesem Zeitpunkt erzielte Zwischenergebnis angezeigt.

4.4.2.7 Anzeige der Informationen von Subsystemen

Die Werkbank bietet die Möglichkeit auch die Parameter und Datenstrukturen aller Subsysteme bzw. RAMIs zu modifizieren bzw. zu visualisieren. Diese Werte sind unabhängig von den Parametern der übergeordneten RAMIs. So können beispielsweise die Zykluslänge und das minimale Prozeßquantum für jede RAMI gesondert vorgegeben und verändert werden. Die einzige Größe, die für alle Teilsysteme Gültigkeit besitzt, ist die Anzahl der insgesamt zur Verfügung stehenden Prozesse.

4.4.2.8 Abschalten von Werkbankfunktionen

Die erläuterten Anzeige- und Modfikationsmöglichkeiten setzen gewisse Datenstrukturen auf der Seite der Anytime-System-Shell zum Mitprotokollieren der betrachteten Werte voraus (s. Abschnitt 4.5.2.1). Der dadurch entstehende (zeitliche) Overhead kann durch Abschalten der Funktionen der Werkbank (abgesehen von der Ein- und Ausgabe und der Möglichkeit, Steuerungssignale abzusetzen) verringert werden, so daß eine annähernd realistische Einschätzung des Verhaltens des eigentlichen Systems (ohne Einwirkungen der Werkbankfunktionen) gemacht werden kann.

4.4.2.9 Optionen der Java-Werkbank

Es existieren zusätzlich folgende Optionen der Werkbank:

- Wahl des Verfahrens der Ressourcenverteilung
- Abschalten der Benutzung der Datenbasis der akquirierten Performanzprofile
- Abschalten der Kommunikation
- Festlegen der Ebenen der lokalen Ressourcenbäume, die zur Anzeige der Ressourcenverteilung mitprotokolliert werden sollen.

4.4.2.10 Integration anwendungsspezifischer Elemente

Es besteht die Möglichkeit, die Java-Werkbank um anwendungsspezifische Funktionen zu erweitern. Durch die objektorientierte Implementierung und den modularen Aufbau in Hinblick auf die Abtrennung der Java-Werkbank von der Anytime-System-Shell können zusätzliche Funktionalitäten relativ einfach integriert werden. Zur praktischen Vorgehensweise wird auf Abschnitt 4.5.4.3 verwiesen.

4.5 Implementation

In diesem Abschnitt werden die wichtigsten Aspekte der Implementation der Anytime-System-Shell und der zugehörigen Java-Werkbank vorgestellt und erläutert. Nach einigen allgemeinen Bemerkungen folgt die Beschreibung der Implementation der Anytime-System-Shell, der Kommunikation zwischen Anytime-System-Shell und Java-Werkbank und der Java-Werkbank selbst.

4.5.1 Allgemeines

Die verwendeten Programmiersprachen sind *Common Lisp* (Steele, 1990) unter Verwendung der objektorientierten Erweiterung *CLOS* zur Implementation der Anytime-System-Shell und Java 1.1 (Gosling et al., 1996) bei der Werkbank. Obwohl von der Lisp-Implementation abhängige Fähigkeiten wie *Multitasking-Funktionen* (s. Abschnitt 4.5.2.5) und *Advices* (s. Abschnitt 4.5.2.7) verwendet werden, ist das entwickelte System sowohl unter Liquid Common Lisp[5] (LIQUID, 1997) als auch unter Allegro Common Lisp (ACL, 1996) lauffähig[6]. Als Betriebssystem-Plattformen wurden Solaris 2.1, Solaris 2.5 und Linux benutzt. Die verwendeten Rechner waren vom Typ Sun Sparc Ultra 1, Sun Sparc Station 10 und IBM-PC.

Der Quellcode der Anytime-System-Shell umfaßt ca. 150 Kilobytes, der der Java-Werkbank ca. 200 Kilobytes.

Durch die Abtrennung der Werkbank von der Anytime-System-Shell besteht die Möglichkeit, die beiden Komponenten verteilt auf verschiedenen Rechnern laufen zu lassen. Dadurch kann eine Performanzsteigerung des gesamten Systems erzielt werden. In den oben beschriebenen Umgebungen wurde durch diese Maßnahme ungefähr eine Halbierung der Laufzeit für einen Durchlauf des in Kapitel 5 vorgestellten Beispielsystems erreicht.

Nach diesen allgemeinen Vorbemerkungen werden im restlichen Verlauf dieses Kapitels Aspekte der Implementation der verschiedenen Teile des vorgestellten Systems beschrieben.

4.5.2 Anytime-System-Shell

Die Anytime-System-Shell ist die Komponente, die für das Ressourcenmanagement verantwortlich ist (s. Abschnitt 4.3.3). Im folgenden werden die zu ihrer Implementation verwendeten Hilfsmittel und Techniken vorgestellt. Als erstes werden die benutzten Datenstrukturen erläutert.

4.5.2.1 Datenstrukturen

Die Modulkontrollen stellen die zentralen Koordinierungskomponenten der Anytime-System-Shell dar. Ihre Klassendeklaration lautet[7]:

```
(defclass mc ()

  ((name                          :initform "Generic")
   (subsystems                    :initform nil)
   (config                        :initform nil)
   (query                         :initform nil)
   (pot-result                    :initform nil)
```

[5] Liquid Common Lisp ist der Nachfolger von Lucid Common Lisp.

[6] Es ist recht einfach möglich, weitere Unterstützungen anderer Lisp-Implementationen zur Verfügung zu stellen (s. Abschnitte 4.5.2.6 und 4.5.2.7).

[7] Bei den folgenden Klassendeklarationen werden nur die initialen Werte der Slots angegeben. Mit Hilfe der hier nicht aufgeführten `:initarg`-Key-Parameter werden zur Laufzeit die aktuellen Werte eingetragen.

```
 (pot-output                  :initform nil)
 (current-quality             :initform 0.0)
 (pp-names                    :initform nil)
 (pp-current                  :initform nil)))
```

Die Bedeutungen der Slots dieser Datenstruktur sind:

name: Name des zugehörigen (Sub-)Systems.

subsystems: RAMIs eventuell vorhandener Subsysteme.

config: Konfigurationsdaten zum Aufbau des lokalen Ressourcenbaums.

query: Aufgabenstellung des zugehörigen Systems.

pot-result: Aktuelles (Zwischen-)Ergebnis des zugehörigen Systems.

pot-output: Aktuelle Ausgabe (als String).

current-quality: Qualität des aktuellen (Zwischen-)Ergebnisses.

pp-names: Namen der mitprotokollierten Performanzprofile.

pp-current: Mitprotkollierte Performanzprofile.

Dazu kommen die Slots, die dem Datenaustausch der Module untereinander dienen. Sie sind anwendungsspezifisch und müssen vom Entwickler festgelegt werden.

Die zentrale Datenstruktur der Anytime-System-Shell ist die `RAMI`-Klasse, deren Deklaration folgendermaßen aussieht[8]:

```
(defclass RAMI ()

  ((system-name             :initform "Generic")
   (parent                  :initform nil)
   (children                :initform nil)
   (interaction-process     :initform nil)
   (number-of-processes     :initform nil)
   (JAMES-processes         :initform nil)
   (free-JAMES-processes    :initform nil)
   (waiting-for-processes   :initform nil)
   (wait-for-transaction    :initform nil)
   (use-pp-database         :initform nil)
   (cycles-before-gc        :initform nil)
   (RAMI-scheduler-process  :initform nil)
   (minimum-process-quantum :initform nil)
   (tree                    :initform nil)
   (level-of-rd             :initform nil)
```

[8] Es sind hier nur die wichtigsten Slots aufgelistet. Die Klasse besitzt noch weitere, die nur zu internen Zwecken benötigt werden und an dieser Stelle nicht von Interesse sind.

```
(runnable-tasks          :initform nil)
(task-log                :initform nil)
(running-task            :initform nil)
(resources-per-cycle     :initform nil)
(current-cycle           :initform 0)
(available-processes     :initform 0)
(no-more-resources       :initform nil)
(no-tasks                :initform nil)
(before-start            :initform nil)
(after-end               :initform nil)
(started                 :initform nil)
(suspended               :initform nil)
(stopped                 :initform nil)
(status                  :initform 'init)
(mc                      :initform nil)))
```

Eine Instanz dieser Klasse beinhaltet alle innerhalb einer RAMI benötigten Daten. Nachfolgend werden die Inhalte und Funktionen der Slots kurz aufgelistet.

system-name: Name des zur RAMI gehörigen (Sub-)Systems.

parent: Referenz auf die übergeordnete RAMI (falls existent).

children: Referenzen auf die RAMIs eventuell vorhandener untergeordneter Subsysteme.

interaction-process: Lisp-Prozeß, der für die Kommunikation mit der Oberfläche zuständig ist (nur auf oberster Systemebene genutzt).

number-of-processes: Gesamtanzahl der JAMES-Prozesse (nur auf oberster Ebene genutzt).

JAMES-Processes: Liste aller JAMES-Prozesse (nur auf oberster Ebene genutzt).

free-JAMES-processes: Liste der freien JAMES-Prozesse (nur auf oberster Ebene genutzt).

waiting-for-processes: Flag, das gesetzt wird, wenn es einer RAMI nicht möglich ist, JAMES-Prozesse zu akquirieren (nur auf oberster Ebene genutzt).

wait-for-transaction: Flag, das gesetzt wird, wenn ein Task in eine Transaktion eintritt (nur auf oberster Ebene genutzt).

use-pp-database: Flag, das gesetzt wird, falls die Datenbasis der Performanzprofile bei der Bestimmung der Ressourcenverteilung benutzt werden soll (nur auf oberster Ebene genutzt).

cycles-before-gc: Anzahl der Zyklen, nach deren Ablauf eine kontrollierte Garbage Collection angestoßen wird (s. Abschnitt 4.5.2.9) (nur auf oberster Ebene genutzt).

RAMI-scheduler-process: Lisp-Prozeß, in dem der Scheduler der RAMI läuft.

minimum-process-quantum: Größe des minimalen Prozeßquantums.

tree: Referenz auf den lokalen Ressourcenbaum.

level-of-rd: Ebene des lokalen Ressourcenbaumes, die zur Ressourcenverteilung genutzt wird (falls das Verfahren unter Verwendung der Datenbasis der akquirierten Performanzprofile angewendet wird).

runnable-tasks: Liste aller lauffähigen Tasks.

task-log: Liste aller Tasks (lauffähige und beendete).

running-task: Laufender Task.

resources-per-cycle: Länge der Zyklen.

current-cycle: Nummer des aktuellen Zyklus.

available-processes: Anzahl der der RAMI im Rahmen der Ressourcenverteilung zugeteilten JAMES-Prozesse.

no-more-resources: Flag, das gesetzt wird, wenn sämtliche Ressourcen des aktuellen Zyklus aufgebraucht sind.

no-tasks: Flag, das gesetzt wird, falls eine RAMI zur Zeit keine lauffähigen Tasks besitzt.

before-start, after-end: Zeitpunkte, bevor bzw. nachdem ein lauffähiger Task seine ihm zugewiesene Berechnung durchführt bzw. durchgeführt hat.

started, suspended, stopped: Flags, die beim Start, bei einer Unterbrechung bzw. bei der Beendigung eines (Sub-)Systems gesetzt werden.

status: Status des (Sub-)Systems.

mc: Referenz auf die der RAMI zugehörige Modulkontrolle.

Mit Hilfe dieser Datenstruktur können die in Abschnitt 4.3.3 beschriebenen Verfahren realisiert werden.

Weitere wichtige Entitäten, die durch Klassen repräsentiert werden, sind virtuelle und reale Tasks. Sie sind beide von der generischen Klasse `task` abgeleitet. Die entsprechenden Deklarationen sehen wiefolgt aus:

```
(defclass task ()

  ((name                      :initform "Generic")
   (task-id                   :initform nil)
   (weight                    :initform 0.0)
   (global-weight             :initform 0.0)
   (args                      :initform nil)
```

```
   (processes               :initform 0)
   (number-of-runable-tasks :initform 0)
   (number-of-tasks         :initform 0)
   (depth                   :initform nil)
   (activation              :initform nil)
   (parent                  :initform nil)
   (children                :initform 0.0)
   (children-total-weight   :initform nil)))

(defclass virtual-task (task) ())

(defclass real-task (task)

  ((resources         :initform nil)
   (activation        :initform nil)
   (initial-function  :initform 'mc::calculate)))
```

Die Bedeutung der Slots der Klassen sind:

name: Name des Tasks.

task-id: Numerische ID des Tasks.

weight: Relatives Gewicht des Tasks.

global-weight: Globales Gewicht (bei virtuellen Tasks dient dieser Slot der Speicherung von Zwischenergebnissen zur Berechnung von globalen Gewichten der im lokalen Ressourcenbaum untergeordneten realen Tasks).

args: Argumente für die (untergeordneten) realen Tasks, die durch die Vorgängerknoten bestimmt werden (s. Abschnitt 4.3.1.2).

processes: Anzahl der Prozesse, die dem Unterbaum eines virtuellen Task bei der Ressourcenverteilung zugeordnet werden.

number-of-runable-tasks: Aktuelle Anzahl der lauffähigen Tasks im Unterbaum.

number-of-tasks: Gesamtanzahl der realen Tasks im Unterbaum.

depth: Tiefe des Tasks im lokalen Ressourcenbaum.

parent: Vorgängerknoten im lokalen Ressourcenbaum.

children: Nachfolgeknoten im lokalen Ressourcenbaum.

children-total-weight: Summe der relative Gewichte der Nachfolgeknoten im lokalen Ressourcenbaum (wird zur Ressourcenverteilung in Formel 4.4 benötigt).

resources: Berechnete Ressourcen innerhalb des aktuellen Zyklus.

activation: Liste der Aktivierungs- und Deaktivierungszeitpunkte des Tasks.

initial-function: Initiale Funktion `mc::calculate`, durch deren Aufruf die Berechnung des realen Tasks angestoßen wird. Diese Funktion muß vom Entwickler bereitgestellt werden und anhand der Diskriminatoren einen Dispatch auf das zugehörige Berechnungsverfahren durchführen.

Damit wurden die wichtigsten Datenstrukturen vorgestellt, die innerhalb der Anytime-System-Shell verwendet werden. In den nächsten Abschnitten werden verschiedene Aspekte der Implementation des Schedulingvorgangs und der dazu benötigten Datenstrukturen dargestellt.

4.5.2.2 Aufbauen des lokalen Ressourcenbaums

Voraussetzung des Schedulings ist der Aufbau des lokalen Ressourcenbaums der RAMI. Dazu wird in der zugehörigen Modulkontrolle die homogene hierarchische Systemstruktur in Form von Konfigurationsdaten abgelegt. Der Slot `config` beinhaltet die Informationen über die Ebenen des lokalen Ressourcenbaumes in Form einer Liste, deren Elemente weitere Slots der Modulkontrolle repräsentieren. Diese enthalten jeweils die Beschreibungen der einzelnen Ebenen.

Dieses Konzept wird im folgenden an einem Beispiel verdeutlicht. Enthalte der Slot `config` die Liste `(ebene1 ebene2 ebene3)`, dann müssen die drei Slots `ebene1`, `ebene2` und `ebene3` vom Entwickler des Gastsystems in die Modulkontrolle eingefügt werden. Enthalten dann diese Slots wiederum folgende Listen:

- `ebene1: (diskriminator11 diskriminator12)`
- `ebene2: (diskriminator21 diskriminator22 diskriminator23)`
- `ebene3: (diskriminator31 diskriminator32)`

Dann hat der zugehörige lokale Ressourcenbaum, das in Abbildung 4.24 dargestellte Aussehen (dabei entspricht `d`*ij* dem Diskriminator `diskriminator`*ij*).

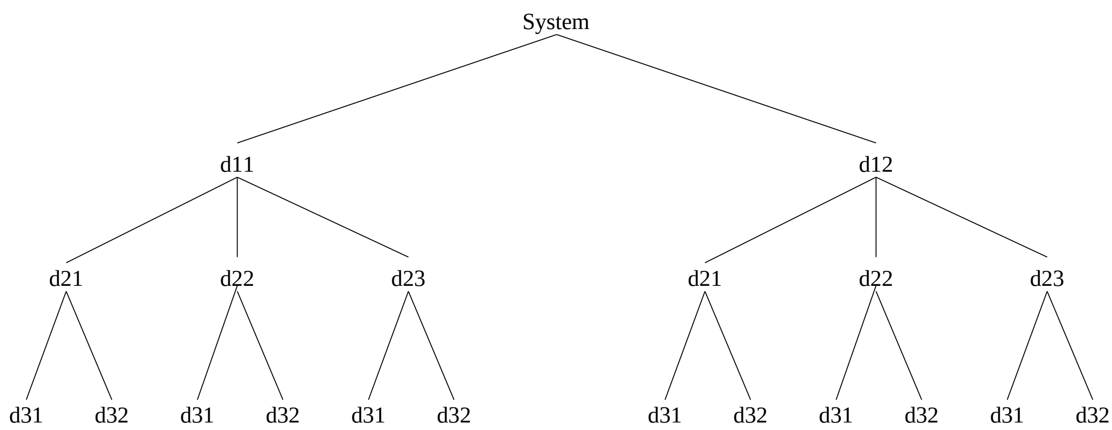

Abbildung 4.24: Beispiel zum Aufbau des lokalen Ressourcenbaums

4.5.2.3 Festlegung der relativen Gewichte

Eine weitere wichtige Voraussetzung ist die Festlegung der relativen Gewichte der virtuellen und realen Tasks des lokalen Ressourcenbaums. Da diese vom Entwickler des Gastsystems bestimmt werden, wird eine Schnittstelle in Form der Lispfunktion `get-weight` bereitgestellt. Diese anwendungsspezifische Funktion muß vom Entwickler so konstruiert werden, daß sie zu jeder Entität, die im lokalen Ressourcenbaum auftritt, das zugehörige Gewicht zurückliefert. Im Beispiel des vorhergehenden Abschnitts muß `get-weight` für alle Diskriminatoren `diskriminator`*xy* eine reelle Zahl als relatives Gewicht zur Verfügung stellen.

4.5.2.4 Schedulerebenen

Während des Laufs eines mit Hilfe der Anytime-System-Shell konstruierten Gesamtsystems, sind, wie in Abbildung 4.25 aufgezeigt, drei verschiedene Scheduler aktiv.

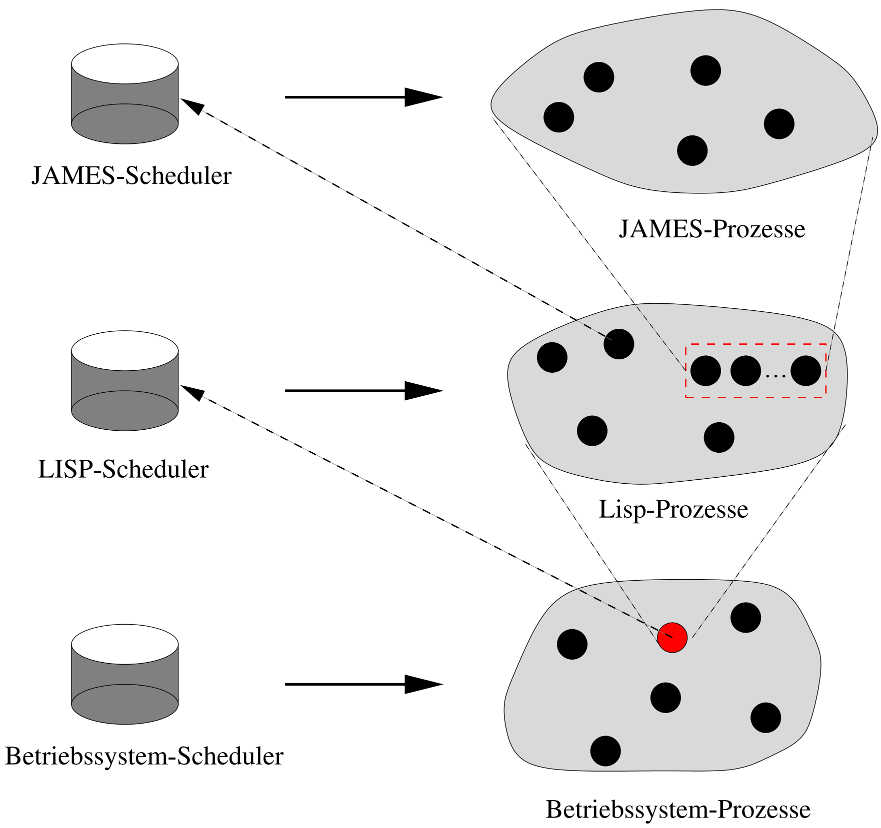

Abbildung 4.25: Schedulerebenen

Auf unterster Ebene arbeitet der *Betriebssystemscheduler* des Solaris- oder Linux-Kernels. Einer der von ihm bearbeiteten Betriebssystemprozesse stellt das benötigte Lisp-Image dar. In diesem läuft wiederum das Gesamtsystem als eine Menge von Lisp-Prozessen. Hier werden auch die von der Anytime-System-Shell genutzten Lisp-Prozesse verwaltet. Der *Scheduler der Anytime-System-Shell* nutzt den *Lisp-Scheduler*, in dem er seine eigenen Datenstrukturen und Funktionen, wie z.B.

die JAMES-Prozesse und die Funktion zu deren Unterbrechen, auf den vom Lisp-Scheduler bereitgestellten aufbaut. Dies ist das Thema der nächsten beiden Abschnitte.

4.5.2.5 JAMES-Prozesse

Ein Konstrukt der Anytime-System-Shell, das auf einer vom Lisp zur Verfügung gestellten Datenstruktur aufbaut, ist der JAMES-*Prozeß*.

Als Grundlage wird auf Lisp-Prozesse[9] zurückgegriffen, die den Charakter eines *Threads* besitzen (s. (Silberschatz & Galvin, 1994)). Im Gegensatz zu echten Prozessen, wie sie im Forschungsgebiet der Betriebssysteme verstanden werden, besitzen diese einen gemeinsamen Speicher, das bedeutet, sie können auf gemeinsame Daten zugreifen. Diese Tatsache birgt das Problem der Synchronisation solcher Zugriffe in sich. Im mit Hilfe der Anytime-System-Shell konstruierten System werden zur Vermeidung dieser Probleme Transaktionen verwendet (s. Abschnitt 4.1.4).

Die Deklaration eines JAMES-Prozesses sieht folgendermaßen aus:

```
(defclass JAMES-process ()

  ((process-status   :initform nil)
   (internal-process :initform nil)
   (task             :initform nil)))
```

Ein JAMES-Prozeß besitzt als internen Prozeß (Slot `internal-process`) einen Lisp-Prozeß, den er für die ihm zugeordneten Berechnungen nutzt. Dieser wird im Rahmen der Initialisierung des JAMES-Prozesses mit einer generischen Funktion belegt, die den Lisp-Prozeß solange in den Wartezustand versetzt, bis dem JAMES-Prozeß eine Aufgabe zugeordnet wird. Dazu wird der Slot `process-status` benötigt. Er kann die Werte "initialized" oder "running" annehmen. Hat der JAMES-Prozeß noch keine Aufgabe, d.h. ist er noch keinem Task zugeordnet, so befinden er und damit auch der Lisp-Prozeß sich im Zustand "initialized". Andernfalls ist er "running". Der Übergang zwischen diesen beiden Zuständen wird dadurch erreicht, daß eine Reinitialisierung des Lisp-Prozesses erfolgt, wenn dem JAMES-Prozeß ein Task zugewiesen wird. Die generische Funktion des Lisp-Prozesses erkennt dann, daß sich der JAMES-Prozeß im Zustand "initialized" befindet und geht in den Arbeitsmodus über. Durch den Aufruf seiner initialen Funktion wird die ihm zugewiesene Aufgabe bearbeitet. Ist der JAMES-Prozeß einem Task zugeordnet, so verweist der Slot `task` auf diesen. Andernfalls befindet der JAMES-Prozeß in der Liste der freien JAMES-Prozesse, die dem Gesamtsystem zur Verfügung stehen.

4.5.2.6 Generische Ansteuerung der Lisp-Prozeßfunktionen

Um eine möglichst universelle Einsetzbarkeit der konstruierten Systeme bezüglich unterschiedlicher Lisp-Implementationen zu erzielen, wurde bei der Implementierung der Anytime-System-Shell eine generische Ansteuerung der Lisp-Prozeßfunktionen

[9] Die konkrete Ausprägung dieser Lisp-Prozesse ist implementationsabhängig, da für sie keine Standardisierung vorliegt.

verwendet. Alle diese Funktionsaufrufe laufen über eine Zwischenstufe, den *Prozeß-Handler*, der die generischen Namen in implementationsspezifische Funktionsnamen umsetzt (s. Abbildung 4.26(a)).

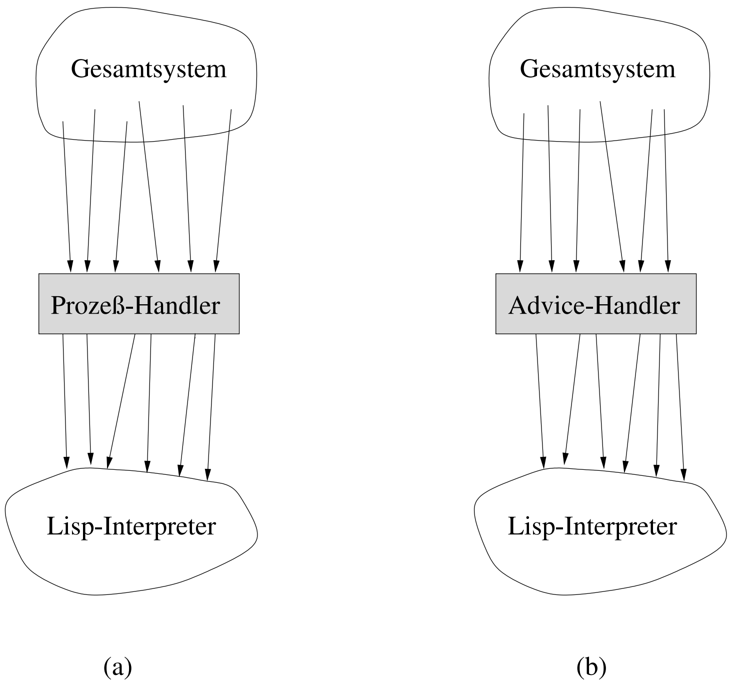

Abbildung 4.26: Prozeß- und Advice-Handler

Bisher werden das Liquid Common Lisp sowie das Allegro Common Lisp unterstützt. Durch dieses flexible Konzept ist es sehr einfach möglich, das System um die Unterstützung weiterer Lisp-Implementationen zu erweitern.

4.5.2.7 Generische Implementation der Transaktionen

Ebenso wurde auch die Implementation der Transaktionsverwaltung auf generische Weise gelöst. Auch hier werden generische in implementationsspezifische Befehle durch eine Zwischeneinheit, den *Advice-Handler*, transformiert (s. Abbildung 4.26(b)). Die Transaktionen sind unter Verwendung des *Advice*-Konzeptes realisiert. Dabei handelt es sich um die Möglichkeit, das Verhalten einer Lisp-Funktion zu modifizieren, indem ihr ein “Rat” in Form von zusätzlichen Code zugeordnet wird (s. (LIQUID, 1997; ACL, 1996)).

Dieser “Rat” besteht an dieser Stelle in dem Setzen und Löschen des Flags `wait-for-transaction` der obersten `RAMI`-Instanz vor dem Eintritt in bzw. dem Austritt aus dem die Transaktion ausmachenden Code. Damit kann die Anytime-System-Shell zu jedem Zeitpunkt feststellen, ob sich die Berechnung eines Tasks innerhalb einer Transaktion befindet.

Unter Ausnutzung dieses Slots wird die korrekte Ausführung einer Unterbrechung garantiert. Tritt eine Unterbrechungsanforderung auf, so wird diese solange verzögert bis die noch zu beendende Transaktion verlassen wird. Um den Eintritt in eine direkt anschließende Transaktion zu verhindern, wird `wait-for-transaction` bis die Unterbrechung ausgeführt wurde auf einen Wert gesetzt, der den Zustand der Beendigung einer Transaktion anzeigt. Abbildung 4.27 verdeutlicht diese Vorgehensweise.

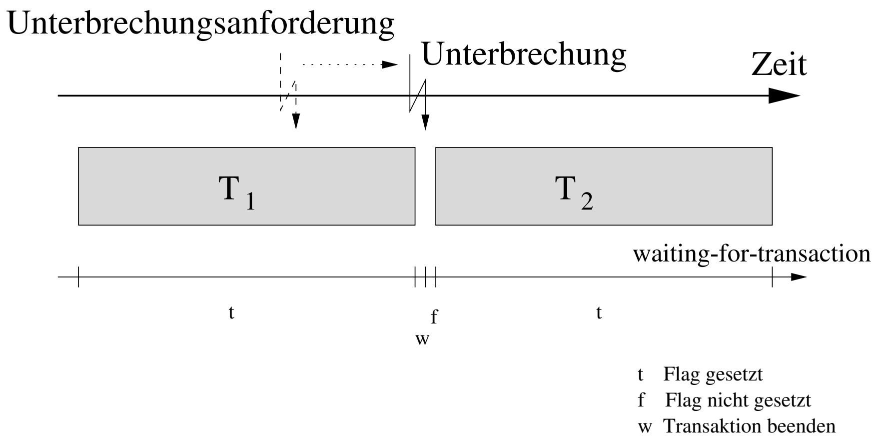

Abbildung 4.27: Behandlung einer Unterbrechungsanforderung während einer Transaktion

Es wurde hier auf die Verwendung von before-, after- oder around-Lisp-Methoden verzichtet, damit diese vom Entwickler des Gastsystems ohne Einschränkungen genutzt werden können.

4.5.2.8 Zeitmessung

Die Zeitmessung zur Bestimmung der echten Laufzeit eines Tasks im Rahmen des Schedulings der Anytime-System-Shell basiert auf der Verwendung der Lisp-Funktion `get-internal-real-time`. Mit ihrer Hilfe wurde ein Timer programmiert, der die Möglichkeit der Unterbrechung bietet. Die Nutzung dieser Lisp-Funktion stellt einen problematischen Sachverhalt dar, da die im Liquid Common Lisp und Allegro Common Lisp zur Verfügung stehende Genauigkeit von `get-internal-real-time` nicht unbedingt die gewünschten Anforderungen erfüllt, die an die Konstruktion eines Realzeitsystems gestellt werden.

4.5.2.9 Kontrolle des Garbage-Collection-Mechanismus

Ein weiteres Problem, das bei der Implementation der Anytime-System-Shell zur Konstruktion von Realzeitsystemen auftritt, ist das unkontrollierbare Auftreten von Garbage-Collections zum Aufräumen des Speicherinhaltes. Diese sind sehr zeitintensiv und können die gesamte Performanz des Systems soweit beeinträchtigen, daß das Ergebnis bezüglich der Zeit unbrauchbar wird. Da es keine sinnvolle Möglichkeit gibt, den Mechanismus der Garbage-Collections auszuschalten, wird versucht, durch kontrolliertes Anstoßen zu ausgewählten Zeitpunkten, ein unkontrolliertes Auftreten zu verhindern. So wird nach jeder beendeten Bearbeitung einer Aufgabenstellung eine Garbage-Collection durchgeführt, in der Absicht, damit den Speicher soweit aufzuräumen, daß der nächste Durchlauf ohne weitere Verzögerung ausgeführt werden kann. Auch bei einer externen Unterbrechung durch den Benutzer über die

Java-Werkbank findet eine Garbage-Collection statt. Zu diesem Zeitpunkt ist die Zeitmessung des Systems angehalten und der Benutzer mit der kognitiven Verarbeitung des Zwischenergebnisses beschäftigt. Deshalb hat eine Garbage-Collection an dieser Stelle keine negativen Auswirkungen. Ist das konstruierte Gesamtsystem zu umfangreich und kommt es trotz dieser Vorkehrungen zu unkontrollierten Garbage-Collections, so existiert die Möglichkeit, die sogenannte *Garbage-Collection-Frequenz* anzugeben. Sie bestimmt eine Anzahl von Zyklen, nach deren Ablauf eine kontrollierte Garbage-Collection angestoßen wird. Damit kann dann versucht werden, ein zu langes bzw. zu häufiges Auftreten zu verhindern.

Anschließend wird die Kommunikation als Bindeglied zwischen der Anytime-System-Shell und der Java-Werkbank bezüglich ihrer Implementation thematisiert.

4.5.3 Kommunikation zwischen der Java-Werkbank und der Anytime-System-Shell

Zur Implementation der Kommunikation zwischen der Anytime-System-Shell und der Java-Werkbank wurde auf die Systeme CHANNELS (Cooperating Heterogeneous Agents for a Natural Language System) und JALCOSTE (JAva Lisp COnnection SysTEm) zurückgegriffen.

JALCOSTE stellt ein System zur Kommunikation zwischen einem Lisp-Image und einem Java-Applet dar. Es nutzt auf der Lisp-Seite die vom System CHANNELS bereitgestellten Funktionalität.

Das von Alassane Ndiaye vorgestellte System CHANNELS (Ndiaye, 1998) dient der Bereitstellung einer Multi-Agenten-Architektur für ein zu entwickelndes System. Es entstand im Rahmen der Konstruktion des PRACMA[10]-Projekts (Jameson et al., 1994).

Unter Agenten werden in diesem Zusammenhang einzelne Systemkomponenten verstanden, die verschiedenen Teilaufgaben dienen. CHANNELS bietet ihnen die Möglichkeit, mittels Austausch von Nachrichten, flexibel zu interagieren. Dies geschieht durch ein *kommunikationsakt-basiertes Protokoll*, das im folgenden beschrieben wird.

Für jeden Agenten wird festgelegt, wer ihm Nachrichten senden bzw. wer von ihm gesendete Nachrichten empfangen darf. Dies wird unter dem Begriff *Bekanntschaften* des Agenten zusammengefaßt. Zum Empfangen und Versenden besitzt jeder Agent einen *Puffer* in Form einer Liste, in dem die Nachrichten bis zu ihrer Verarbeitung gespeichert werden. Es existieren im wesentlichen drei Typen von Nachrichten: ein `Inform` repräsentiert unaufgefordert versendete Informationen, ein `Ask` stellt eine Anfrage an einen Agenten dar, der das Ergebnis mittels eines `Reply` mitteilt.

Es wird zwischen *synchronem* und *asynchronem* Kommunikationsmodus unterschieden. Der erste Fall liegt vor, wenn der die Nachricht verschickende Agent nach dem Senden in einen Wartezustand übergeht und solange seine Berechnungen unterbricht, bis er von der Antwort auf die gestellte Anfrage wieder aktiviert wird und dann seine Berechnungen fortsetzen kann. Der asynchrone Kommunikationsmodus bewirkt nach dem Versenden keine Unterbrechung der Berechnungen des Agenten.

[10] PRocessing Arguments among Controversially-Minded Actors

Er arbeitet weiter und verarbeitet das Ergebnis auf seine Anfrage erst später, wenn es bei ihm vorliegt.

Channels bietet die Möglichkeit die Agenten über mehrere vernetzte Rechner zu verteilen. Damit kann durch Aufteilen der Rechenaufgaben unter Umständen eine signifikante Steigerung der Gesamtperformanz des Systems, das die Fähigkeiten von Channels nutzt, erzielt werden.

Im Vergleich zur vorgestellten Anytime-System-Shell bietet Channels nicht die Möglichkeit, Ressourcenbeschränkungen zu berücksichtigen. Es können den einzelnen Agenten keine Ressourcen für ihre Berechnungen zugeteilt werden. Damit eignet sich Channels nicht für die Anwendung in Anytime-Systemen.

Ein unter Verwendung von Jalcoste konstruiertes System besteht aus drei Teilen:

- *Lisp-System*
- *Masterapplet* (Java)
- *Verteilerserver*

Abbildung 4.28 zeigt den Datenfluß zwischen dieses Teilen. Die gesamte Kommunikation wird über den Verteilerserver abgewickelt. Er stellt die zentrale Verwaltungseinheit dar. Er verteilt bei ihm ankommende Nachrichten an die entsprechenden Applets bzw. Lisp-Systeme. Die Schnittstelle zwischen dem eigentlichen Lisp-System und dem Verteilerserver wurde unter Verwendung der Funktionalität von Channels realisiert. Es existiert je ein Agent, der für das Verschicken bzw. für das Empfangen von Nachrichten zuständig ist.

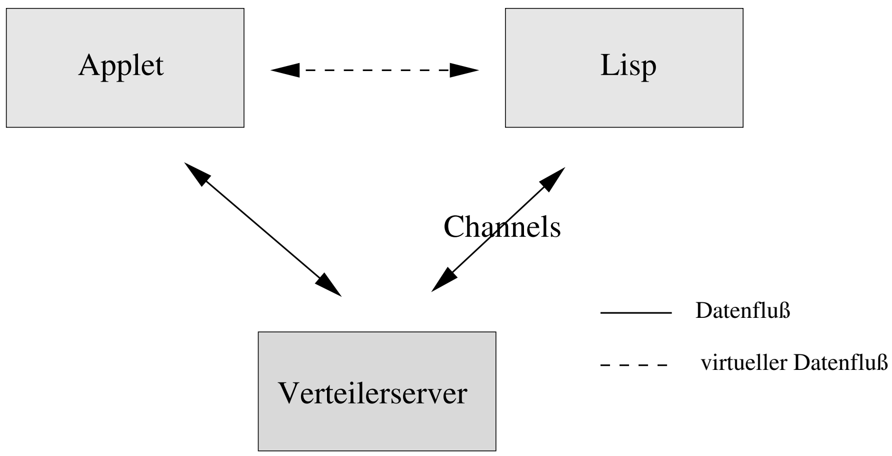

Abbildung 4.28: Datenfluß bei Verwendung von Jalcoste

Gestartet wird ein solches System durch das Betätigen eines ihm zugewiesenen Buttons einer Web-Seite. Dadurch nimmt das Applet Kontakt mit dem Verteilerserver auf und teilt diesem mit, welches Lisp-System hochgefahren werden soll. Ist das Lisp-System bereit, wird dies dem Applet vom Verteilerserver mitgeteilt. Ab diesem Zeitpunkt können Daten zwischen den beiden Seiten übertragen werden.

Jalcoste stellt dem Benutzer eine Bibliothek vom Lisp-Funktionen zur Verfügung, um Aktionen auf der Java-Seite zu veranlassen, insbesondere können Fenster, Buttons und andere Elemente der Java-AWT[11] von der Lisp-Seite aus verwaltet werden.

Um den Overhead der Kommunikation, der durch diese Verwaltung anfällt, zu vermeiden, existiert eine weitere Möglichkeit der Erzeugung und Bearbeitung einer Java-Oberfläche. Das Masterapplet kann um die gewünschten anwendungsspezifischen Funktionen auf der Java-Seite erweitert werden. Dies geschieht unter Verwendung von *Komplexen*. Dazu stellt JALCOSTE eine Klasse `Complex` bereit, deren Instanzen die Werkzeuge bereitstellen, eigene Nachrichten an die Lisp-Seite zu schicken bzw. von ihr zu erhalten und zu verarbeiten. Eine solche Instanz wird unter Verwendung einer von `ComplexGenerator` abgeleiteten Klasse erzeugt. Beide genannten Klassen müssen vom Entwickler durch die Konstruktion einer abgeleiteten Klasse an das betrachtete System angepaßt werden. Auf der Lisp-Seite können mittels einer von JALCOSTE bereitgestellten Funktion die ankommenden Daten entgegengenommen und zur Bearbeitung an das Lisp-System weitergereicht werden.

Eine ausführliche Beschreibung der Anwendung von JALCOSTE findet sich in (Kern, 1998).

4.5.3.1 Kommunikationsprotokoll

Die bidirektionale Kommunikation zwischen der Java-Werkbank und der Anytime-System-Shell geschieht durch das Versenden von Nachrichten unter Verwendung der bereitgestellten Möglichkeiten von JALCOSTE. Zuerst werden die Erstellung und die möglichen Ausprägungen der Nachrichten der Werkbank an die Anytime-System-Shell dargestellt. Im Anschluß daran wird die umgekehrte Richtung betrachtet.

4.5.3.1.1 Nachrichten der Java-Werkbank an die Anytime-System-Shell

Um eine Nachricht an die Anytime-System-Shell senden zu können, muß sie einem bestimmten Typ angehören. Diese Typen werden in der Funktion `getEvents()` der Klasse `JamesComplexGenerator` (s. Abschnitt 4.5.3) deklariert. Tabelle 4.1 listet alle vorhandene Typen samt ihrer Bedeutung auf.

Diese unterschiedlichen Typen determinieren die Vorgehensweise der Bearbeitung der in der Nachricht enthaltenen Daten auf der Lisp-Seite. Die Konstruktion einer Nachricht wird am folgenden Beispiel eines Unterbrechungssignals aufgezeigt:

```
String[] n = new String[1];
n[0] = "SUSPEND";
MixValue[] mv = new MixValue[1];
mv[0] = new MixValue();
mv[0].valueType= MixValue.STRING;
mv[0].stringVal = "SUSPEND";

ComplexEvent ce = new ComplexEvent(this, "SUSPEND", n, mv);
listener.complexEvent(ce);
```

Eine Instanz der Klasse `MixValue` nimmt die Daten der Nachricht auf. In diesem Fall handelt es sich um den String “SUSPEND”. Mit ihrer Hilfe wird eine Instanz

[11] Abstract Windowing Toolkit: Klassenbibliothek zur Verwaltung von GUI-Elementen.

Typ	Bedeutung
START	Start-Signal
SUSPEND	Unterbrechungssignal
STOP	Stop-Signal
INIT	Initialisierungssignal
RETRIEVE	Retrieve-Signal
RTASKINFO	Anforderung der Daten eines realen Tasks
JTON	Übertragung der Daten eines lokalen Ressourcenbaumes beginnen
JTOFF	Übertragung der Daten eines lokalen Ressourcenbaumes beenden
JTUPDATE	Veränderungen von Parametern übernehmen
STATS	Anforderung von statistischen Daten der Tasks
VTASKINFO	Anforderung der Daten eines virtuellen Tasks
SUBID	Anforderung der Daten eines Subsystems
SUBJT	Anforderung der Daten des lokalen Ressourcenbaums eines Subsystems
PPON	Anschalten der Ressourcenberechnung unter Zuhilfenahme der Datenbasis
PPOFF	Ausschalten der Ressourcenberechnung unter Zuhilfenahme der Datenbasis
COMON	Wiederanschalten großer Teile der Funktionen der Werkbank
COMOFF	Ausschalten großer Teile der Funktionen der Werkbank
DIST1	Wahl der ersten Alternative der Ressourcenverteilung
DIST2	Wahl der zweiten Alternative der Ressourcenverteilung
DOMAIN	Anwendungsspezifisches Signal (s. Abschnitt 4.5.4.3)
QUIT	Beendet des Gesamtsystems

Tabelle 4.1: Typen der Nachrichten der Java-Werkbank an die Anytime-System-Shell

von `ComplexEvent` erzeugt, die in diesem Beispiel den Namen "SUSPEND" hat und von dem aktuellen Objekt `this`, einer `JamesComplex`-Instanz, erzeugt wird. Die eigentliche Versendung durch das JALCOSTE-System geschieht durch den Aufruf der Funktion `ComplexEvent()` einer Instanz der Klasse `ComplexEventListener`, die in diesem Beispiel durch die Variable `listener` repräsentiert wird. Für eine ausführlichere Beschreibung der Verwendung und inneren Abläufe des JALCOSTE-Systems wird auf die zugehörige Dokumentation verwiesen (Kern, 1998).

Ankommende Nachrichten können auf der Lisp-Seite mit Hilfe der Funktion `jalcoste::next-event` abgefragt werden. Sie gibt eine Liste zurück, deren erstes Element den Typ der Nachricht beinhaltet. Damit ist es möglich, die einzelnen Nachrichtentypen von unterschiedlichen Funktionen bearbeiten zu lassen.

4.5.3.1.2 Nachrichten der Anytime-System-Shell an die Java-Werkbank

Die umgekehrte Kommunikationsrichtung, von der Anytime-System-Shell ausgehend an die Java-Werkbank gerichtet, wird durch Versenden von Nachrichten mittels der JALCOSTE-Lisp-Funktion `jalcoste::change-complex` durchgeführt. Dazu werden Listen aus Paaren bestehend aus einem Identifier und dem zugehörigen Datum aufgebaut, die als Parameter dieser Funktion übergeben werden. Der folgenden Lisp-Code zeigt hierfür ein Beispiel:

```
(jalcoste::change-complex
       mod
       com
       :values (list
                'CT (list (task-id (current-task rami))
                          (subsys-id rami))
                'WEI (list (weight (current-job rami))
                           (subsys-id rami))
                'RES (list (resources (current-job rami))
                           (subsys-id rami)))))
```

Die beiden ersten Parameter dienen internen Zwecken von JALCOSTE, die hier nicht von Interesse sind. Die mit Hilfe des Key-Parameters `:values` spezifizierte Liste stellt die angesprochene Übergabe der Nachricht dar. Diese Liste besteht in diesem Beispiel aus drei Identifier-Datum-Paaren. Der Identifier `CT` gibt an, daß der anschließende Wert die ID des aktuellen Tasks der durch `rami` spezifizierten RAMI repräsentiert. Durch `WEI` und `RES` werden das zugehörige relative Gewicht und die zugeordneten Ressourcen in die Nachricht integriert.

Die so übermittelten Daten werden auf der Java-Seite innerhalb von JALCOSTE in Instanzen der Klasse `MixValues` eingetragen, über die ein Zugriff der Java-Werkbank möglich ist.

4.5.4 Java-Werkbank

Nachdem im vorhergehenden die Implementation der Anytime-System-Shell und der Kommunikation zwischen ihr und der Java-Werkbank erläutert wurden, ist die Implementation des zugehörigen Java-Applets das Thema dieses Abschnitts.

4.5.4.1 Implementation als Java-Applet

Um eine möglichst flexible Ansteuerung der entwickelten Systeme zu gewährleisten, ist die Java-Werkbank als Applet realisiert. Durch seine Einbindung in eine Web-Seite ist es möglich, das System weitgehend ortsunabhängig zu starten und zu kontrollieren. Ein Beispiel des zur Einbindung benötigten HTML[12]-Codes sieht folgendermaßen aus:

```
<HTML>
<HEAD>
<TITLE>Applet der Java-Werkbank</TITLE>
</HEAD>
<BODY>
<APPLET codebase="file:/WW/lucifer/REAL/new-systems/james/java"
        code="DE.uni_sb.jalcoste.core.DSApplet.class"
        width="300"
        height="20">
<PARAM name="SYSTEM" value="JAMES">
<PARAM name="STATUS" value="STANDARD">
<PARAM name="PORT" value="5668">
<PARAM name="BUTTON" value="Los gehts">
<PARAM name="COMPLEX" value="DE.uni_sb.james JamesComplexGenerator">
</APPLET>
</BODY>
</HTML>
```

Auf der entsprechenden Web-Seite erscheint ein Button, der zum Starten des Systems dient. Wird er gedrückt, so erscheint die Oberfläche der Java-Werkbank als neues Fenster auf dem Bildschirm. Dort können dann die in Abschnitt 4.4.2 beschriebenen Funktionen benutzt werden.

In Tabelle 4.2 sind die wichtigsten zur Implementation der Java-Werkbank und ihrer Funktionen erzeugten Klassen mit ihrer jeweiligen Aufgabe, den Klassen, die sie benutzen, den Klassen, von denen sie abgeleitet sind und den Interfaces, die sie implementieren, aufgeführt.

In den beiden abschließende Abschnitten dieses Themengebietes werden die implementatorischen Aspekte zweier wichtiger Funktionen der Werkbank ausführlicher beschrieben.

4.5.4.2 Darstellung der Performanzprofile

Die Darstellung der Performanzprofile, die das Verhalten eines Systemlaufs repräsentieren, ist eine der zentralen Leistungen der Java-Werkbank. Die dazu benötigte Vorgehensweise, sowohl auf der Seite der Anytime-System-Shell, als auch auf der Seite der Java-Werkbank, werden im folgenden erläutert.

[12] HyperText Markup Language. Sie ist eine Scriptensprache zur Erstellung von Hypertext-Dokumenten, insbesondere Web-Seiten.

Klasse	Funktionen	benutzt von	abgeleitet von	implementiert Java-Interface
JamesComplex	Hauptklasse der Java-Werkbank, Darstellung der Hauptfensters, Verschicken und Empfangen von Signalen an und von der Anytime-System-Shell	JamesComplex-Generator	Frame	Complex
JamesComplexGenerator	Erstellung des James-Complexes, Deklaration der möglichen Signale von der Java-Seite an die Lisp-Seite		ComplexGenerator	
JamesTasks	Anzeige der Ressourcenverteilung mit Hilfe des zweidimensionalen Balkendiagramms, Verwaltung der dazu benötigten Daten	JamesComplex, SubSystem	Frame	AdjustmentListener
JamesTree	Anzeige des lokalen Ressourcenbaums, Modifikationsmöglichkeit der Parameter der einzelnen Tasks	JamesComplex, SubSystem	Frame	AdjustmentListener
JamesNode	Repräsentation eines virtuellen oder realen Tasks, Anzeige der zugehörigen Daten	JamesTree	Button	
JamesStats	Anzeige statistischer Daten der Tasks	JamesComplex, SubSystem	Frame	MouseListener
JamesStatus	Graphische Darstellung des Systemzustands	JamesComplex	Canvas	
SubsystemFrame	Anzeige der Daten des Subsystems, Parametermodifikation, Zugriff auf lokalen Ressourcenbaum, Diagramm der Ressourcenverteilung und Statistiken der Tasks	JamesComplex	Frame	ActionListener
SubSystem	Repräsentation eines Subsystems	SubSystemFrame, JamesComplex		
HedgeGUI	Anzeige der Performanzprofile	JamesComplex	Applet	
ParameterPanel	Anzeige der Optionen und deren Modifikation	JamesComplex	Panel	ItemListener

Tabelle 4.2: Übersicht über die Klassen der Implementation der Java-Werkbank

Wie in Abschnitt 4.3.2.4 beschrieben, werden die Performanzdaten eines Systemlaufs in den Modulkontrollen der RAMIs mitprotokolliert. Diese Werte werden dann bei einer Unterbrechung oder Beendigung des Systemlaufs benutzt, um die Performanzprofile zu erzeugen. Diese werden in eine Datei geschrieben, auf die die entsprechende Klasse `HedgeGUI` der Java-Werkbank zugreift und sie auf Wunsch des Benutzers anzeigt.

Hier wird auf eine Benutzung der beschriebenen Kommunikation verzichtet, da es sich um so große Datenmengen handeln kann, die eine unakzeptable Belastung des Systems bewirken würden. Durch den "Umweg" über eine Datei wird sichergestellt, daß eine Unterbrechung des Systems jederzeit aufgehoben werden kann und nicht durch eine Übertragung der Daten der Performanzprofile blockiert wird.

Diese Anzeigemöglichkeit der Performanzprofile entstand im Rahmen der Diplomarbeit von Christian Kray (Kray, 1998).

4.5.4.3 Anwendungsspezifische Elemente

Es ist möglich, die Java-Werkbank um weitere (anwendungsspezifische) Funktionen zu erweitern. Bei der Konstruktion des Objektlokalisationssystems, wie in Kapitel 5 beschrieben, wurde zum Beispiel eine Anzeigemöglichkeit von VRML[13]-Dateien zur 3D-Darstellung von Anwendungsdomänen hinzugefügt. Abbildung 4.29 zeigt als Beispiel hierfür eine Büroumgebung.

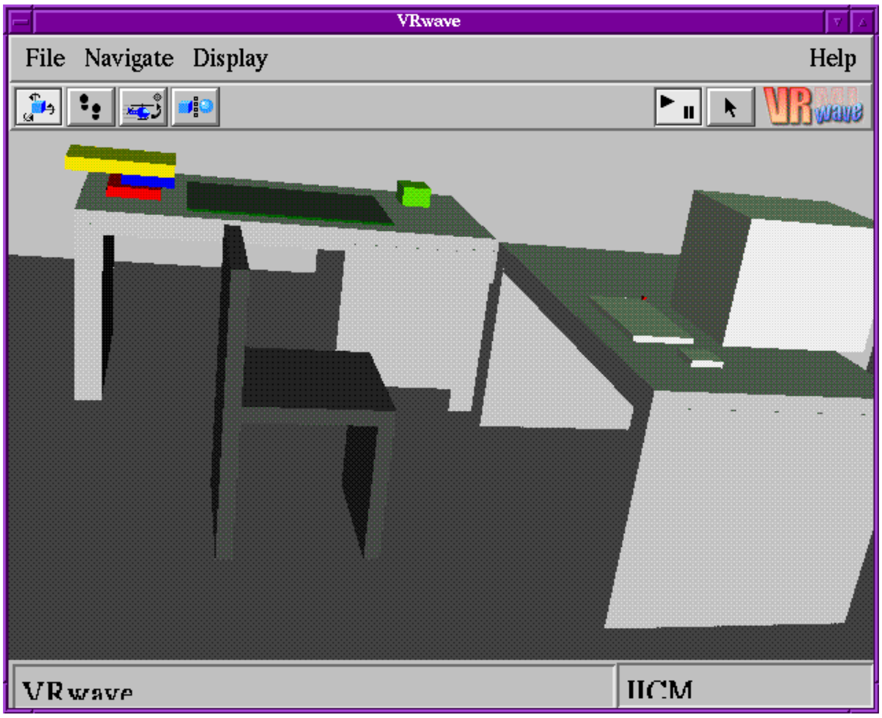

Abbildung 4.29: Beispiel einer VRML-Welt

Durch das Drücken des zugehörigen Buttons wird von der Java-Werkbank ein Signal an die Lisp-Seite gesendet, das das Starten des Programms VRWAVE (VR-

[13] Virtual Reality Modelling Language (Bell, Pesce & Parisi, 1995)

WAVE, 1997) mit dem entsprechenden VRML-File bewirkt. Dies geschieht mit folgendem Lisp-Code unter Verwendung der Lisp-Funktion `run-programm`:

```
(defun start-vrwave()
  "
This function starts vrwave to show VRML-Domains
"
  (run-program "rsh"
               :arguments
               (list *VRWAVE-host*
                     "tset xterm"
                     (concatenate
                      'string
                      "csh; setenv DISPLAY " *display*
                      ":0.0; vrwave "
                      *VRML-Path*
                      *VRML-File*))
               :wait nil;; do not wait
               :input :stream
               :error-output "/dev/null"
               :if-error-output-exists :append))
```

Das Aussehen des auf der Java-Seite benötigten Codes zum Erzeugen des Signals ist analog zu dem in Abschnitt 4.5.3.1.1 beispielhaft dargestellten.

Die Funktionsweisen des in diesem Kapitel vorgestellten Systems zum Ressourcenmanagement in Anytime-Systemen werden im nächsten Kapitel noch einmal anhand einer praktischen Anwendung dargestellt.

Kapitel 5

Anwendung: Konstruktion eines ressourcenadaptierenden Objektlokalisationsagenten

In diesem Kapitel wird am Beispiel der Entwicklung eines Systems zur Objektlokalisation die Verwendung der Anytime-System-Shell zur Konstruktion von Realzeitsystemen beschrieben.

Nach einigen einführenden Bemerkungen bezüglich des vorgestellten Beispiels wird die Architektur und deren praktische Implementation des in diesem Kapitel konstruierten Anytime-Systems dargestellt. Anhand einer Beispielanfrage werden verschiedene interessante Aspekte der Funktionsweise des entwickelten Objektlokalisationssystems verdeutlicht. Den Abschluß bildet eine Untersuchung der Einflüsse von Veränderungen verschiedener Parameter des Anytime-Systems auf sein Performanzverhalten.

5.1 Ein beschränkt-optimaler Lokalisationsagent

5.1.1 Allgemeines

Das unter Verwendung der Anytime-System-Shell realisierte System BOLA (<u>Be</u>schränkt-<u>O</u>ptimaler <u>L</u>okalisations<u>A</u>gent) (Blocher, 1999) dient der Beantwortung von *Wo*-Fragen eines Benutzers bezüglich eines Objektes der implementierten Szenarien. Diese Fragen werden in der Form “Wo ist *LO*” gestellt, wobei *LO* das intendierte, zu lokalisierende Objekt bezeichnet. BOLA generiert eine raumbezogene Ortsbeschreibung als Antwort. Dazu werden ein Referenzobjekt und eine dazugehörige räumliche Relation ermittelt und in natürlichsprachlicher Form ausgegeben. Das System baut auf dem von Klaus-Peter Gapp entwickelten Objektlokalisationssystem OLS (Gapp, 1997) auf und erweitert dieses unter anderem um die im vorhergehenden Kapitel angeführten Anytime-Eigenschaften. BOLA stellt somit ein Realzeitsystem zur Bestimmung einer natürlichsprachlichen raumbezogenen Ortsbeschreibung dar.

Die Generierung dieser Ortsbeschreibung erfolgt, bedingt durch die Verwendung der Anytime-System-Shell und damit auch ihrer nebenläufigen Prozesse zur Bearbeitung von Teilaufgaben, inkrementell, womit die Eigenschaft der monoton wachsenden Ergebnisqualität eines Anytime-Systems erfüllt ist. Zu den wichtigsten Teilaufgaben gehören die Wahl eines besten Referenzobjekts, die Auswahl einer räumlichen Relation, die die Lagebeziehung zwischen dem zu lokalisierenden Objekt und dem potentiellen Referenzobjekt möglichst gut beschreibt, sowie eine Verbalisierungskomponente zur Ausgabe des jeweils aktuellen Ergebnisses.

In dem implementierten Prototyp wurden sowohl zweidimensionale als auch dreidimensionale Domänen modelliert. Beispiele hierfür sind eine Deutschlandkarte zur Lokalisation von Bundesländern und Städten (s. Abbildung 5.1) und eine Bürodomäne, die unter anderem einen Computer mit Maus, Tastatur und Monitor beinhaltet (s. Abbildung 4.29).

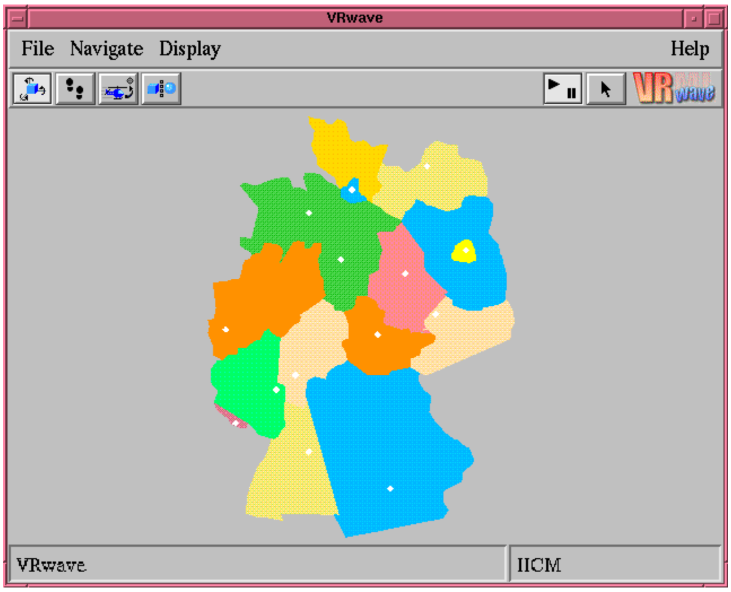

Abbildung 5.1: Beispieldomäne: Deutschlandkarte

5.2 Architektur

Es folgt die Erläuterung der Architektur des entwickelten Objektlokalisationssystems. Dabei wird explizit auf die Mechanismen und Vorgehensweisen zur Übergabe der Konfigurationsdaten des Gastsystems an die Anytime-System-Shell eingegangen.

5.2.1 Subsysteme

Zur Generierung von Lokalisationsbeschreibungen werden mehrere Subsysteme verwendet. Auf oberster Ebene der Architektur von Bola werden die zwei untergeordneten Systeme Best-RO und Multi-Relations zur Berechnung eines besten Referenzobjektes bzw. Relationenbestimmung benutzt. Best-RO besitzt seinerseits keinerlei Subsysteme, wohingegen Multi-Relations sich in Abhängigkeit von der Anzahl der möglichen Referenzobjekte wiederum in Subsysteme aufteilt, wobei pro Referenzobjekt (RO) ein Subsystem Relations-*RO* erzeugt wird. Somit ergibt sich die in Abbildung 5.2 dargestellte Systemstruktur.

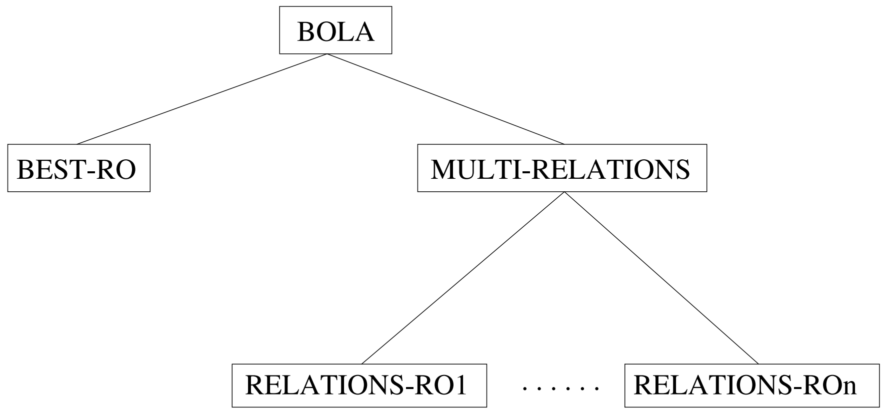

Abbildung 5.2: Architektur des Systems BOLA

Die (internen) homogenen hierarchischen Systemstrukturen wie in Abschnitt 4.3.1.2 eingeführt und die Aufgaben und Lösungsstrategien der so verwalteten Tasks sind Inhalt des nächsten Abschnitts.

5.2.2 Homogene hierarchische Systemstrukturen

5.2.2.1 BOLA

Die oberste Ebene der Systemstruktur besitzt einen sehr einfachen Aufbau. Es existieren innerhalb der zugehörigen RAMI lediglich zwei reale Tasks, die die beiden Subsysteme BEST-RO und MULTI-RELATIONS repräsentieren. Hier werden keine elementaren Lösungsstrategien verwendet, die gesamte Berechnung findet in den RAMIs der untergeordneten Systeme statt. Damit ergibt sich für die oberste RAMI des Gesamtsystems der in Abbildung 5.3 aufgezeigte lokale Ressourcenbaum, der die homogene hierarchische Systemstruktur darstellt.

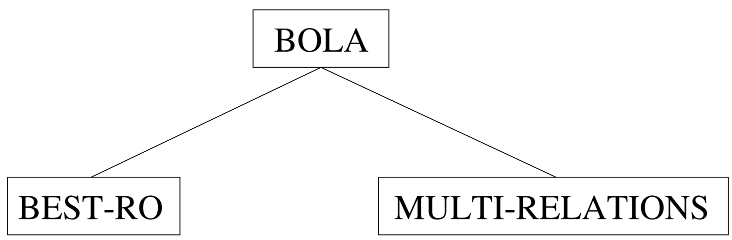

Abbildung 5.3: Homogene hierarchische Systemstruktur von BOLA

Um der Anytime-Kontrolle diesen Aufbau mitzuteilen, wird der `config`-Slot der zugehörigen Modulkontrolle bei deren Initialisierung mit der Liste `(features)` belegt. Dieser Wert gibt die zusätzlichen Slots der Modulkontrolle an, die die einzelnen Ebenen der homogenen hierarchischen Systemstruktur beschreiben. In diesem Fall existiert nur eine Ebene, abgesehen von der Wurzel, die aus den Subsystemen BEST-RO und MULTI-RELATIONS besteht. Somit ist es notwendig, daß ein Slot `features` angelegt wird, der mit der Liste `(Best-RO Multi-Relations)` besetzt wird.

5.2.2.2 Best-RO

Das Subsystem Best-RO dient der Bestimmung eines besten Referenzobjektes. Dazu werden spezielle Aspekte bezüglich der Eigenschaften des potentiellen Referenzobjektes und seiner Beziehung zu dem zu lokalisierenden Objekt untersucht. Dies sind:

- **Größe:** Im allgemeinen werden größere Objekte bei der Wahl eines bestmöglichen Referenzobjektes gegenüber kleineren präferiert. Der Grund dafür besteht in der besseren Sichtbarkeit und der damit verbundenen höheren Auffälligkeit großer Objekte.

- **Distanz:** Die Distanz spielt bei der Ermittlung der Referenzobjekte eine wichtige Rolle. Läßt man die Einflüsse anderer Aspekte außer Acht, so gilt bis zu einem gewissen Grad: Je geringer die Entfernung der beiden an der Lokalisation beteiligten Objekte, desto besser ist die Eignung des betrachteten potentiellen Referenzobjektes.

- **Vorerwähntheit:** Dieser Aspekt kommt zum Tragen, wenn mehr als eine Objektlokalisation direkt nacheinander durchgeführt werden. Es ist vorteilhaft, ein bereits bekanntes Objekt als Referenzobjekt zu wählen. Der Benutzer hat bereits dessen Lage im Raum verinnerlicht und hat somit eine bessere Ausgangsposition, eine erfolgreiche Lokalisation durchzuführen.

- **Intrinsische Front:** Üblicherweise wird eine bestimmte Front eines Objektes bevorzugt. Ein Beispiel dafür ist die Seite eines Hauses, die den Haupteingang besitzt, im Gegensatz zu einem Ball, an dem keine intrinsische Front ausgezeichnet werden kann. Existiert eine solche Seite eines Objekts, so wird es bei der Bestimmung eines Referenzobjektes bevorzugt.

- **Beweglichkeit:** Die Beweglichkeit eines potentiellen Referenzobjektes ist ein weiterer Aspekt bei der Ermittelung eines besten Referenten. Im allgemeinen werden immobile Objekte bevorzugt. Eine andere Situation entsteht, wenn sich das zu lokalisierende Objekt bewegt. Dann kann es von Vorteil sein, ein sich ebenfalls bewegendes Objekt als bestes Referenzobjekt auszuzeichnen.

- **Funktionale Abhängigkeit:** Gegenstände, die zueinander in Beziehung stehen, eignen sich bevorzugt zu einer Verwendung im Rahmen einer Lokalisation. Ein Beispiel einer funktionalen Abhängigkeit ist die Beziehung zwischen einem Monitor und der dazugehörigen Tastatur.

- **Störobjekte:** Als Störobjekte werden solche Objekte bezeichnet, die sich räumlich zwischen dem zu lokalisierenden und dem potentiellen Referenzobjekt befinden. Sie können seine Qualität als Referenz durch Verdeckung verringern oder es gänzlich unbrauchbar machen.

Eine genauere Darstellung der Bedeutung der einzelnen Aspekte im Rahmen der Ermittlung eines besten Referenzobjektes und die dazu angewendeten Vorgehensweisen werden in (Gapp, 1997) und (Blocher, 1999) gegeben. An dieser Stelle ist nur die allgemeine Struktur der Lösungsstrategien von Interesse.

Durch das Setzen des Wertes des `config`-Slots auf `(features)` und der Einführung eines Slots `features` in der Modulkontrolle der zu Best-RO gehörenden RAMI, mit dem die oben angeführten Aspekte repräsentierenden Inhalt `(Size Distance Mentioned Intrinsic-Front Movability Functional-Dependency Intervening-Objects)`, wird die in Abbildung 5.4 anhand ihres lokalen Ressourcenbaumes aufgezeigte homogene hierarchische Systemstruktur von Best-RO beschrieben.

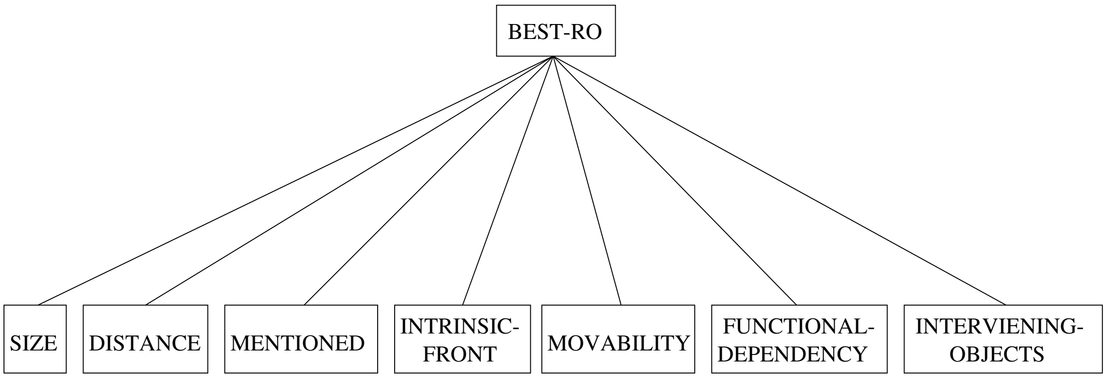

Abbildung 5.4: Homogene hierarchische Systemstruktur von Best-RO

Die Betrachtung jedes Aspektes der Bestimmung eines besten Referenzobjektes wird als selbstständige Teilaufgabe aufgefaßt und somit als eigenständiger realer Task verwaltet. Die Komplexität variiert sehr stark und trägt zur Optimierung des Anytime-Verhaltens des entstehenden Gesamtsystems bei, indem die Qualität des Ergebnisses immer weiter verbessert wird, bis schließlich das bestmögliche bestimmt ist.

Bei den Berechnungen zu den einzelnen Aspekten werden keine weiteren Subsysteme verwendet.

5.2.2.3 Multi-Relations

Multi-Relations, das zweite Subsystem der obersten RAMI, hat die Aufgabe, die jeweils besten Relationen zwischen dem zu lokalisierenden Objekt und den potentiellen Referenzobjekten zu bestimmen. Zusammen mit dem Ergebnis des im vorigen Abschnitt erläuterten Subsystems, ergibt sich so eine optimale Beschreibung der räumlichen Lage des zu lokalisierenden Objekts.

Multi-Relations besitzt für jedes potentielle Referenzobjekt ein Subsystem Relations, das der Bestimmung der bestmöglichen Relation bezüglich des zu lokalisierenden Objekts dient. Der Slot `config` der zugehörigen Modulkontrolle beinhaltet den Wert `(ROs)` und der damit bezeichnete zusätzlich eingeführte Slot `ROs` enthält die Liste der potentiellen Referenzobjekte `(RO1, RO2, ..., ROn)`. Abbildung 5.5 stellt die mittels dieser Konfigurationsdaten beschriebene homogene hierarchische Systemstruktur dar.

Die RAMI des Subsystems Multi-Relations besitzt somit ausschließlich Tasks, die wiederum RAMIs weiterer Subsysteme beinhalten.

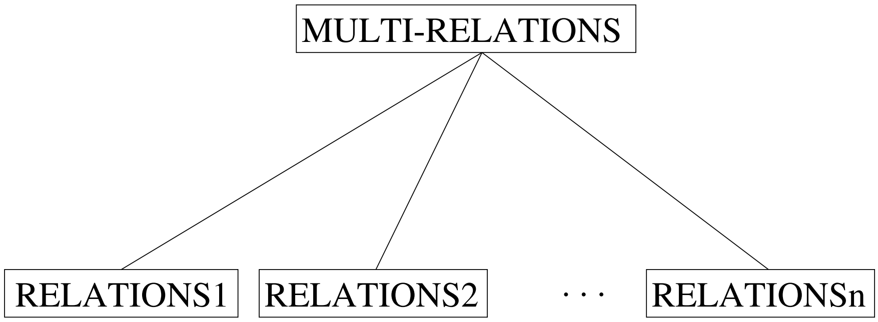

Abbildung 5.5: Homogene hierarchische Systemstruktur von MULTI-RELATIONS

5.2.2.4 RELATIONS

Die RELATIONS-Subsysteme von MULTI-RELATIONS, die der Ermittlung der Relationen zwischen dem zu lokalisierenden Objekt und den Referenzobjekten dienen, besitzen die komplexeste homogene hierarchische Systemstruktur aller verwendeten Subsysteme. Dies ist dadurch bedingt, daß es eine Vielzahl an Ausprägungen von Parameter gibt, die eine unterschiedlich genaue Bestimmung der Relationen ermöglichen. Im folgenden werden diese Parameter kurz aufgeführt:

- **Dimension:** Dieser Parameter gibt an, in welchem geometrischem Raum die Relation bestimmt wird. Zur Wahl stehen der zweidimensionale und der dreidimensionale Raum. Letzterer liefert die qualitativ besseren Ergebnisse, denn die zusätzliche Dimension ermöglicht eine exaktere Generierung der Ortsbeschreibung. Es steht eine größere Menge an räumlichen Relationen zur Verfügung. Beispielsweise kann zwischen 'hinter' und 'über' differenziert werden, was im zweidimensionalen Fall nicht möglich ist.

- **Idealisierung:** Die Idealisierung der Objekte hat einen großen Einfluß auf das Ergebnis der Relationenbestimmung. Die primitivste Form ist die Idealisierung eines Objektes als Punkt. Dazu wird der Mittelpunkt im Sinne des Schwerpunkts ermittelt und im Rahmen der weiteren Relationenberechnung verwendet. Die nächstkomplexeren Idealisierungen sind die achsenparallel umfassende Box, die minimal umfassende Box und letztlich die exakte Form des betrachteten Objektes, die dann im eigentlichen Sinn keine Idealisierung mehr darstellt.

- **Relationen-Level:** Der Relationen-Level legt die Art der untersuchten Relation fest, wie z.B. Winkelrelationen, Distanzrelationen und geographische Relationen. Zusätzlich werden auch linguistische Hecken betrachtet (s. hierzu (Kray, 1998)), die im strengen Sinne keine Relationen darstellen und damit einen Spezialfall bilden.

Die zur Bestimmung verwendeten Algorithmen werden in (Gapp, 1997) und (Blocher, 1999) ausführlich erläutert.

Der Inhalt des `config`-Slots der zugehörigen Modulkontrolle sieht folgendermaßen aus: `(RO DIMS IDEALS REL-Levels)`. Die somit benötigten zusätzlichen Slots enthalten:

- **RO:** Das betrachtete Referenzobjekt: (*RO*).
- **DIMS:** Die untersuchten Dimensionen: `(2D 3D)`.
- **IDEALS:** Die untersuchten Idealisierungen: `(Pt B-Axis B-Mini Full)`.
- **REL-Levels:** Die betrachteten Relationen-Levels: `(Containment Combined-Angle-HSM HSM-Model Hedged-Rels Angle-Rels Distance-Rels Combined-Mixed-Rels Combined-Special-Rels Combined-Angle-Rels Geographic-Rels Special-Rels)`. Ihre Bedeutungen können in (Blocher, 1999) nachgelesen werden.

Abbildung 5.6 zeigt die zu diesen Konfigurationsdaten gehörige homogene hierarchische Systemstruktur.

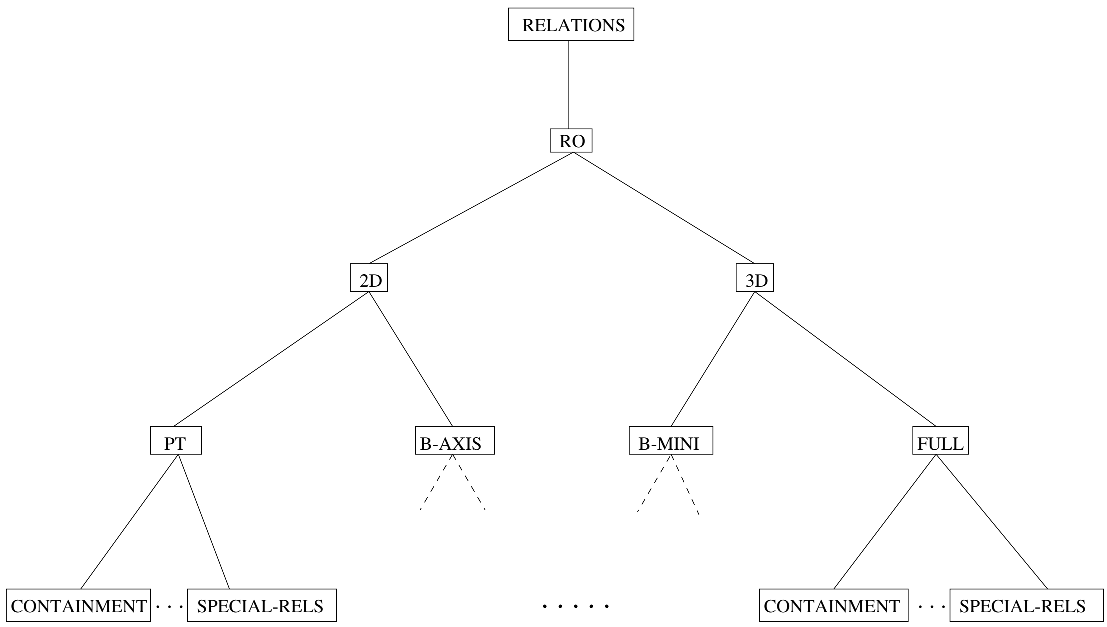

Abbildung 5.6: Homogene hierarchische Systemstruktur von RELATIONS

Damit sind die Erläuterungen zur inneren Struktur des entwickelten Systems zur Objektlokalisation abgeschlossen. Im folgenden werden noch einige Ausführungen bezüglich der Festlegung der relativen Gewichte der virtuellen Tasks der lokalen Ressourcenbäume, der Definition der benutzen Transaktionen, dem Aufruf des Gesamtsystems und dem Einbinden der Subsysteme gemacht.

5.2.3 Relative Gewichte

Wie in Abschnitt 4.5.2.3 beschrieben, dient die Funktion `get-weight` der Festlegung der relativen Gewichte aller an der Konstruktion der lokalen Ressourcenbäume beteiligten Entitäten. Im vorliegenden Beispiel wurden sämtliche relativen Gewichte so gewählt, daß alle virtuellen Tasks einer Ebene eines lokalen Ressourcenbaumes gleichstark bei der Ressourcenverteilung berücksichtigt werden. Existieren auf einer Ebene n virtuelle Tasks, so besitzt jeder von ihnen das relative Gewicht $1/n$. Aus den mit diesen Werten erzielten Erfahrungen, die in die Performanzprofile der Datenbasis einfließen, können dann in einer späteren Nutzungsphase des Objektlokalisationssystems, mittels der in Abschnitt 4.3.3.2.3 dargelegten Verfahren, die zur Ressourcenverteilung benötigten globalen Gewichte der Tasks errechnet werden.

5.2.4 Transaktionen

Transaktionen werden in diesem beispielhaft betrachteten System zur Objektlokalisation zur Kapselung von kritischen Manipulationen an den Objekten genutzt. Beispielsweise werden im Rahmen der Relationenbestimmung Drehungen der beteiligten Objekte vorgenommen. Diese müssen so in Transaktionen gekapselt werden, daß nach deren Beendigung der ursprüngliche Zustand wiederhergestellt ist. Damit wird vermieden, daß andere nebenläufig stattfindende Berechnungen ebenfalls die manipulierten Objekte benutzten und falsche Ergebnisse liefern.

5.2.5 Aufruf des Anytime-Systems

Die Funktionalität des Anytime-Systems zur Objektlokalisation steckt in der Funktion `wo?`, die im Lisp-Package `james` angesiedelt ist und durch das Senden eines Start-Signals von der Java-Werkbank aufgerufen wird und folgendermaßen aussieht:

```
(defun wo? (rami
            LO
            &rest
            args)

  (cond (LO
         (let ((mc (mc::initialize :bola LO args)))
           (RAMI rami mc)
           (mc::finalize :bola mc)
         mc))))
```

Der einzige Parameter, der durch die Verwendung der Anytime-System-Shell bedingt ist, ist `rami`, der die zugehörige RAMI beinhaltet. Diese oberste RAMI wird beim Hochfahren des Systems[1] auf generische Weise erzeugt und wird in dieser Funktion mit den aktuellen Werten belegt. Alle anderen Parameter sind anwendungsspezifisch und werden ohne Modifikation an das Gastsystem, weitergereicht.

In einem ersten Schritt wird mittels der Funktion `initialize` eine Modulkontrolle erzeugt und mit den initialen Werten belegt, die dieser RAMI zugeordnet wird. Da diese Funktion anwendungsspezifisch ist, wird sie vom Entwickler des Anytime-Systems bereitgestellt. Danach folgt der eigentliche Aufruf der Funktionalität der RAMI durch den Funktionsaufruf `RAMI`, der als Parameter die RAMI und die erzeugte Modulkontrolle erhält. Die ebenfalls anwendungsspezifische Funktion `finalize` erledigt Aufräumarbeiten, die zum Ende eines Laufs des Gastsystems notwendig sind.

[1] Auf die Vorgehensweise zum Hochfahren des Systems wird im Anhang ausführlich eingegangen.

5.2.6 Einbinden der Subsysteme

Zum Einbinden eines Subsystems wird die Lispfunktion, die es im übergeordneten Teilsystem repräsentiert, ähnlich wie die im vorigen Abschnitt beschriebene Funktion zum Aufruf des Gesamtsystems konstruiert. Der Code der entsprechenden Funktion `Best-RO?` des Lisp-Packages `bola` zur Integration des Subsystems Best-RO lautet:

```
(defun best-RO? (RO
           &rest
           args)

  (let* ((best-RO-mc (mc::initialize :bola RO args))
         (rami (james::initialize-RAMI best-RO-mc
                                        :system-name "BEST-RO"
                                        :parent james::*main-rami*
                                        :subsystem t)))

    (james::RAMI rami best-RO-mc :subsystem t)
    (mc::finalize :bola best-RO-mc)
    best-RO-mc))
```

Zu Beginn wird eine Modulkontrolle erzeugt. Der nächste Schritt unterscheidet sich zur vorher betrachteten Vorgehensweise. Es wird mittels der Funktion `initialize-RAMI` eine RAMI erzeugt, die verschiedene Parameter erhält. In diesem Zusammenhang essentiell sind `parent` und `subsystem`. Ersterer repräsentiert einen Verweis auf die übergeordnete RAMI. Der Wert des zweiten Parameters `t` gibt an, daß es sich bei dieser RAMI um eine ein Subsystem repräsentierende RAMI handelt. Die folgenden Schritte entsprechen dem schon vorgestellten Aufbau einer solchen Funktion. Die Funktionalität der RAMI wird in Gang gesetzt und nach dem Beenden des Subsystems werden anfallende Aufräumarbeiten durchgeführt.

Das Einbinden der anderen Subsysteme erfolgt analog.

Nach der Vorstellung des Aufbaus des Beispielsystems folgt eine Beschreibung seines Verhaltens in Praxis.

5.3 Beispiellauf

Als Beispiel dient die Fragestellung "Wo ist die Tastatur?" in der oben vorgestellten Bürodomäne. Abbildung 5.7 zeigt das Aussehen der Java-Werkbank zum Zeitpunkt der Aufgabenstellung.

Die Zykluslänge beträgt 6.0 Zeiteinheiten (ZE) und die Anzahl der Prozesse ist auf 40 festgelegt. Durch Drücken des Start-Buttons wird der Berechnungsvorgang mit den angegebenen Parametern angestoßen. Die Zeit spielt in diesem Beispiel eine untergeordnete Rolle, da sämtliche Funktionen der Werkbank aktiviert waren und die Performanz des Systems stark beeinflußten.

Abbildung 5.8 stellt die Oberfläche bei einer Unterbrechung dar. Das bis zum Zeitpunkt 7.6 Sekunden bestimmte Zwischenergebnis lautet: "Monitor" mit einer

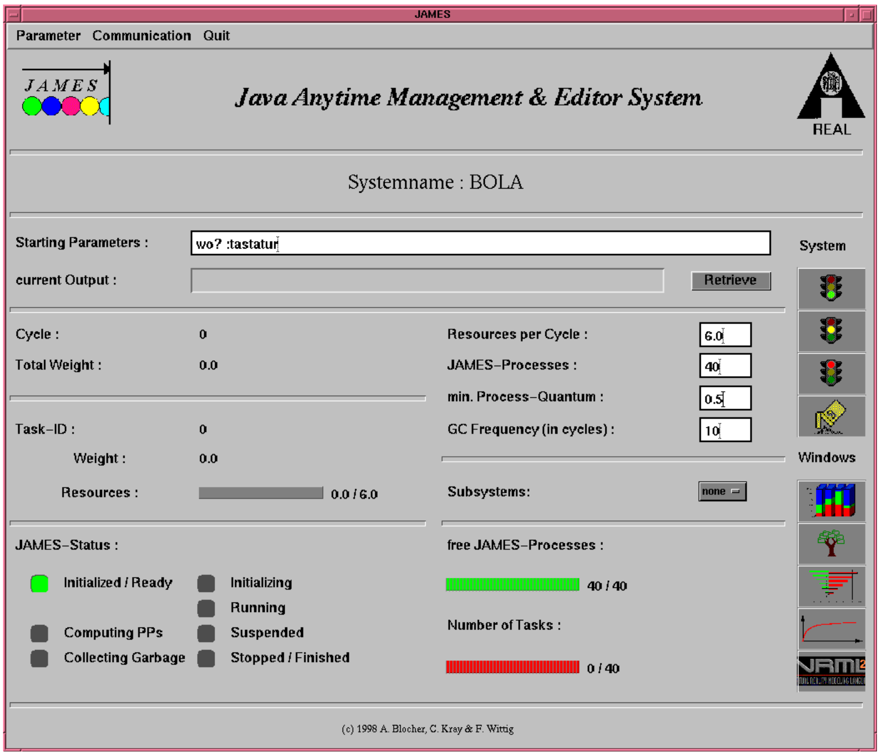

Abbildung 5.7: Beispiellauf - Anfragestellung

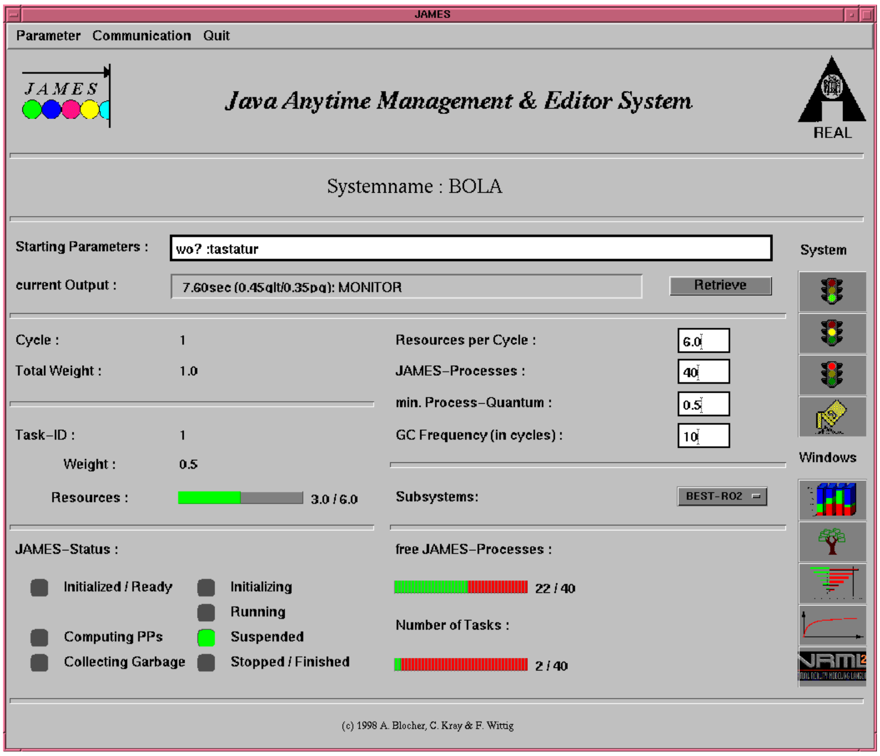

Abbildung 5.8: Beispiellauf - Unterbrechung

Qualität von 0.45. Es wurde also bislang als bestes potentielles Referenzobjekt der Monitor ermittelt. Eine Relation existiert noch nicht. Ein Ausschnitt des lokalen Ressourcenbaums der RAMI des Subsystems BEST-RO wird in Abbildung 5.9 aufgezeigt. An der unterschiedlichen Anzahl der Nachfolgeknoten verschiedener innerer Knoten wird deutlich , daß schon einige Tasks vollständig abgearbeitet wurden.

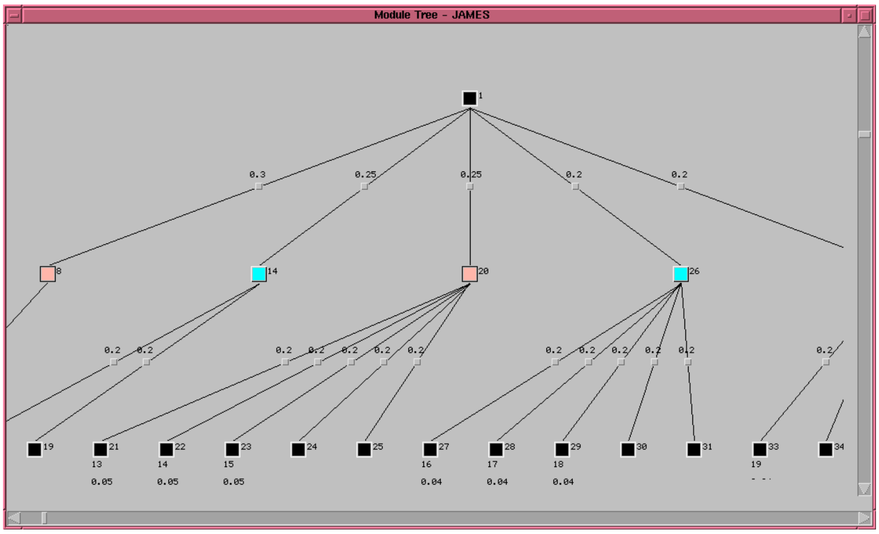

Abbildung 5.9: Beispiellauf - Lokaler Ressourcenbaum von BEST-RO

Nach der Fortsetzung der Berechnung ergibt sich als Endergebnis die Ortsbeschreibung “Die Tastatur befindet sich links vor dem Monitor.” mit einer Qualität von 1.0. Es wurden innerhalb der obersten RAMI 13 Zyklen zur kompletten Durchführung der Berechnung benötigt (s. Abbildung 5.10).

Abbildung 5.11 enthält das zugehörige zweidimensionale Balkendiagramm der Ressourcenverteilung der RAMI des Subsystems MULTI-RELATIONS. Es existieren fünf potentielle Referenzobjekte für die jeweils die beste Relation bezüglich der Tastatur ermittelt werden muß. Die Gleichverteilung der Ressourcen wird an den ungefähr gleichen Flächeninhalten der farbigen Bereiche deutlich. Abweichungen ergeben sich durch die vorhandenen Transaktionen. Es treten auch Zyklen auf, in denen nicht alle Tasks bearbeitet werden. Dies ist dadurch bedingt, daß einer oder mehrere Tasks Transaktionen durchführen, die deutlich mehr Ressourcen benötigen als für ihn bzw. sie vorberechnet wurden. Gegen Ende der Berechnung sind alle bis auf einen Task abgearbeitet. Er kann dann die komplette Zykluszeit ausnutzten.

Das Aussehen des Performanzprofils des Gesamtsystems kann anhand Abbildung 5.12 ersehen werden. Jede Stufe des Graphen repräsentiert die Beendigung einer Teilaufgabe der Gesamtberechnung.

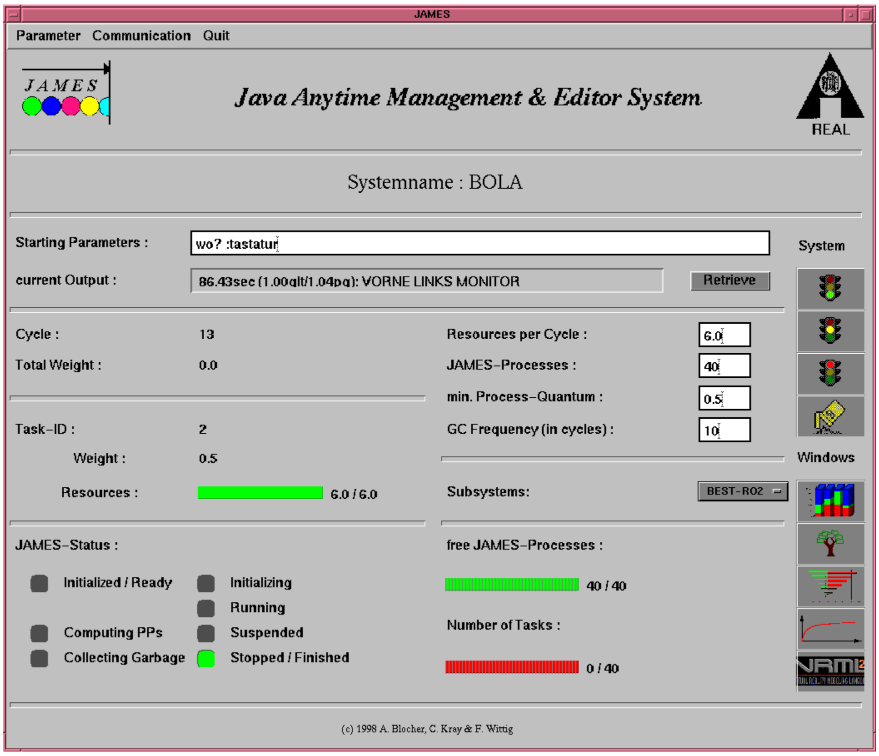

Abbildung 5.10: Beispiellauf - Endergebnis

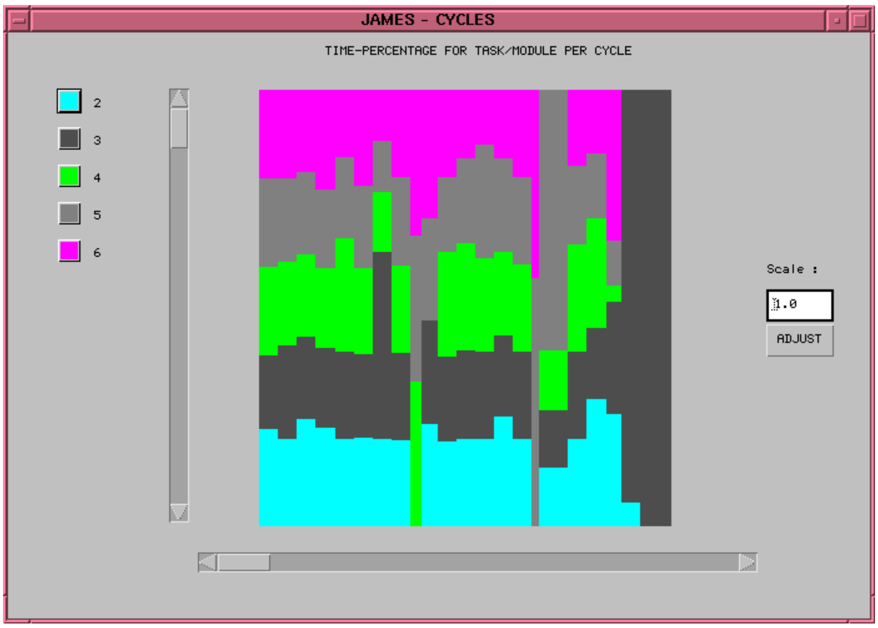

Abbildung 5.11: Beispiellauf - Ressourcenverteilung von MULTI-RELATIONS

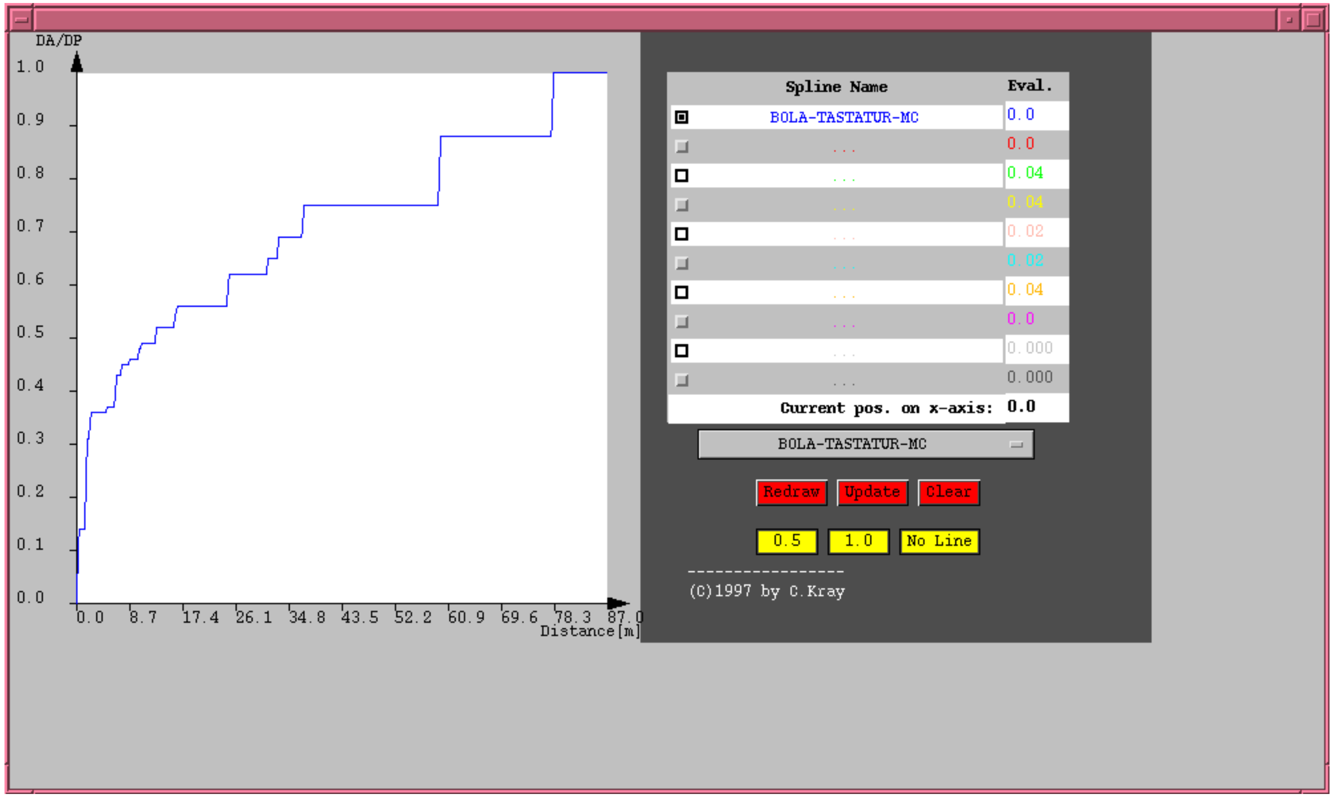

Abbildung 5.12: Beispiellauf - Performanzprofil des Gesamtsystems

Prozeßanzahl	Zykluslänge (in ZE)	Laufzeit (in Sekunden)
5	3	ca. 26
5	6	ca. 21
5	9	ca. 21
7	3	ca. 16
7	6	ca. 16
7	9	ca. 16
12	3	ca. 14
12	6	ca. 13
12	9	ca. 13
20	3	ca. 14
20	6	ca. 13
20	9	ca. 13

Tabelle 5.1: Verhalten der Laufzeit des Gesamtsystems bei unterschiedlichen Parameterwerten

5.4 Einflüsse der Parameter

Durch Variation der Parameterwerte ändert sich das Verhalten des Gesamtsystems. Die folgenden Zeiten wurden im Gegensatz zu Abschnitt 5.3 bei minimaler Funktionalität der Java-Werkbank bestimmt, um mögliche unerwünschte Einflüsse auf das Performanzverhalten des Gesamtsystems zu vermeiden.

In Tabelle 5.1 sind die Laufzeiten des obigen Beispiels bei unterschiedlichen Parameterwerten aufgeführt. Als Werte für die Prozeßanzahl wurden 5, 7, 12 und 20 gewählt. Dazu wurden jeweils die Zeiten bei den Zykluslängen 3, 6 und 9 ZE der obersten RAMI ermittelt. Die Parameter der untergeordenten RAMIs wurden aus Gründen der zu hohen Komplexität der Problematik festgehalten. Die so bestimmten Werte reichen von 13 bis 26 Sekunden. Es handelt sich hierbei um ungefähre Werte, da die Umgebung nicht absolut stabil gehalten werden konnte (CPU-Auslastung, Netzwerklast). Betrachtet man beispielsweise die Laufzeiten für die unterschiedliche Zykluslängen bei fünf Prozessen, dann wird deutlich, daß es ab einer festen Grenze keinen Sinn mehr macht, die Zykluslänge zu erhöhen, da keine Beschleunigung der Laufzeit mehr eintritt. Dies ist dadurch bedingt, daß die fünf lauffähigen Tasks ihre in der Lage sind, ihre Berechnungen komplett durchzuführen, und somit nicht die ganze zur Verfügung stehende Zykluszeit nutzen müssen. In diesem Fall findet im wesentlichen eine sequentielle Abarbeitung aller Tasks des Gesamtsystems statt. Ebenso gilt, daß eine Erhöhung der Prozeßanzahl ab einem gewissen Wert keine kürzeren Laufzeiten mehr bringt. Hier wirken sich die im Gastsystem Bola recht zahlreich vorhandenen Transkationen aus. Ihr Vorhandensein führt dazu, daß nur eine feste Anzahl von Tasks pro Zyklus arbeiten können, die dessen gesamte Dauer benötigen und somit die Berechnungen der übrigen lauffähigen Tasks verhindern.

Die Veränderung der relativen Gewichte hat keine großen Auswirkungen auf die Gesamtlaufzeit des Systems. Es müssen in ihrer Gesamtheit die gleichen Berechnungen ausgeführt werden. Durch die dadurch veränderte Ressourcenverteilung kann sich allerdings ein anderes Aussehen des Performanzprofils ergeben. Je nach Wahl der relativen Gewichte ergeben sich unterschiedliche Reihenfolgen der Berechungen sowie verschiedene Ressourcenzuteilungen an die einzelnen Tasks. Somit kann durch entsprechende Festlegung der relativen Gewichte ein angestrebtes Performanzverhalten des Gesamtsystems erzielt werden. Beispielsweise kann ein starkes Ansteigen der Qualität zu Beginn unter Generierung einer geringen Anzahl von Zwischenergebnissen bis zu diesem Zeitpunkt oder ein langsameres Ansteigen mit vielen sich stetig verbessernden Zwischenergebnissen erreicht werden.

Eine möglichst optimale Wahl der Parameter ist im allgemeinen sehr stark von der Struktur des Gastsystems abhängig, insbesondere die Anzahl und die Länge der vorhandenen Transaktionen spielt in diesem Zusammenhang eine wichtige Rolle.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Aufbauend auf dem Begriff des Anytime-Algorithmus und deren Compilierung nach Zilberstein wurde ein System zum Ressourcenmanagement in Anytime-Systemen vorgestellt. Diese Anytime-System-Shell ermöglicht die Konstruktion beliebiger beschränkt-optimaler Realzeitsysteme, die die in Tabelle 2.1 aufgeführten wünschenswerten Eigenschaften besitzten. Durch das Einbringen von Erfahrungen früherer Durchläufe der konstruierten Systeme bei der Bestimmung der Ressourcenzuteilung an die einzelnen Teilaufgaben der aktuellen Problemstellung wird ein ressourcenadaptierendes Verhalten induziert. Die zugehörige Benutzerschnittstelle wurde als Werkbank konzipiert und ermöglicht die Manipulation verschiedener Parameter, insbesondere besteht die Möglichkeit, die Ressourcenverteilung während eines Systemlaufs zu beeinflußen. Durch die Verwendung der Programmiersprache Java im Rahmen der Implementation können mit Hilfe der Anytime-System-Shell entwickelte Realzeitsysteme über des World Wide Web (an-)gesteuert und kontrolliert werden. Am Beispiel der Entwicklung eines Prototyps eines beschränkt-optimalen Lokalisationsagenten wurde die praktische Anwendung des beschrieben Systems gezeigt.

6.2 Ausblick

Die Anytime-System-Shell kennt als von ihr verwaltbare Ressourcentypen außer der Zeit bereits zusätzlich die Anzahl der der Berechnung zur Verfügung stehenden Prozesse. Die Größe des verfügbaren Speichers oder die Anzahl der vorhandenen Prozessoren wären weitere Ressourcentypen, die in dieses Konzepts integriert werden könnten.

Eine weitere Möglichkeit der Erweiterung ist das Einbringen von adaptierenden Fähigkeiten der Anytime-System-Shell bezüglich verschiedener ihrer Parameter, wie der Zykluszeit, der Anzahl der verwendeten Prozesse, des minimalen Prozeßquantums und der Garbage-Collection-Frequenz.

Eine Performanzsteigerung ließe sich durch eine Parallelisierung des Schedulings der lauffähigen Tasks mittels einer Unterstützung der Verwendung mehrerer Prozessoren erreichen. Dies müßte wahrscheinlich durch Zurückgreifen auf Methoden aus dem Gebiet der Betriebssysteme realisiert werden.

Ein kritischer Aspekt der Anytime-System-Shell ist die Zeitmessung mit Hilfe der Lisp-Funktion `get-internal-real-time`. Ihre Genauigkeit ist abhängig von der Lisp-Implementation und genügt nicht den Anforderungen an ein Realzeitsystem.

Bisher wurde die Anytime-System-Shell nur zur Konstruktion des beschränkt-optimalen Objektlokalisationsagenten Bola genutzt. Interessant wäre es, mit ihr andere Anytime-Systeme zu entwickeln und damit eine Validierung des Shell-Charakters vorzunehmen. Hierzu würde sich beispielsweise ein Fahrernavigationssystem anbieten, denn dort könnte dann unter Umständen das Objektlokalisationssystem als Subsystem integriert werden.

Anhang A

Handbuch

Im folgenden werden das Vorgehen und die dazu benötigten Funktionen zum praktischen Einsatz der Anytime-System-Shell erläutert. Begonnen wird mit allgemeinen Bemerkungen zum Aufbau des Systems und den benötigten Schritten zum Hochfahren, gefolgt von der Konstruktion von Anytime-Systemen unter Verwendung der Anytime-System-Shell. Abschließend werden die wichtigsten Funktionen mit ihren Parametern beschrieben.

A.1 Allgemeines

Tabelle A.1 gibt einen Überblick über die Inhalte der Verzeichnisse der JAMES-Directory-Hierarchie.

Diese Hierarchie muß in dieser Form aufrecht erhalten werden, da in der Anytime-System-Shell und der Java-Werkbank relative Pfade zum Zugriff auf die verschiedenen Informationen benutzt werden.

A.2 Hochfahren

JAMES benötigt die Systeme JALCOSTE und CHANNELS. Deshalb muß sichergestellt sein, daß vor jeder Nutzung von JAMES bzw. eines mit seiner Hilfe konstruierten Systems diese beiden Hilfssysteme eingeladen werden.

Es existieren zwei Arten, ein mit Hilfe der Anytime-System-Shell entwickeltes System zu starten. Einerseits die in Abschnitt 4.5.4.1 beschriebene Ansteuerung über eine vom Entwickler unter Verwendung des JALCOSTE-Systems bereitgestellte WWW-Seite und andererseits das konventionelle Hochfahren des Systems auf einem oder mehreren lokalen Rechner. Die erste Möglichkeit ist die für den Benutzer komfortablere, denn es muß lediglich ein in der entsprechenden WWW-Seite integrierter Button betätigt werden. Wie hierzu vorzugehen ist, kann im Handbuch des JALCOSTE-Systems nachgelesen werden (Kern, 1998). Deshalb wird anschließend lediglich die zweite Vorgehensweise erläutert.

Das konventionelle Hochfahren eines mit Hilfe der Anytime-System-Shell entwickelten Systems erfolgt in vier Schritten:

1. Einladen der benötigten (compilierten) Lisp-Dateien

Verzeichnis	Inhalt
src	Lisp-Quellcode-Dateien der Anytime-System-Shell
bin	Lisp-Binär-Dateien der Anytime-System-Shell
demo	Diverse Hilfsdateien zum Hochfahren des Systems, temporäre Dateien
java	Übergeordnetes Verzeichnis aller die Java-Werkbank betreffender Dateien
java/html	Den Systemen zugeordnete HTML-Dateien mit den JALCOSTE-Applets
java/DE/uni_sb/jalcoste	JALCOSTE-Klassenverzeichnis
java/DE/uni_sb/james	Klassendateien der Java-Werkbank
vrml	(anwendungsspezifisch) VRML-Dateien der BOLA-Domänen

Tabelle A.1: Inhalte der Verzeichnisse der JAMES-Directory-Hierarchie

2. Initialisierung der Lisp-Seite des JALCOSTE-Systems zur Durchführung der Kommunikation des Anytime-Systems mit der Java-Werkbank

3. Starten des Appletviewers mit der Java-Seite des JALCOSTE-Systems

4. Initialisierung des Anytime-Systems

Im ersten Schritt müssen sämtliche Dateien der beteiligten Systeme eingeladen werden, insbesondere die der Anytime-System-Shell, des Gastsystems und des JALCOSTE-Systems.

Zur Initialisierung der Lisp-Seite des für die Kommunikation verantwortlichen JALCOSTE-Systems wird der folgende Befehl verwendet:

```
(jalcoste::initialize-jalcoste :host host :port port)
```

Der Parameter *host* gibt den Namen des Rechners als String an, auf dem der im nächsten Schritt gestartete Appletviewer laufen soll. *Port* repräsentiert die Nummer des zur Kommunikation genutzten TCP/IP-Ports.

Beim implementierten Prototyp des beschränkt-optimalen Objektlokalisationsagenten aus Kapitel 5 sieht dieser Befehl folgendermaßen aus:

```
(jalcoste::initialize-jalcoste :host "tintin" :port 5668)
```

Damit wird eine Kommunikationsverbindung zum Rechner "tintin" unter Verwendung des TCP/IP-Ports 5668 aufgebaut.

Der dritte Schritt besteht im Starten des Appletviewers in einer Unix-Shell auf dem im vorigen Schritt festgelegten Rechner:

```
appletviewer Datei.html &
```

Dabei beinhaltet die Datei `Datei.html` das vom JALCOSTE-System benötigte Java-Applet. Als Beispiel dient die Datei `JamesComplex.html` des implementierten Prototyps BOLA:

```
<HTML>
<HEAD>
<TITLE>Testumgebung fuer Java-Applets</TITLE>
<!-- Changed by: Frank Wittig, 21-Oct-1997 -->
</HEAD>
<BODY>
<APPLET codebase="file:/WW/lucifer/REAL/new-systems/james/java"
        code="DE.uni_sb.jalcoste.core.DSAppletTest.class"
        width="300"
        height="20">
<PARAM name="SYSTEM" value="JAMES">
<PARAM name="STATUS" value="STANDARD">
<PARAM name="PORT" value="5668">
<PARAM name="BUTTON" value="Los gehts">
<PARAM name="COMPLEX" value="DE.uni_sb.james.JamesComplexGenerator">
</APPLET>
</BODY>
</HTML>
```

Danach kann dann die Initialisierungsphase des Anytime-Systems mittels der Funktion `initialize-rami` angestoßen werden. Dies sieht im betrachteten Beispiel so aus:

```
(initialize-RAMI nil
                 :system-name "BOLA"
                 :number-of-processes 40
                 :cycles-before-gc 10
                 :minimum-process-quantum 0.5
                 :resources-per-cycle 6.0
                 :use-pp-database nil
                 :display "abacus"
                 :gui t
                 :verbose 0)
```

Auf dem Rechner "abacus" erscheint die Oberfläche der Java-Werkbank mit deren Hilfe nach Abschluß der Initialisierungsphase Anfragen an das auf diesem Rechner arbeitende Anytime-System gestellt werden können.

Dieser Vorgang des konventionellen Hochfahrens kann in einem Script zusammengefaßt werden und somit vereinfacht werden[1].

A.3 Konstruktion von Anytime-Systemen

In diesem Abschnitt werden die wichtigsten Aspekte der praktischen Konstruktion von Anytime-Systemen mittels der Anytime-System-Shell beschrieben. Dies sind sie

[1] Am Lehrstuhl für Künstliche Intelligenz der Universität des Saarlandes wurde hierfür ein Account eingerichtet, der eine Umgebung für das komfortable Durchführen von Demonstrationen der Anytime-System-Shell und der mit ihrer Hilfe entwickelten Gesamtsysteme bereitstellt.

Übergabe der Konfigurationsdaten, das Einbinden von Subsystemen, die Integration von anwendungsspezifischen Funktionen der Java-Werkbank und die Unterstützung weiterer Lisp-Funktionen. Einige dieser Aspekte sind ausführlich in vorhergehenden Abschnitten dieser Arbeit dargestellt und deshalb hier nur kurz angesprochen.

A.3.1 Konfigurationsdaten

Die Übergabe der Konfigurationsdaten des Gastsystems an die Anytime-Kontrolle geschieht mittels der Modulkontrolle. Dazu werden wie in Abschnitt 4.5.2.2 erläutert der `config`-Slot und eventuell zusätzliche vom Entwickler einzuführende Slots genutzt.

Die Übergabe der relativen Gewichte wird wie in Abschnitt 4.5.2.3 beschrieben mittels der Funktion `get-weight` durchgeführt. Diese Funktion muß vom Entwickler des Gastsystems im Lisp-Package `mc` bereitgestellt werden. Sie wird von der Anytime-Systems-Shell zum Aufbau des lokalen Ressourcenbaums genutzt.

---

get-weight *Funktion*

Syntax: get-weight *objekt*

Aufgabe: Bestimmung der relativen Gewichte.

Parameter:
objekt: Das Objekt, dessen relatives Gewicht ermittelt werden soll.

Rückgabewert: Das relative Gewicht des Objekts *objekt*.

Lisp-Package: mc

Die im konstruierten Gesamtsystem verwendeten Transaktionen werden durch das Eintragen ihrer Funktionsnamen in den Slot `transactions` der obersten Modulkontrolle innerhalb der Systemstruktur festgelegt. Voraussetzung hierfür ist das Kapseln des eine Transaktion ausmachenden Lisp-Codes in einer Lisp-Funktion.

A.3.2 Einbinden von Subsystemen

Subsysteme werden in das Gesamtsystem integriert, indem sie in eine Lisp-Funktion gekapselt werden, die in den übergeordneten Systemen aufgerufen werden. Ein Prototyp einer solchen Funktion hat das folgende Aussehen:

```
(defun subsystem-prototyp (args)

  (let* (; Modulkontrolle erzeugen
         (mc (mc::initialize args))
```

```
          ; RAMI erzeugen
          (rami (james::initialize-RAMI
                  mc
                  :system-name "Prototyp"      ; Name
                  :parent parent               ; Vorgaenger
                  :cycles-before-gc 10         ; GC-Frequenz
                  :minimum-process-quantum 0.2 ; min. Prozessquantum
                  :resources-per-cycle 2.0     ; Zykluslaenge
                  :subsystem t)))              ; Subsystem

     ; Starten der RAMI
     (james::RAMI rami mc :subsystem t)

     ; Aufraeumen
     (mc::finalize args)))
```

Der erste Schritt besteht in der Konstruktion einer Modulkontrolle mittels der Funktion `initialize`. Das Vorhandensein einer Modulkontrolle stellt eine Voraussetzung dar, um anschließend durch den Aufruf `initialize-RAMI` die das Subsystem repräsentierende RAMI zu erzeugen. Deren Funktionalität wird durch `RAMI` angestoßen. Den Abschluß bildet die Funktion `finalize`, die anfallende Aufräumarbeiten erledigt.

Solche Funktionen müssen im Lisp-Package des zugehörigen Subsystems definiert werden.

Ebenso wie `get-weight` sind `initialize` und `finalize` anwendungsspezifisch und müssen vom Entwickler des Gastsystems implementiert werden. Dabei muß innerhalb der Funktion `initialize` die homogene hierarchische Systemstruktur durch Setzen des `config`-Slots und dem Anlegen der benötigten zusätzlichen Slots der Modulkontrolle übergeben werden. Die beiden Funktionen stellen defaultmäßig Transaktionen dar.

initialize *Funktion*

Syntax: initialize *args*

Aufgabe: Konstruktion und Initialisierung einer Modulkontrolle.

Parameter:
args: Werte, die in die Slots der erzeugten Modulkontrolle eingetragen werden.

Rückgabewert: Die erzeugte und initialisierte Modulkontrolle.

Lisp-Package: mc

finalize *Funktion*

Syntax: finalize *args*

Aufgabe: Erledigen von Aufräumarbeiten nach Beendigung der Berechnungen einer RAMI.

Parameter:
args: Anwendungsspezifisch.

Lisp-Package: mc

A.3.3 Integration anwendungsspezifischer Funktionen der Java-Werkbank

Wie die Java-Werkbank um zusätzliche anwendungsspezifische Funktionalitäten erweitert werden kann wird ausführlich in Abschnitt 4.5.4.3 an dem Beispiel der Integration eines Programms zur Anzeige von VRML-Welten erläutert.

A.3.4 Unterstützung weiterer Lisp-Implementationen

Um die Verwendung der Anytime-System-Shell unter weiteren Lisp-Implementationen (außer den bereits unterstützten Liquid Common Lisp und Allegro Common Lisp) zu ermöglichen, ist es notwendig, verschiedene Modifikationen an den Inhalten der beiden Dateien `process-handler-james.lisp` und `advice-handler.lisp` vorzunehmen. Dort werden die generischen Funktionsaufrufe in implementationsspezifische umgesetzt (s. Abschnitte 4.5.2.6 und 4.5.2.7). In die einzelnen Funktionsdefinitionen können weitere Lisp-Implementationen integriert werden. Als einfaches Beispiel dient die generische Funktion zum Deaktivieren eines Lisp-Prozesses:

```
(defun deactivate-process (p)
  #+lucid   (lcl::deactivate-process p)
  #+allegro (mp:process-disable p))
```

Hier wird als generischer Name `deactivate-process` benutzt. Dieser wird dann im Liquid Common Lisp in `lcl::deactivate-process` und im Allegro Common Lisp in `mp:process-disable` umgesetzt.

A.3.5 Elemente der Java-Werkbank und ihre Funktionen

In Abbildung A.1 werden die Elemente des Hauptfensters der Java-Werkbank dargestellt und ihre Funktionen zugeordnet.

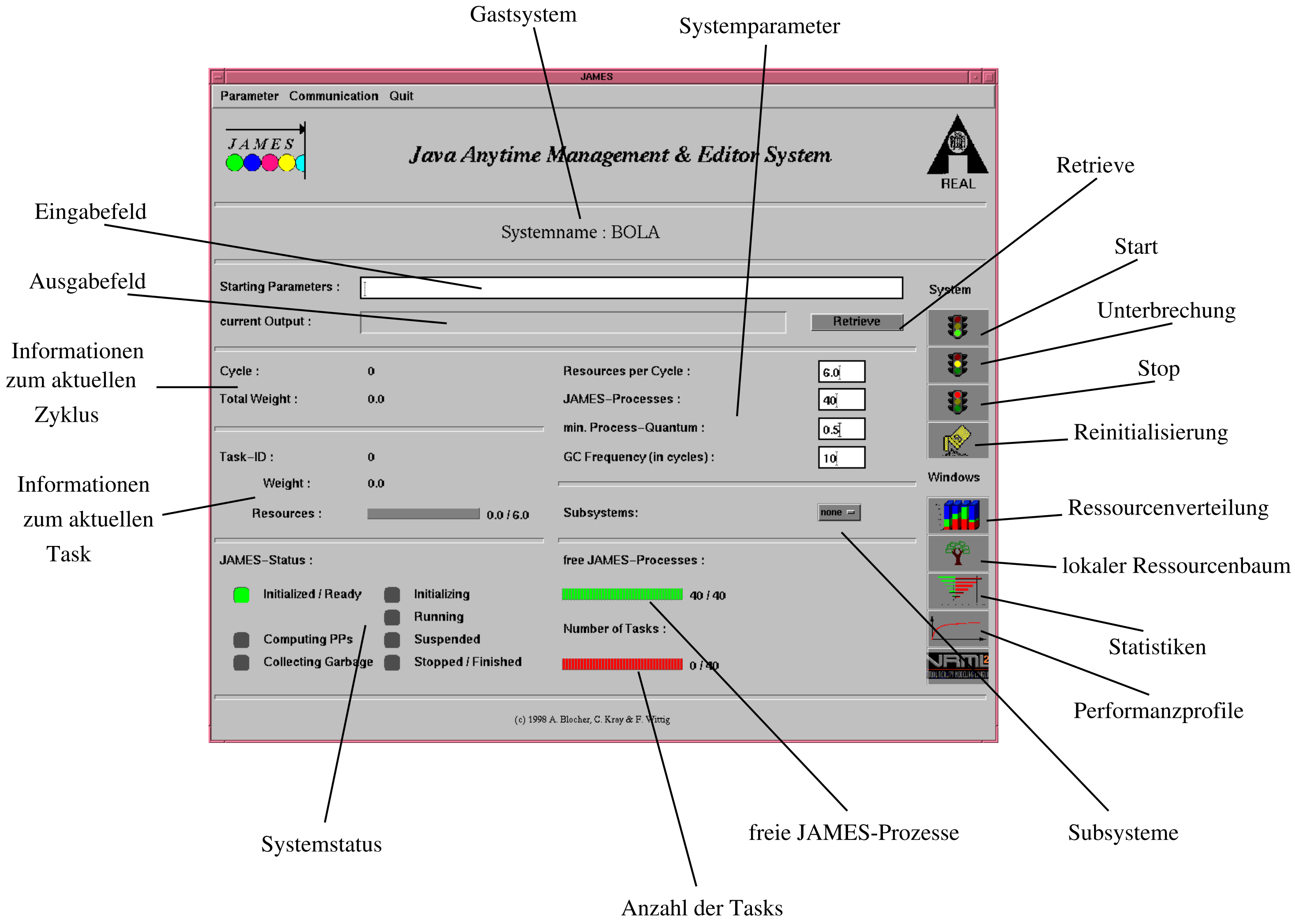

Abbildung A.1: Elemente der Java-Werkbank

A.4 Wichtige Funktionen und globale Variablen

Abschließend werden die wichtigsten Lisp-Funktionen der Anytime-System-Shell mit ihren Aufgaben und den möglichen Parametern ihres Aufrufs sowie die globalen Variablen in alphabetischer Reihenfolge aufgelistet:

---

adjust-PP-database *Funktion*

Syntax: adjust-PP-database *rami*

Aufgabe: Diese Funktion kombiniert die im aktuellen Durchlauf des Systems erzeugten Performanzprofile mit den entsprechenden der Datenbasis und bringt somit die Informationen der Performanz des aktuellen Durchlaufs in die aufgezeichneten Daten ein.

Parameter:
rami: die in der Architektur des Systems am höchsten angesiedelte RAMI.

Datei: pp-manager.lisp

---

all-RAMIs *globale Variable*

Inhalt: Diese globale Variable enthält eine Liste aller erzeugten RAMIs.

---

build-resources-tree *Methode*

Syntax: build-resources-tree *rami*

Aufgabe: Hiermit wird der lokale Ressourcenbaum der als Parameter übergebenen RAMI aufgebaut.

Parameter:
rami: die RAMI, deren lokaler Ressourcenbaum konstruiert werden soll.

Datei: module-handler.lisp

compute-distribution-of-resources *Funktion*

Syntax: compute-distribution-of-resources *rami*

Aufgabe: Anhand der aktuellen Ergebnisqualität, der benötigten Zeit und den Performanzprofilen der Datenbasis wird die Ressourcenverteilung für den nächsten Zyklus berechnet.

Parameter:
rami: die RAMI, für deren lauffähige Tasks die Ressourcenverteilung bestimmt werden soll.

Datei: pp-manager.lisp

distribute-processes *Funktion*

Syntax: distribute-processes *rami*

Aufgabe: Die der RAMI zur Verfügung stehenden Prozesse werden anhand der relativen Gewichte der Tasks des lokalen Ressourcenbaums verteilt.

Parameter:
rami: die RAMI, in der die Verteilung der Prozesse vorgenommen werden soll.

Datei: module-handler.lisp

expand-resources-tree *Methode*

Syntax: expand-resources-tree *tree rami*

Aufgabe: Diese Funktion erweitert den aktuellen lokalen Ressourcenbaum solange bis die zur Verfügung stehenden Prozesse aufgebraucht sind. Sie wird nur im Rahmen der alternativen Vorgehensweise der Anytime-Kontrolle zur Verwaltung des lokalen Ressourcenbaums genutzt.

Parameter:
tree: der lokalen Ressourcenbaum, der erweitert werden soll.
rami: die RAMI, deren lokaler Ressourcenbaum erweitert werden soll.

Datei: module-handler.lisp

gui-RAMI *globale Variable*

Inhalt: Diese globale Variable enthält die (oberste) RAMI, die für die Kommunikation mit der Java-Werkbank verantwortlich ist. Sie ist nur belegt, wenn die Anytime-System-Shell zusammen mit der Java-Werkbank läuft.

initialize-RAMI *Methode*

Syntax: **initialize-RAMI** *mc* :system-name *name* :parent *parent* :number-of-processes *number* :resources-per-cycle *rpc* :minimum-process-quantum *mpq* :cycles-before-gc *cbgc* :subsystem *subsystem* :use-pp-database *uppd* :display *display* :gui *gui* :verbose *verbose*

Aufgabe: Es wird eine RAMI mit den Eigenschaften der angegebenen Parametern erzeugt.

Parameter:
mc: die zugehörige Modulkontrolle
name: (key) Name des zugehörigen Subsystems, Default: "Generic"
parent: (key) die übergeordnete RAMI, Default: `nil`
number: (key) die Anzahl der freien Prozesse (nur in oberster RAMI benutzt), Default: `15`
rpc: (key) die Zykluslänge, Default: `8.0`
mpq: (key) das minimale Prozeßquantum, Default: `0.2`
cbgc: (key) die Garbage-Collection-Frequenz, Default: `10`
subsystem: (key) Flag, das angibt, ob es sich um ein Subsystem handelt, Default: `nil`
uppd: (key) Flag, das angibt, ob die Datenbasis der Performanzprofile genutzt werden soll, Default: `nil`
display: (key) das X-Display, das genutzt wird, Default: das X-Display des Rechners, auf dem das System läuft
gui: (key) Flag zur Angabe, ob das System mit der Java-Werkbank läuft, Default: `nil`
verbose: (key) der Verbose-Level der Ausgabe von Systemmeldungen, Default: `1`

Rückgabewert: die erzeugte RAMI.

Datei: anytime-process-handler.lisp

interaction *Funktion*

Syntax: interaction *rami*

Aufgabe: Diese Funktion wickelt die Kommunikation zwischen der Lisp-Seite und der Java-Werkbank ab.

Parameter:
rami: die in der Systemarchitektur am höchsten angesiedelte RAMI.

Datei: james-gui.lisp

JAMES-cycle *Methode*

Syntax: JAMES-cycle *rami*

Aufgabe: Hier wird ein Zyklus der angegebene RAMI durchgeführt.

Parameter:
rami: die RAMI, deren Zyklus durchgeführt wird.

Datei: anytime-process-scheduler.lisp

JAMES-Scheduler *Methode*

Syntax: JAMES-Scheduler *rami*

Aufgabe: In dieser Funktion wird das Scheduling der lauffähigen Tasks der angegebenen RAMI durchgeführt.

Parameter:
rami: die RAMI, deren Scheduling durchgeführt wird.

Datei: anytime-process-scheduler.lisp

main-RAMI *globale Variable*

Inhalt: Diese globale Variable repräsentiert die in der Systemarchitektur am höchsten angesiedelte RAMI.

PP-database-path *globale Variable*

Inhalt: Die Variable enthält den Pfad der Datenbasis der akquirierten Performanzprofile.

print-resources-tree *Methode*

Syntax: **print-resources-tree** *rami*

Aufgabe: Der lokale Ressourcenbaum wird ausgegeben. Die Funktion ist für Debugging-Zwecke vorgesehen.

Parameter:
rami: die RAMI, deren lokaler Ressourcenbaum angezeigt werden soll.

Datei: module-handler.lisp

RAMI *Methode*

Syntax: **RAMI** *rami mc* :subsystem *subsystem* :alternativ-distribution-algorithm *ada* :resources-per-cycle *rpc* :cycles-before-gc *cbgc* :minimum-process-quantum *mpq* :level-of-rd *lrd*

Aufgabe: Diese Funktion beinhaltet die Funktionalität einer RAMI.

Parameter:
rami: die betrachtete RAMI
mc: die zugehörige Modulkontrolle
subsystem: (key) Flag, das angibt, ob es sich um eine RAMI eines Subsystems

handelt, Default: `nil`
ada: (key) Flag zur Angabe der benutzten Vorgehensweise zur Verwaltung des lokalen Ressourcenbaumes, Default: `nil`
rpc: (key) die Zykluslänge, Default: `nil`
cbgc: (key) die Garbage-Collection-Frequenz, Default: `nil`
mpq: (key) das minmale Prozeßquantum, Default: `nil`
lrd: (key) Tiefe der Ebene des lokalen Ressourcenbaums auf der die Ressourcenverteilung vorgenommen wird, Default: maximale Tiefe
Die entsprechenden Parameter von **initialize-RAMI** können durch eine Angabe beim Aufruf dieser Funktion überschrieben werden.

Datei: anytime-process-handler.lisp

reinitialize-JAMES *Methode*

Syntax: reinitialize-JAMES *rami*

Aufgabe: Hiermit wird eine Reinitialisierung des Systems vorgenommen, d.h. alle Daten des vorhergehenden Durchlaufs sind nach dem Ausführen dieser Funktion verloren.

Parameter:
rami: die oberste RAMI.

Datei: anytime-process-handler.lisp

run-task-with-resources *Methode*

Syntax: run-task-with-resources *task rami*

Aufgabe: Diese Funktion läßt den angegebenen Task der RAMI entsprechend den ihm zugeteilten Ressourcen unter Berücksichtigung eventuell vorhandener Transaktionen arbeiten.

Parameter:
task: der Task, der Prozessorzeit zugewiesen bekommt (alle lauffähigen Tasks werden in einer Schleife abgearbeitet).
rami: die aktuelle RAMI.

Datei: anytime-process-handler.lisp

task-to-RAMI-hashtable *globale Variable*

Inhalt: Diese Variable beinhaltet eine Hashtable, die einen Task auf die zugehörige RAMI abbildet.

Alle diese Funktionen und globalen Variablen sind im Lisp-Package `james` angesiedelt.

Literatur

ACL (1996). *Allegro Common Lisp User Guide.* Franz Inc.

Beckert, A. & Baus, A. (1998). *Implementation von Methoden zur Compilierung von Anytime Algorithmen.* (Fortgeschrittenenpraktikum am Lehrstuhl Prof. Dr. Wahlster, Universität des Saarlandes, Saarbrücken)

Bell, G., Pesce, M. & Parisi, A. (1995). *The Virtual Reality Modeling Language - Version 1.0 Specification.* Silicon Graphics, Inc. und InterVista Software, Inc.

Blocher, A. (1999). *Ressourcenadaptierende Raumbeschreibung: Ein beschränkt-optimaler Lokalisationsagent.* Dissertationsprojekt, Fachbereich 14 - Informatik, Universität des Saarlandes, Saarbrücken.

Blocher, A. & Stopp, E. (1998). Time-Dependent Generation of Minimal Sets of Spatial Descriptions. In P. Olivier & K.-P. Gapp (Hrsg.), *Representation and Processing of Spatial Expressions* (S. 57-72). Lawrence Erlbaum Associates.

Boddy, M. (1991a). Anytime Problem Solving using Dynamic Programming. In *Proceedings of the Ninth National Conference on Artificial Intelligence* (S. 738-743). Anaheim, California.

Boddy, M. (1991b). *Solving Time-Dependent Problems: A Decision-Theoretic Approach to Planning in Dynamic Environments.* Doktorarbeit, Departement of Computer Science at Brown University.

Brandherm, B., Wagner, D. & Wittig, F. (1997). *Lösungsalgorithmen in mehrfach verbundenen Bayes'schen Netzen.* URL: http://w5.cs.uni-sb.de/~dwagner/publications.html. (Fortgeschrittenenpraktikum am Lehrstuhl Prof. Dr. Wahlster im Fachbereich Informatik der Universität des Saarlandes, Saarbrücken)

Bronstein, I. N. & Semendjajew, K. A. (1991). *Taschenbuch der Mathematik.* Stuttgart: B.G. Teubner.

Cormen, T. H., Leiserson, C. E. & Rivest, R. L. (1990). *Introduction to Algorithms.* Cambridge, Massachusetts: MIT Press.

Dean, T. & Boddy, M. (1988). An Analysis of Time-Dependent Planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence* (S. 49-54). Minneapolis, Minnesota.

Gapp, K.-P. (1997). *Objektlokalisation: Ein System zur sprachlichen Raumbeschreibung.* Wiesbaden: Deutscher Universitätsverlag.

Garvey, A., Humphrey, M. & Lesser, V. (1993). Task Interdependencies in Design-to-time Real-time Scheduling. In *Procceedings of the Eleventh National Conference on Artificial Intelligence* (S. 580-585). Washington D.C.

Garvey, A. & Lesser, V. (1993). *A Survey of Research in Deliberative Real-Time Artificial Intelligence* (UMass Computer Science Technical Report Nr. 93-84). Department of Computer Science, University of Massachusetts.

Garvey, A. & Lesser, V. (1994). Design-to-time Real-Time Scheduling. *IEEE Transactions on Systems, Man and Cybernetics*, *23*(6), 1491-1502.

Garvey, A. & Lesser, V. (1996). Design-to-time Scheduling and Anytime-Algorithms. *SIGART Bulletin*, *7*(3).

Good, I. J. (1971). Twenty-seven principles of rationality. In V. P. Godambe & D. A. Sprott (Hrsg.), *Foundation of Statistical Inference.* Toronto: Holt, Rinehart & Winston.

Görz, G. & Kesseler, M. (1994a). An Anytime Protocol for Speech Parsing. In H. Niemann, R. de Mori & G. Hanrieder (Hrsg.), *Progress and Prospects of Speech Research and Technology* (S. 257-261).

Görz, G. & Kesseler, M. (1994b). Anytime Algorithms for Speech Parsing? In *Proceedings COLING-94* (S. 997-1001). Kyoto.

Gosling, J., Joy, B. & Steele, G. (1996). *The JAVA Language Specification.* Addison-Wesley.

Grass, J. (1996). Reasoning about computational resource allocation - An introduction to anytime algorithms. *ACM Crossroads.*

Grass, J. & Zilberstein, S. (1995). *Anytime Algorithm Development Tools* (CMO-SCI Nr. 95-94). University of Massachusetts at Amherst: Computer Science Department.

Horvitz, E. J. (1987). Reasoning about Beliefs and Actions under Computational Resource Constraints. In *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence, Seattle WA* (S. 429-444). Mountain View, CA.

Jameson, A. & Buchholz, K. (1998). Einleitung zum Themenheft "Ressourcenadaptive kognitive Prozesse". *Kognitionswissenschaft*, *7*(3), 95-100.

Jameson, A., Kipper, B., Ndiaye, A., Schäfer, R., Simons, J., Weis, T. & Zimmerman, D. (1994). Cooperative to be noncooperative: The dialog system PRACMA. In B. Nebel & L. Dreschler-Fischer (Hrsg.), *Proceedings of the Eighteenth German Conference on Artificial Intelligence, Saarbrücken, Germany, September 1994* (S. 106-117). Berlin, Heidelberg: Springer.

Kern, U. (1998). *Java Lisp Connection System.* Universität des Saarlandes, Saarbrücken. (URL: http://w5.cs.uni-sb.de/jalcoste)

Königsberger, K. (1990). *Analysis 1.* Berlin, Heidelberg: Springer.

Kray, C. (1998). *Ressourcenadaptierende Verfahren zur Präzision von Lokalisationsausdrücken und zur Generierung von linguistischen Hecken* (Memo Nr. 67). Universität des Saarlandes, Saarbrücken: Sonderforschungsbereich 378, Fachbereich 14 Informatik.

Lewis, H. R. & Papdimitriou, C. H. (1981). *Elements of the Theory of Computation.* Prentice-Hall International, Inc.

LIQUID (1997). *Liquid Common Lisp documentation.* URL: http://www.harlequin.com/education/books/lcl_doc.html.

Ndiaye, A. (1998). *Rollenübernahme als Benutzermodellierungsmethode: Globale Antizipation in einem transmutierbaren Dialogsystem.* Doktorarbeit, Fachbereich 14 - Informatik, Universität des Saarlandes, Saarbrücken.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Interference.* San Mateo, California: Morgan Kaufmann.

REAL (1996). *REAL - Ressourcenapdaptive Lokalisation.* URL: http://w5.cs.uni-sb.de/real.html.

Russell, S. & Norvig, P. (1995). *Artificial Intelligence - A Modern Approach.* Prentice Hall, Inc.

Russell, S. J. & Subramanian, D. (1995). Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, *1*, 1-36.

Russell, S. J. & Wefald, E. (1991). *Do the right thing: studies in limited rationality.* Cambridge, MA: MIT Press.

SFB (1997). *Sonderforschungsbereich 378 "Ressourcenadaptive kognitive Prozesse".* URL http://www.coli.uni-sb.de/sfb378/.

Silberschatz, A. & Galvin, P. B. (1994). *Operating System Concepts* (4. Aufl.). Addison Wesley.

Steele, G. L. (1990). *Common Lisp: The Language* (2. Aufl.). Digital Press.

Tanenbaum, A. S. (1992). *Modern Operating Systems.* Prentice-Hall International, Inc.

VERBMOBIL (1991). *VERBMOBIL - Translation of Spontaneous Speech.* URL: http://www.dfki.de/verbmobil/.

VRWAVE (1997). *VRwave.* URL: http://www2.iicm.edu/vrwave/.

Wahlster, W. (1993). Verbmobil-Translation of Face-to-Face Dialogs. In *Proceedings of MT Summit IV* (S. 127-135). Kobe, Japan.

Wahlster, W., Blocher, A., Baus, J., Stopp, E. & Speiser, H. (1998). Ressourcenadaptierende Objektlokalisation: Sprachliche Raumbeschreibung unter Zeitdruck. *Kognitionswissenschaft*, *7*(3), 111-117.

Wahlster, W. & Tack, W. (1997). SFB 378: Ressourcenadaptive Kognitive Prozesse. In M. Jarke, K. Pasedach & K. Pohl (Hrsg.), *Informatik'97 - Informatik als Innovationsmotor, 27. Jahresta gung der Gesellschaft für Informatik, Aachen, 24.-26. September 1997* (S. 51-57). Berlin, Heidelberg: Springer.

Wang, P. (1996). Problem-Solving under Insufficient Resources. In *Working Notes of the AAAI Fall Symposium on Flexible Computation.* Cambridge, Massachusetts.

Zilberstein, S. (1993). *Operational Rationality through Compilation of Anytime Algorithms.* Doktorarbeit, University of California at Berkeley.

Zilberstein, S. (1996). Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, *Fall 1996*, 73-83.

Zilberstein, S. & Russell, S. (1996). Optimal composition of real-time systems. *Artificial Intelligence*, *1-2*(82), 181-213.

Index