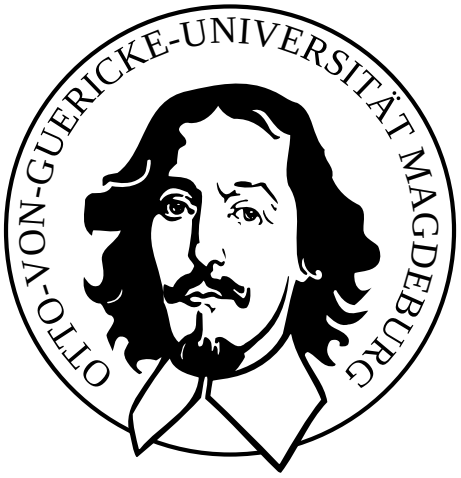
Maßgeschneidertes Planungswesen in Betriebssystemfamilien

Diplomarbeit

Michael Schulze



„Otto-von-Guericke“ Universität Magdeburg
Fakultät für Informatik
Institut für Verteilte Systeme

Aufgabenstellung: Prof. Dr. Wolfgang Schröder-Preikschat

Betreuung: Dipl. Inf. Holger Papajewski

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Verbunden mit dem Wunsch des Menschen nach immer mehr Komfort und Sicherheit geht eine stetig steigende Technisierung der alltäglichen Umwelt einher. Oftmals ist nicht einmal zu erkennen, daß in einem Gerät, Werkzeug oder sonstigen Gebrauchsgegenstand ein Mikroprozessor oder Controller seine Arbeit verrichtet. Der Rechenchip ist in die Umgebung eingebettet, weshalb solche Systeme als eingebettete Systeme bezeichnet werden. Auf diesen Architekturen laufen so genannte eingebettete Betriebssysteme, welche die meist recht knappen Speicher und Rechenressourcen verwalten. Daher dürfen sie selbst nicht verschwenderisch damit umgehen. Aus diesem Grund kann kein Mehrzweckbetriebssystem Verwendung finden, wie es z.B. Linux [SP99b] und Windows™ [Cus93] sind.

Der Anwendungsfall fordert somit ein speziell angepaßtes Betriebssystem, welches durch den Entwickler zu schaffen ist. Dies stellt einen sehr kostenintensiven Aspekt eines Produktes dar, weshalb Betriebssysteme in Baukastenform eine Möglichkeit bieten, eine schnelle Adaption von einem zum folgenden Produkt zu erreichen [SP99a]. Es wird in solchen Fällen auch von Betriebssystemfamilien gesprochen. Ein Beispiel für eine derartige Betriebssystemfamilie stellt PURE[1] [BGP+99] dar. Eine Familie besteht aus den unterschiedlichsten Mitgliedern und eine Anwendung sucht die für sich passende Ausprägung aus. Somit bestimmt die Problemstellung das Aussehen des eingebetteten Betriebssystems.

Weiterhin soll die Anwendung nur für die Funktionalität zahlen, welche sie auch tatsächlich nutzt. Daher ist es grundlegend erforderlich ein System maßschneidern zu können. Das Maßschneidern erfolgt während der Konfiguration, bei der die Bestandteile des Systems ausgewählt werden. Ziel ist es nur die tatsächlich benötigte und genutzte Funktionalität ins entgültige System aufzunehmen. Durch den Familiengedanken vorgegeben, sind Bausteine eines Teilbereiches meist in mehreren Ausprägungen vorhanden. Die Bausteine einer Region haben die glei-

[1] Portable Universal Runtime Environment

che Schnittstelle, aber differierende Eigenschaften und somit ein unterschiedliches Systemverhalten. Das zusammengesetzte System aus den Bausteinen wird als Familienmitglied bezeichnet.

Ein Teilbereich im Betriebssystem ist das Planungswesen (engl. scheduling). In seine Zuständigkeit fällt die Verwaltung des Betriebsmittels CPU und die Vergabe von Rechenzeit an die Prozesse. Durch Planungskriterien wird Einfluß auf die Zuteilung der Ressource Prozessor genommen. Ein Umschalten von Prozeß zu Prozeß wird immer durch das Planungswesen initiiert. Unterschiedliche Problemstellungen verlangen unterschiedliche Planungsstrategien, weshalb der Anwendungsfall vorgibt, welche Strategie zu wählen ist.

Ein Parameter für die Wahl ist das Wissen über das Verhalten der Prozesse, da jene sporadisch, periodisch oder aperiodisch arbeiten können. Weiterhin unterliegt ein Prozeß möglicherweise speziellen Zeitanforderungen, die naturgemäß auch einzuhalten sind. Das Schlagwort heißt Echtzeitfähigkeit, wobei hier auch zwischen weichem und hartem Zeitverhalten unterschieden werden kann. Eine weitere Kenngröße beeinflußt die Wahl der Strategie. Sind Prozesse neu erzeugbar oder ist die Anzahl im System konstant? Diese und weitere Punkte sind ausschlaggebend, ob eine statische, dynamische oder echtzeitfähige Planung einzusetzen ist.

1.2 Familienkonzept vs. Objektorientierung

Eine Programmfamilie [Par75] besteht aus einer Menge von Programmen, die viele Gemeinsamkeiten besitzen. Daher ist es von Vorteil diese zusammen zu entwickeln und zu implementieren, damit kein mehrfacher Aufwand betrieben werden muß.

Im Artikel [HFC76] wird dieser Ansatz erneut aufgegriffen. Das Programmfamilienkonzept geht dabei von einer minimalen Basis von Funktionalitäten aus, welche durch minimale Systemerweiterungen vergrößert wird. Die Erweiterungen sollten dabei so klein wie möglich ausfallen, da sie jeweils neue Eigenschaften darstellen. Dadurch entsteht ein System aus vielen dünnen Schichten, wobei jede Ebene eine neue minimale Basis darstellt. Die Funktionalität und Komplexität des Systems steigt mit jeder neu hinzugefügten Schicht an. Eine schrittweise Verfeinerung der Funktionalität der Familie wird angestrebt [Wir71]. Eine Ebene verwendet nur Funktionen der darunterliegenden Ebenen, wodurch das System nach jeder Ebene vollständig validiert und eingesetzt werden kann. Entwurfsentscheidungen sollten aus diesem Grund soweit wie möglich auf einen späteren Zeitpunkt verschoben werden, um Anwendungen nicht mehr als nötig einzuschränken [SP94].

Eine spezielle Umsetzung des Konzeptes durch eine eigens dafür entwickelte Sprache ist nicht nötig. Die objektorientierte Implementierungsart eignet sich hervorragend, um den Gedanken der Familie umzusetzen. Eine Basisklasse steht

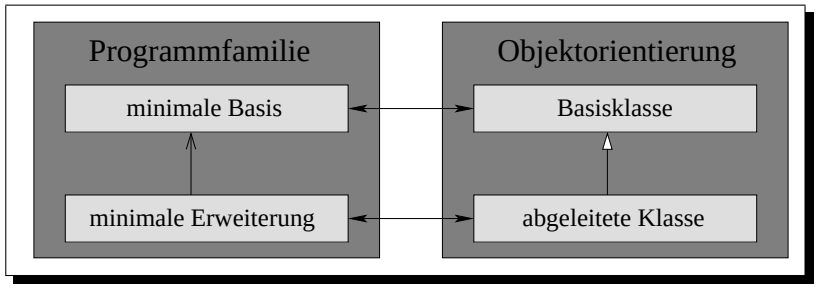


Abbildung 1.1: Familienkonzept vs. Objektorientierung

hier für die minimale Basis und die funktionalen Erweiterungen werden durch abgeleitete Klassen dargestellt. Dabei entsteht ein abstrakter Datentyp, welcher wiederum als minimale Basis betrachtet werden kann. Diese Basis besitzt jedoch einen gesteigerten Funktionsumfang. Die Abbildung 1.1 macht den Sachverhalt noch einmal graphisch deutlich. Es können auch für ein und die selbe Klasse verschiedene Implementierungen vorliegen und die Anwendung kann durch Konfiguration die für sie geeignetste Variante aussuchen.

Das Hinauszögern von Entwurfsentscheidungen bis zum spät möglichsten Zeitpunkt kann durch die Verwendung des Mechanismus des späten Bindens umgesetzt werden. Prinzipiell erlaubt es der Anwendung eigene Implementierungen in das System einzugliedern. Für diese Möglichkeit sind Kosten zu tragen, denn hierbei ist zur Übersetzungszeit des Programmtextes nicht ersichtlich, welche Funktion aufgerufen werden muß. Daher sind Datenstrukturen bereitzustellen, die den Aufruf gestatten. Die Informationen belegen naturgemäß Speicher, wodurch ein gewisser Teil der Kosten entsteht. Ein weiterer zu zahlender Teil entsteht beim Aufruf der Funktion selbst, weil noch nicht feststeht, welche Variante dieser genutzt werden soll. Durch Auswertung der zusätzlichen Informationen, läßt sich dies jedoch herausfinden. So ist durch die objektorientierte Verwirklichung der Familiengedanke sehr gut umzusetzen.

1.3 Pure

Wie bereits erwähnt, handelt es sich bei PURE nicht nur um ein Betriebssystem, sondern um eine Betriebssystemfamilie. PURE setzt das Konzept der Familie vom Entwurf bis zur Implementierung konsequent um. Daher besteht an vielen unterschiedlichen Stellen die Möglichkeit einer Konfiguration, wodurch eine große Anzahl von verschiedenen Familienmitgliedern erzeugt werden kann. So ist es beispielsweise möglich, Systeme zu generieren, die im tiefsten eingebetteten Bereich ihre Verwendung finden, in dem extreme Ressourcenknappheit die Regel ist. Daher ist es von strategischer Bedeutung, daß Systeme nur den Funktionsumfang besitzen, den die Anwendung auch tatsächlich benötigt.

PURE wurde objektorientiert entworfen und mit Hilfe der Sprache C++ [Str97, May92] implementiert. Durch eine feingranulare Klassenhierarchie wurde eine extrem starke Konfigurierbarkeit erreicht. Momentan kann PURE auf den Architekturen Intel x86, Motorola PowerPC und HC12, Siemens C167 sowie Atmel ARM nativ ausgeführt werden. Die Weiterentwicklung kann oberhalb von beispielsweise Linux als Gastebenenversion vorangebracht werden. Bei einer Portierung von PURE auf andere Prozessortypen sind nur wenige Teile des Systems anzupassen, da die Hardwareeigenschaften in speziellen maschinenabhängigen Klassen gekapselt sind.

Bei der Konfiguration des Systems für eine bestimmte Anwendung werden die Klassen herausgesucht, welche die tatsächlich benötigte Funktionalität bereit stellen. Ist die gebrauchte Funktionalität noch nicht Bestandteil der Klassenbibliothek, so wird diese erstellt und der Programmfamilie hinzugefügt.

Zur Zeit unterstützt die bestehende Klassenbibliothek das kontrollierte Ausführen von Programmfäden, die Synchronisation und Kommunikation zwischen diesen, sowie die Behandlung und Synchronisation von Unterbrechungen.

Die Arbeit beschäftigt sich nun mit dem Gebiet des Planungswesens, dessen Aufgabe es ist, die Ausführung von Fäden zu koordinieren. Es entsteht somit eine neue Familie von Schedulern, welche einfach zu erweitern und zu konfigurieren ist.

1.4 Echtzeitsysteme

Unter Echtzeitfähigkeit wird Rechtzeitigkeit verstanden. Das bedeutet, daß das Ergebnis einer Berechnung oder die Reaktion auf ein Ereignis zur rechten Zeit erfolgen muß. Echtzeitfähigkeit wird in vielen Lebensbereichen unserer täglichen Umwelt vorausgesetzt. Beispielsweise übernehmen Echtzeitsysteme das Steuern und Regeln von technischen Prozessen, oder sie überwachen medizinische Apparaturen und vieles andere mehr. Die größte Verbreitung von Echtzeitsystemen wird im sicherheitskritischen Umfeld angetroffen. Durch ein Versagen des Systems kann es zu fatalen Schäden an Material und Menschen kommen. Deshalb müssen Systeme diese Art mit besonderer Sorgfalt entwickelt werden.

Eine Definition für den Echtzeitbetrieb lautet nach /DIN 44300/ folgendermaßen:

> Echtzeitbetrieb ist der Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind derart, daß die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zufälligen zeitlichen Verteilung oder zu vorbestimmten Zeitpunkten auftreten.

Echtzeitsysteme sind, wie aus der Definition zu entnehmen ist, durch Zeitbedingungen gekennzeichnet, jedoch muß nicht jeder Bestandteil eines solchen Systems Zeitschranken unterworfen sein. Ist es für die Anwendung ausreichend, daß die Zeitbedingungen für den größten Teil der Fälle erfüllt werden, oder sich geringfügige Überschreitungen der Zeitschranken ergeben dürfen, dann wird von weichen Zeitbedingungen gesprochen. Harte Zeitbedingungen zeichnen sich hingegen dadurch aus, daß unter allen Umständen keinerlei Überschreitungen von Zeitschranken auftreten dürfen. Bei einer harten Zeitschranke muß ergo das Ergebnis bis zu der anvisierten Zeit vorliegen, da ein Verfehlen der Zeitschranke möglicherweise schwerwiegende Auswirkungen haben kann, weil das Ergebnis bereits wertlos ist. Daher muß es in solchen System Maßnahmen zur Fehlertoleranz geben, die greifen, wenn beispielsweise in einer chemischen Großanlage eine Störung auftritt. Eine mögliche Reaktion auf den Fehlerfall ist das Abschalten der Anlage.

Kapitel 2

Grundlagen

2.1 Prozeßmodell

Der Prozeß, oft auch als Task oder Faden[1] bezeichnet, ist die Ausführung eines sequentiellen Programms auf einem Prozessor in seiner spezifischen Umgebung. Diese wird auch als Kontext bezeichnet. Hierbei ist er der Träger der Aktivität, wobei er sich als Abstraktion vom Prozessor darstellt. Der Task erfüllt eine vom Programm spezifizierte Aufgabe.

Ein Kontext ist immer nur genau einem Prozeß zugeordnet, niemals einem Programm, denn Programme können auch aus mehreren Prozessen bestehen. Für die Auführung eines Prozesses benötigt dieser Ressourcen, welche beispielsweise Prozessorzeit, Speicher und Peripherie sind.

Die Ausführungszeit eines Prozesses ist maßgebliche von verschiedenen Faktoren abhängig:

- Leistungsfähigkeit des Prozessors
- Verfügbarkeit von Betriebsmitteln
- Peripherie des Prozessors
- die in die Ausführung eingehenden Daten
- Verzögerungen durch Erbringung wichtigerer Aufgaben

Ein Prozeß ist die kleinste planbare Einheit. Das Planen von verschiedenen unabhängigen Prozessen gestattet es, mehrere unterschiedliche Aufgaben bzw. deren Lösung auf einem Prozessor zu erbringen. Dabei wird jede Aufgabe durch einen oder mehrere Prozesse repräsentiert.

In der schematischen Darstellung 2.1 ist der Vorgang entsprechend aufgezeigt. Die Abbildung zeigt die Anwendung, die einen oder mehrere Prozesse zur

[1] Im folgenden werden die Begriff Faden, Thread und Task auch als Synonyme für Prozeß verwendet.

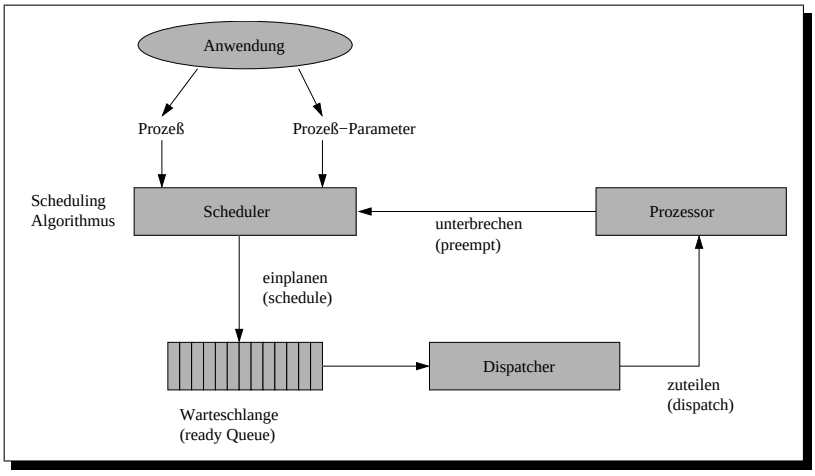


Abbildung 2.1: Schematische Darstellung des Scheduling und Dispatching

Erbringung der Aufgabe an den Scheduler[2] übergibt. Dieser führt eine Planung anhand von Prozeßparametern (Abschnitt 2.1.3) durch. Als Resultat entsteht eine Warteschlange, die durch den Dispatcher[3] abgearbeitet wird. Die Zuteilung des Prozessors erfolgt an den als Ersten in der Warteschlange stehenden.

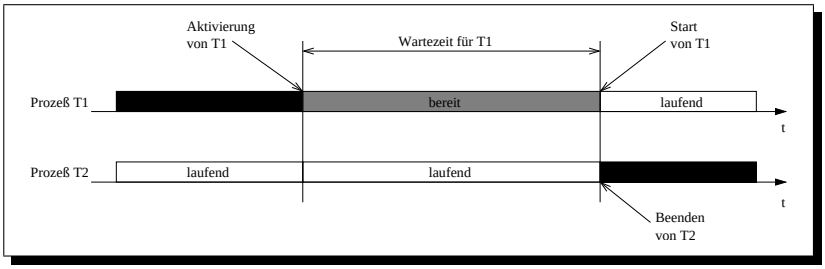
Abhängig von den Anforderungen der Anwendung ist zwischen verschiedenen Ausprägungen von Prozessen auszugehen,

- bei denen die Ausführung zwischen Start und Abschluß nicht unterbrochen werden kann (engl. non-preemptive),
- bei denen die Ausführung nach jeder Anweisung unterbrochen werden kann (engl. preemptive), außer es sind besondere Einschränkungen vorhanden.

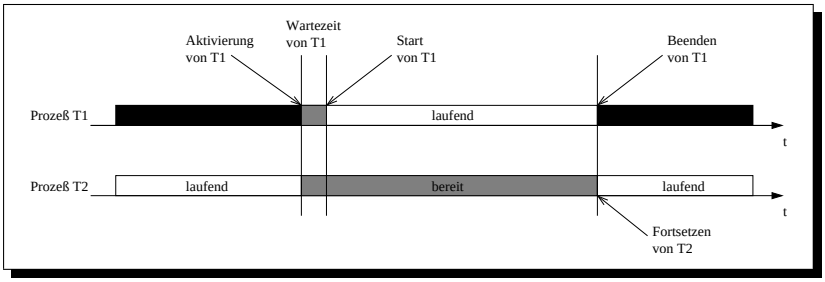
Demzufolge ist es nicht immer durchführbar, den momentan laufenden Prozeß vom Prozessor zu verdrängen. Das Verdrängen eines Prozesses wird als Context Switch[4] bezeichnet, welcher durch den Dispatcher durchgeführt wird. Dabei werden die prozeßspezifischen Verwaltungsinformationen, zu denen auch der Prozessorregistersatz zählt, in den zu verdrängenden Kontext gesichert und die vom zu startenden Prozeß geladen bzw. initialisiert.

Durch das präemptive bzw. nicht präemptive Verhalten der Prozesse entstehen naturgemäß unterschiedliche Pläne für die Abarbeitung auf dem Prozessor. Sind Prozesse präemptiv, ist beispielsweise ein Verfahren mit Zeitscheiben möglich, bei dem die Prozesse auf der CPU gemultiplext werden. Weiterhin können (wichtigere) Prozesse bevorzugt behandelt werden, indem sie (unwichtigere) verdrängen. All dies ist bei nicht unterbrechbaren Tasks nicht möglich. Sie geben implizit die

[2] Scheduler(engl.) = Steuerprogramm
[3] Dispatcher(engl.) = Rechenzeitverteiler
[4] Context Switch(engl.) = Prozeßumschaltung



(a) nicht unterbrechbare Prozesse



(b) unterbrechbare Prozesse

Abbildung 2.2: unterschiedliche Abarbeitung abhängig von der Prozeßvariante

Kontrolle über den Prozessor auf, in dem sie sich beendigen oder blockieren. Um die Anwendung optimal zu unterstützen, können entweder präemptive oder nicht präemptive Prozesse oder auch beide Varianten im System vorhanden sein.

In den Abbildungen 2.2(a) und 2.2(b) wird das unterschiedliche Verhalten der Prozesse für die beiden Betriebsmodi dargestellt. Dabei kann im Bild 2.2(a) erkannt werden, daß der Prozeß 2 sich nicht unterbrechen läßt, obwohl Prozeß 1 der wichtigere von beiden ist. Hingegen ist in der anderen Darstellung 2.2(b) zu sehen, wie Prozeß 2, beim aktivieren von Prozeß 1, aufgeschoben wird. Dadurch kommt Prozeß 1 umgehend zum Rechnen.

Weiterhin ist anzumerken, daß die Prozesse beim Wechseln ihre Betriebszustände ändern. Welche Zustände dabei ein Prozeß annehmen kann, wird im folgenden Abschnitt ausführlich behandelt.

2.1.1 Prozeßzustände

Wie bereits erwähnt, können Prozesse verschiede Zustände im Verlauf ihres Bestehens annehmen. Das Zustandsdiagramm in der Abbildung 2.3 zeigt die einzelnen Stadien, die eine Task teilweise auch mehrfach durchlaufen kann. Die Zustände werden im Bild als ausgefüllte Kreise dargestellt, wobei die Pfeile die Aktionen

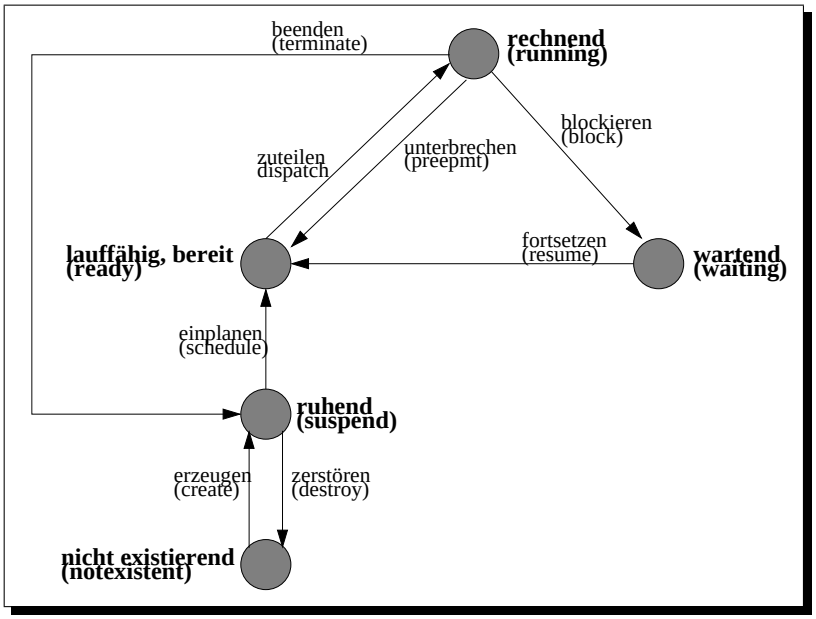


Abbildung 2.3: Zustandsdiagramm eines Prozesses

kennzeichnen, die zu ihnen führen. Zusätzlich ist zu jedem graphischen Element eine Annotation im Deutschen und in Klammern im Englischen vorgenommen, damit deutlich wird, um welchen Zustand oder welche Aktionen es sich handelt. Die Zustände sind durch Fettschreibung etwas hervorgehoben.

Der Übergang von einem Zustand zum nächsten kann durch den Prozeß selber oder aber durch eine externe Quelle ausgelöst werden. Beispielsweise ist der Übergang vom **rechnenden** zum **wartenden** Zustand eine freiwillige Aufgabe des Prozessors, wobei der Wechsel von **rechnend** zu **bereit** unfreiwillig geschehen kann, wenn eine externe Unterbrechung den Zustandswechsel auslöst.

Bezugnehmend auf die Skizze kann der Lebensweg eines Prozesses nachvollzogen werden. Vor dem Beginn seines Daseins steht obligatorischerweise die Erzeugung und am Ende die Zerstörung der Task. Vor Beginn und nach dem Ende ist deshalb der Prozeß **nicht existierend**. Wird der Task erschaffen, geht er in den Zustand eines **ruhenden** Prozesses über. Hier besteht die Möglichkeit, ihn **einzuplanen**, damit er auf dem Prozessor zu Ausführung kommt. Während der Existenz eines Prozesses kann dieser seinen Status mehrfach wechseln und dabei zwischen den Zuständen **lauffähig**, **rechnend** und **wartend** wandern. Hat er seine Aufgabe erfüllt, wird er sich beenden und in den **ruhenden** Zustand übertreten. Von hier aus kann er erneut starten oder wird durch seine Zerstörung vollständig aus dem Kreislauf entfernt.

Die Zustände der Prozesse müssen nicht explizit mit im Kontext des Prozesses gespeichert werden, da sie zumeist aus dem momentanen Aufenthaltsort des Prozesses im System ablesbar sind. Wenn eine Task **lauffähig oder bereit** ist,

wird sie auf der Warteliste der aktiven Prozesse zu finden sein. Weiterhin wird der momentan **rechnende** auf dem Prozessor anzutreffen sein und ein blockierter Prozeß steht in einer, seiner Blockierung entsprechenden, Liste für **wartende** Tasks. Ein **ruhender** Faden ist somit kein Bestandteil des Kreislaufes, wodurch auch dieser Zustand nicht mitprotokolliert werden braucht.

Prozesse müssen in ihrem Kontext dennoch Daten speichern, die das Einplanen von Rechenzeit auf der CPU ermöglichen. Jene Daten werden auch als Prozeßparameter oder Spezifika bezeichnet. Der Abschnitt 2.1.3 beschäftigt sich mit dem Gebiet ausführlicher.

Die Zustandsdiagramme dienen im allgemeinen dem Verständnis des Lebenszykluses eines Prozesses, womit sie eine hervorragende Möglichkeit bieten, auf einem abstrakten Niveau Tasks darzustellen und zu untersuchen.

2.1.2 Sporadisch vs. Periodisch vs. Aperiodisch

Aus dem vorherigen Abschnitt 2.1.1 über Prozeßzustände ist zu entnehmen, daß Prozesse im Verlaufe ihres Lebens verschiedene Zustände teilweise auch mehrfach durchlaufen können. Dies schließt auch einen erneuten Start nach dem Vollenden der Aufgabe mit ein. So gibt es verschiedene Erscheinungsformen von Prozessen bezüglich der Anzahl der Aktivierungen. Grob gesehen, sind nur zwei Varianten zu unterscheiden. Wird ein Prozeß immer wieder zyklisch aktiviert, handelt es sich um eine periodischen Task. Hingegen werden Prozesse, die nicht zyklisch abgeleistet werden, als sporadisch oder aperiodisch bezeichnet.

Sporadische Tasks :
Sporadische Tasks treten nicht regulär auf. Es wird eine obere Schranke bezüglicher der Häufigkeit ihres Aufrufes während der Laufzeit des Systems vorausgesetzt.

Aperiodische Tasks :
Aperiodische Tasks besitzen keine obere Schranke bezüglich der Häufigkeit ihres Aufrufes.

Periodische Tasks :
Periodische Tasks werden mit einer bestimmten Frequenz f regelmäßig aktiviert. (Periode $= 1/f$)
Das Zeitintervall Δp definiert für eine periodische Task den Rahmen ihrer j-ten Ausführung.

Etwas feiner betrachtet, ist der Unterschied zwischen periodischen und aperiodischen Prozessen nicht sehr groß. Sie differieren nur durch die Art des Ereignisses, durch welches sie aktiviert werden. Bei periodischen Tasks ist dies ein Zeitsignal, wohingegen die aperiodischen Prozesse durch externe Ereignisse in Gang gesetzt werden. Wenn ein externes Signal mit der Regelmäßigkeit einer Uhr auftritt, ist

ein aperiodischer Prozeß nicht von einem periodischen Prozeß zu unterscheiden. Folglich sind sie quasi gleich modellierbar, denn nur das Aktivierungssignal stellt die Unterscheidungsgrundlage dar.

2.1.3 Prozeßparameter

Ein Prozeß wird durch verschiedenste Parameter charakterisiert. Dabei ist es von der Strategie des Planungsverfahrens abhängig, welche Größen benötigt werden. In einem Echtzeitsystem, wo Steuer- und Regelungsaufgaben in der Zeit erbracht werden, müssen erfahrungsgemäß mehr Informationen über Prozesse vorhanden sein. In einem reinen Batchsystem ist dies nicht der Fall und grundlegende Daten sind ausreichend, da die Prozesse meist keinen zeitlichen Anforderungen unterliegen. Für die Einplanung von Rechenzeit auf dem Prozessor sind die Prozeßspezifika von unterschiedlicher Bedeutung. Zu den möglichen Größen gehören:

Prozeßtyp : P
: Spiegelt die Beschreibung einer Aufgabe wieder, wie sie auf Programmebene modelliert ist. Ein Prozeßtyp ist beispielsweise die Abfrage eines Sensors und ein anderer Typ schließt die Berechnung mit anschließender Steuerung eines Aktors ein.

Prozeß : P_i
: Eine Aufgabe kann mehrfach in verschiedenen Programmen zur Anwendung kommen. Dabei verfügt das jeweilige Prozeßobjekt P_i über eigene Daten, die aus der anwendungsspezifischen Ebene herrühren.

Die j-te Ausführung des Prozeßobjektes : P_i^j
: Das Prozeßobjekt kann nach der Beendigung seiner Aufgabe erneut gestartet werden. Für periodische und aperiodische Tasks ist dies besonders wichtig.

Bereitzeit (engl. ready time) : r_i
: Entspricht dem frühesten Zeitpunkt, an dem der Prozessor dem Prozeß P_i zugeteilt werden darf.

Ausführungszeit (engl. execution time) : Δe_i
: Die Zeit entspricht der reinen Rechenzeit des Prozesses. Es werden die Eingabedaten zugrunde gelegt, bei denen die Ausführungzeit ein Maximum annimmt. Die Ausführungszeit wird dann auch als $\Delta WCET_i$ (engl. Worst Case Execution Time) bezeichnet.

Frist (engl. deadline) : d_i
: Die Größe charakterisiert den Zeitpunkt, zu dem ein Prozeß seine Ausführung beendet haben muß.

Startzeit (engl. starting time) : s_i
Entspricht dem Zeitpunkt an dem der Prozeß seine Berechnung auf dem Prozessor beginnt.

Abschlußzeit (engl. completion time) : c_i
Die Ausführung des Prozesses durch den Prozessor endet zu diesem Zeitpunkt.

Zwischen den Paramtern existieren Beziehungen, die in Form von Ungleichungen dargestellt werden können. So ist die allgemeine Beziehung zwischen der Start- und Abschlußzeit sowie der Frist bei unterbrechbaren Prozessen folgende:

$$s_i + \Delta e_i \leq d_i \tag{2.1}$$

Dabei ist die Ausführungszeit auf das Intervall $[s_i, c_i]$ verteilt.

Für periodische Prozesse sind diese Parameter noch nicht ausreichend, daher werden diese leicht modifiziert verwendet. Des weiteren fehlt die Periode selbst als beschreibende Größe.

Priode (engl. period) : Δp_i
Für einen periodischen Prozeß definiert die Zeitspanne Δp_i den Rahmen seiner j-ten Ausführung P_i^j.

Durch die Periode des Prozesses ist ab der ersten Bereitzeit r_i^1 jede weitere Bereitzeit festgelegt durch:

$$r_i^j = (j-1)\Delta p_i + r_i^1 \qquad j \geq 1 \tag{2.2}$$

Das Periodenende legt gleichzeitig die Frist für die Prozeßausführung fest.

$$d_i^j = j\Delta p_i + r_i^1 \qquad j \geq 1 \tag{2.3}$$
$$= r_i^{j+1} \tag{2.4}$$

Priorität (engl. priority) : *prio*
Diese Größe wird aus den oben beschriebenen Parametern errechnet, kann aber auch einfach auf einen Wert festgesetzt werden. Es wird zwischen zwei Varianten der statischen und der dynamischen Priorität unterschieden. Im dynamischen Fall kann sich die Größe während der Laufzeit des Systems ändern, wohingegen die statischen Prioritäten bei der Konstruktion des Systems festgelegt werden.

Von der Anwendung müssen die Bereitzeit, die Ausführungszeit, die Frist und die Perioden als Eingabedaten vorgegeben werden, damit eine Planung erfolgen kann. Bei der Planung werden dann die Start- und Abschlußzeiten und damit verbunden die Prozeßprioritäten berechnet. Dadurch entsteht ein Plan, der durch den Dispatcher abgearbeitet werden kann.

2.2 Planungswesen

Zunächst muß erst einmal definiert werden, welcher Art die zu verplanenden Betriebsmittel sind. Eine Planung kann unter anderem für Prozessor-, Speicher- und Kommunikationsressourcen vorgenommen werden. Allgemein wird unter der Planung ein Zusichern des Betriebsmittels an einen Prozeß verstanden. Das Einplanen von Ressourcen kann dabei unterschiedlich erfolgen, abhängig von der Tatsache, unter welchen Bedingungen das System arbeitet. Insbesondere bei der Entwicklung von System mit Echtzeitplanung (Abschnitt 2.2.2) ist dieser wichtige Punkt zu beachten.

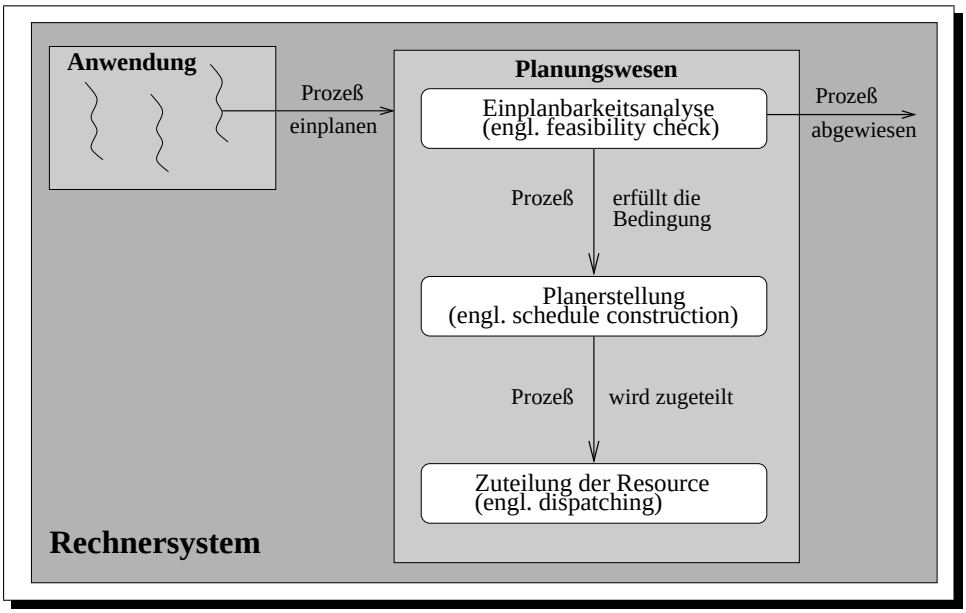


Abbildung 2.4: Vorgänge innerhalb des Planungswesens

Prinzipiell kann die Planung in drei mehr oder minder anspruchsvolle Phasen aufgeteilt werden. Die Darstellung 2.4 zeigt schematisch den Aufbau des Planungswesen, wobei die drei Ebenen deutlich erkennbar sind. Am Anfang steht der Einplanbarkeitstest, der Auskuft gibt, ob ein Prozeß einplanbar ist. Erfüllt er die gestellten Bedingungen nicht, wird er abgewiesen. In einigen Systemen schließt sich hier eine Fehlerbehandlung an, die diese Situation weiter verarbeitet. Kann die Task erfolgreich geplant werden, ist der nächste Schritt die Planerstellung. Dadurch wird der Prozeß in den Plan und damit in die Zuteilungsmenge für das begehrte Betriebsmittel aufgenommen. Als letzter und finaler Schritt steht das Bereitstellen der Ressource für den Prozeß, wodurch dieser seine Arbeit voranbringen kann.

Bei verschiedenen Planungsstrategien und -algorithmen fallen die Planungsschritte unterschiedlich stark ins Gewicht. In einigen Fällen wie etwa in einer einfachen Stapelverarbeitung ist ein Einplanbarkeitstest nicht nötig, bzw. wird er

immer positiv abgeschlossen. In Echtzeitumgebungen kann der Test hingegegen einige Zeit in Anspruch nehmen, weil das Zeitverhalten eines neu einzuplanenden Prozesses gegen sämtliche im System enthaltenen Tasks abgeglichen werden muß. Die Planerstellung hängt ebenfalls vom Algorithmus und der damit verbundenen Strategie ab. Die Zuteilung der Ressource gestaltet sich sehr einfach, da beispielsweise beim Prozessor einfach ein Prozeß auf der CPU zu Ausführung gebracht wird. Dies ist bei allen Prozeßplanungsverfahren äquivalent.

2.2.1 Planungsverfahren

Bei den unterschiedlichsten Planungsverfahren sind rechenbereite Prozesse meist in einer Liste aufgeführt, die in einer bestimmten Reihenfolge sortiert vorliegt. Die Prozesse können z.B. nach Ankunftszeiten oder aber nach Prioritäten geordnet werden. Abhängig vom Verfahren ist auch das Verhalten der Prozesse bezüglich der Unterbrechbarkeit. Aus der Vielzahl von möglichen Verfahren und deren Kombinationen sollen hier nur einige wenige kurz vorgestellt werden.

2.2.1.1 Ankunftszeitorientierte Planung

Ankunftszeitorientierte Planung bedeutet, daß als Kriterium für die Planungsentscheidung die Ankunftszeit der Prozesse beim Planungswesen herangezogen wird. Generell sind zwei Varianten möglich. Einerseits kann das System Prozesse mit früher Ankunftszeit bevorzugen und deren Ausführung voranbringen, andererseits ist auch eine Begünstigung später aufkommender Prozesse denkbar. Die beiden Verfahren werden wie folgt umgesetzt:

FCFS : FCFS[5] (**F**irst **C**ome **F**irst **S**erve)
Der Prozeß, welcher als erster rechenbereit gesetzt wird, bekommt die CPU zur Verfügung gestellt. Alle später eintreffenden Tasks werden in einer Warteschlange aufgehalten, die nach dem Prinzip FIFO[6] (**F**irst **I**n **F**irst **O**ut) sortiert vorliegt, wodurch die weitere Abarbeitungsreihenfolge festgelegt ist.

LCFS : LCFS[7] (**L**ast **C**ome **F**irst **S**erve)
Der Task, der als letzter das Planungswesen betritt, bekommt den Prozessor zur Ausführung zugeteilt. Die bereits vor ihm im Planungswesen befindlichen Prozesse werden in einem Kellerspeicher aufbewahrt, der nach dem Prinzip LIFO[8] (**L**ast **I**n **F**irst **O**ut) geordnet ist. Ob der monentan laufende Prozeß sofort verdrängt wird oder ob er seine Berechnung beenden kann, hängt maßgeblich vom Unterbrechungsverhalten der beteiligeten Prozesse ab.

[5] FCFS(engl.) = Wer zuerst kommt, mahlt zuerst.
[6] FIFO(engl.) = Erster rein, erster raus
[7] LCFS(engl.) = Wer zuletzt kommt, mahlt zuerst.
[8] LIFO(engl.) = Letzter rein, erster raus

Die ankunftszeitorientierte Planung hat eine sehr große Verbreitung gefunden, weil in vielen Fällen eine einfache Steuerung und Koordinierung der Prozesse im System völlig ausreichend ist. Ein weiterer großer Vorteil dieser Strategien ergibt sich durch die leichte Umsetzbarkeit und den geringen Laufzeitaufwand während der Arbeitsphase des Systems.

2.2.1.2 Prioritätenorientierte Planung

Beim prioritätsorientierten Planen kommt im Gegensatz zum ankunftsorientierten Verfahren als Planungskriterium eine den Prozessen zugeordnete Priorität zum Tragen. Der am höchsten priorisierte Prozeß verfügt dabei über den Prozessor, wohingegen die niederwertigen auf einer absteigend sortierten Liste auf die Zuteilung der CPU warten.

Zwei Arten von Prioritäten sind zu unterscheiden. Grundsätzlich werden die statischen und die dynamischen Prioritäten gleich verarbeitet. Im statischen Fall besitzt der Prozeß immer die gleiche Priorität. Der Prozeß verändert seine Position innerhalb der Zuteilungsliste des Prozessors nicht mehr, sobald er einmal ordnungsgemäß einsortiert wurde.

Im dynamischen Fall besitzt der Prozeß am Anfang eine Ausgangspriorität, die sich im Verlauf seiner Existenz ändern kann. Dadurch muß immer eine Umsortierung der Zuteilungsliste erfolgen, sobald sich Prioritätswerte einzelner Prozesse ändern. Dies hat einen gesteigerten Laufzeitverbrauch gegenüber der statischen Variante zur Folge.

Da der Anwendungsfall jedoch die Wahl des Verfahren vorschreibt, haben beide Varianten ihre Berechtigung.

2.2.1.3 Ereignisorientierte Planung

Unter einem Ereignis wird allgemein eine Unterbrechung eines Prozesses durch eine externe Quelle verstanden. Die ereignisorientierte Planung ihrerseits soll das Einplanen der durch das Ereignis ausgelösten Aktionen gewährleisten. Um beispielsweise die Netzwerkhardware optimal auszunutzen, muß auf eine Nachricht möglichst schnell reagiert werden, indem z.B. der nachrichtenverarbeitende Prozeß aktiviert wird. Sobald er mit seiner Aufgabe fertig ist, gibt er den Prozessor wieder an den vormals rechnenden Prozeß ab, um auf neue Nachrichten zu warten. Müßte er hingegen auf die Beendigung der laufenden Task warten, ergäbe sich eine Verzögerung, wodurch in jener Zeit die Netzwerkhardware nicht genutzt werden könnte. Eine schlechte Auslastung des Netzwerkes und der damit verbundene Bandbreitenverlust wäre die Folge. Ereignisgesteuerte Prozesse sind folglich in dem Verfahren vorzuziehen, damit eine möglichst hohe Reaktionsfreudigkeit des Systems erreicht wird.

2.2.1.4 Zeitscheibenorientierte Planung

Das zeitscheibenorientierte Verfahren ist sehr eng mit dem ereignisorientierten verwandt. Das Ereignis, das ein Umschalten von Prozessen auslöst, ist eine abgelaufene Uhr. Der Unterschied zum vorher betrachteten Verfahren ist der Umstand, daß zum Zeitpunkt des Ereignisses noch nicht bekannt ist, zu welchem Prozeß gewechselt wird. Mit einem Blick auf die Warteliste kann die nächste zuteilungsbereite Task gefunden werden, woduch eine Verdrängung zu Gunsten des neuen Prozesses stattfindet. Der verdrängte Prozeß wird wieder auf die Liste der rechenbereiten Prozesse gesetzt. Hierdurch ist gewährleistet, daß er in der Zukunft den Prozessor erneut zur Berechnung erhält. Als letzter Schritt wird die Uhr neu aufgezogen, damit für den neuen Prozeß eine vollständige Zeitscheibe zur Verfügung steht.

2.2.1.5 Mischformen

Die beschriebenen Verfahren der ankunfts-, prioritäts-, ereignis- und zeitscheibenorientieren Planung lassen sich auch kombinieren. So ist in interaktiven Systemen, an denen Benutzer arbeiten, meist eine Mischform eines Planungsalgorithmuses anzutreffen. Es werden Systeme benutzt, die hohe Prioritäten für ereignisgesteuerte Prozesse vergeben und niedrige, für rechenintensive Tasks. Innerhalb einer Prioritätsklasse wird meist nach dem FCFS-Prinzip gearbeitet. Oft wird dies noch mit einem Zeitscheibenmechanismus kombiniert, der eine Umschaltung zu einem anderen Prozeß mit der gleichen Priorität nach Ablauf erbringt.

Wie zu sehen ist, kann keine starre Sicht auf das Planungswesen eingenommen werden, da je nach Anforderung andere Lösungswege eingeschlagen werden. Daher ergibt sich eine Flut von Algorithmen, die sich oft nur in kleinen Details unterscheiden, aber nach außen verschiedene Charakteristika aufweisen.

2.2.2 Echtzeitplanung

Bei der Echtzeitplanung müssen Prozesse eingeplant werden, die Zeitschranken unterliegen. Es gibt verschiedenen Verfahren, die es ermöglichen sollen, dem Rechnung zu tragen. Grob sind diese in Schedulingklassen charakterisiert, wobei Planungsalgorithmen folgendermaßen beschrieben und unterteilt werden können:

Statisches Planen : wird offline durchgeführt, wobei ein vollständiger Plan zur Konstruktionszeit des Systems erstellt wird. Dieser wird zur Laufzeit durch einen Dispatcher abgearbeitet. Die Daten für das Scheduling müssen bei dem Verfahren von vornherein vollständig bekannt sein.

Dynamisches Planen : erfolgt online, da die Daten für das Planen erst zur Laufzeit des Systems vorliegen. Planungsentscheidungen müssen jeweils nach dem Beenden einer Task, nach dem eine Task ihre Bereitzeit erreicht hat oder eine neue Task nach Abarbeitung verlangt, getroffen werden.

Die Ergebnisse der Planung können dem System als vollständiger Plan oder in Form von Regeln (Prioritäten) übergeben werden. Es wird von expliziter vs. impliziter Planung gesprochen.

Für die statische Planung spricht, daß die Einplanbarkeit von Prozessen bewiesen werden kann, da die Planbarkeitsanalyse offline erfolgt. Weiterhin ist nur ein minimaler Overhead zur Laufzeit der Anwendung zu erbringen, da der Plan beispielsweise in Form einer Tasktabelle mit nach Startzeiten geordneten Einträgen vorliegt. Die Planbarkeit kann bei einer großen Prozeßmenge aber schwer zu lösen sein, da der gesamte Lösungsraum durchsucht werden muß. Die optimale Lösung zu finden, ist sehr komplex und fällt in den Bereich der NP-Vollständigkeit [Wag94]. Meist werden aus diesem Grund nur statische nicht unterbrechbare Prozesse betrachtet, da der Aufwand der Beweisbarkeit sehr hoch ist.

Die dynamische Verfahrensweise besitzt diese Einschränkung jedoch nicht, weshalb sie einen Mix aus periodischen, sporadischen, unterbrechbaren und nicht unterbrechbaren Prozessen gestattet. Zur Laufzeit des Systems werden die Prozesse durch eine Einplanbarkeitsanalyse verarbeitet und eingeplant. Die Koordination der Prozesse erfordert jedoch einen gewissen Aufwand, welcher mitzuplanen ist. Wenn die Planung auf einem eigens dafür vorgesehenen Coprozessor vorgenommen wird, stellt dies aber dennoch keinen Nachteil dar.

In den nun folgenden Abschnitten werden einige ausgewählte Echtzeitplanungsalgorithmen vorgestellt und einer kurzen Betrachtung unterzogen, in der die Funktionsweise und der Bedarf an Laufzeit erklärt wird. Für weitere Informationen sowie den Beweisführungen einiger Aussagen sei auf die Bücher [Hüs94, Hüs95, ZA95] verwiesen.

2.2.2.1 Planen durch Suchen

Das Verfahren „Planen durch Suchen“ wird nur auf statische nicht unterbrechbare Prozesse angewandt. Es erstellt offline einen Plan, der zur Laufzeit nur noch von einem Dispatcher abgearbeitet wird. Das Verfahren durchsucht den gesamten Lösungsraum nach einem geeigneten Plan. Bei einer Prozeßmenge von $P =\mid n \mid$ ergibt das einen Aufwand von $O(n!)$, um einen optimalen Plan zu finden, der die Prozesse korrekt ausführt. Somit ist das Problem grundsätzlich NP-vollständig.

Das Verfahren eignet sich in Umgebungen, in denen harte Echtzeitanforderungen gestellt werden. Es dürfen keine Prozesse dynamisch neu erzeugt werden und die Parameter der Prozesse sind offline bekannt und ändern sich auch während der Laufzeit des Systems nicht.

2.2.2.2 Planen nach Fristen

Eine der am verbreitetsten Planungsstrategien im Bereich der Echtzeitplanung ist das „Planen nach Fristen“. Das liegt an der Einsetzbarkeit der Strategie in vielen unterschiedlichen Echtzeitumgebungen. So läßt sie die Anwendung auf un-

terbrechbare und nicht unterbrechbare Prozesse zu, und es ist als statisches sowie dynamisches Planungsverfahren einsetzbar. Ist die dynamische Variante im Einsatz, sind selbst dynamische Prozeßerzeugungen möglich.

Das Verfahren teilt den Prozessor dem rechenbereiten Prozeß zu, dessen Frist den kleinsten Wert anzeigt. Ist kein Prozeß bereit, ist die CPU untätig (engl. idle). Auf unterbrechbare Prozesse kann die Strategie zu jedem beliebigen Zeitpunkt angewendet werden. Bei nicht unterbrechbaren Prozessen ist eine erneute Planung immer erst nach einem Blockierungs- oder Beendigungsvorgang eines Prozesses möglich.

Als Prozeßparameter muß im allgemeinen nur die Frist d_i bekannt sein, da während der Planung nur diese als Kriterium zur Entscheidung herangezogen wird. Im Falle einer statischen offline Prozeßplanung kann mit Hilfe der Laufzeit Δe_i das Einhalten der Zeitschranken berechnet und bewiesen werden.

2.2.2.3 Planen nach Spielräumen

Das „Planen nach Spielräumen“ kann ebenso wie das „Planen nach Fristen“ auf unterbrechbare und nicht unterbrechbare Prozesse angewendet werden. Das Verfahren existiert nur in einer dynamischen Version, wodurch aber auch die Möglichkeit besteht, Prozesse dynamisch zu erzeugen und einzuplanen. Als Prozeßparameter benötigt die Strategie die Frist d_i und die Laufzeit Δe_i der Prozesse, aus denen der Spielraum (engl. laxity) zum Zeitpunkt t folgendermaßen berechnet wird.

$$L_i(t) \quad = \quad d_i(t) - \Delta e_i(t) \tag{2.5}$$

Der Prozeß mit dem kleinsten Spielraum erhält dabei den Prozessor. Der Spielraum stellt somit die dynamische Priorität dar. Bei unterbrechbaren Prozessen kann zu jedem Zeitpunkt die Neuberechung der Spielräume erfolgen, wodurch möglicherweise eine Prozeßumschaltung initiiert wird. Im Falle von nicht unterbrechbaren Prozessen geschieht dies immer nach dem Abschluß oder der Blockierung einer Task.

Das Einplanen von Prozessen verbraucht einen gewissen Anteil an Laufzeit, weshalb diese Zeit mitgeplant werden muß. Leider ist die Zeit für einen Planungsvorgang nicht kostant und die Bestimmung der maximalen Dauer ist maßgeblich von den im System vorhandenen Prozessen abhängig. Um eine möglichst geringe Belastung des Arbeitsprozessors zu gewährleisten, wird in solchen Fällen oft ein eigener Prozessor für die Planung vorgesehen.

2.2.2.4 Planen nach monotonen Raten

Das „Planen nach monotonen Raten“ ist ein Verfahren für unterbrechbare und nicht unterbrechbare Prozesse, wobei diesen eine statische Priorität zugeordnet wird. Vorrangig werden periodische Prozesse betrachtet, da die Priorität abhängig

von der Periode vergeben wird. Sporadische bzw. aperiodische Prozesse sind nur bedingt planbar. Das Verfahren gehört zu der Klasse der Planungsalgorithmen, bei denen kein expliziter Plan existiert, sondern durch die Vergabe von Prioritäten ein impliziter Plan aufgestellt wird.

Die Vergabe der Prioritäten an die Prozesse erfolgt offline. Die Bezugsparameter sind die Frist d_i, die Laufzeit Δe_i sowie die Periode Δp_i, wobei die Priorität wie folgt berechnet wird.

$$prio \quad = \quad 1/\Delta p_i \tag{2.6}$$

Mit Hilfe der Frist und der Laufzeit kann offline berechnet werden, ob alle Prozesse ohne Zeitschrankenüberschreitungen verarbeitet werden können.

Obwohl eine dynamische Prozeßerzeugung zur Laufzeit nicht gestattet ist, besitzt das Verfahren eine große Bedeutung im Anwendungsbereich der Echtzeitsysteme. Dafür sprechen einerseits die einfache Handhabung und andererseits die einfache programmtechnische Umsetzung des Verfahrens. Bei der Entscheidung über die Prozessorzuteilung müssen lediglich die Priorität des laufenden Prozesses mit dem Prioritätswert des gerade bereit werdenden verglichen werden. Durch den einfachen Mechanismus erklärt sich auch, warum dieses Verfahren sehr weit verbreitet ist.

2.3 Merkmalsmodelle

Die oben beschriebenen Planungsverfahren zeichnen sich durch sehr viele Gemeinsamkeiten aus. Zumeist sind nur geringfügige Unterschiede bezüglich der Entscheidungsfindung über die Zuteilung der Ressource Prozessor zu finden. Daher sollen die gemeinsamen Aspekte möglichst zusammen verwirklicht werden, so daß Softwarebausteine wiederverwendet werden können. Die Unterschiede sind zu kennzeichnen und so zu kapseln, daß ein Austauschen und damit verbunden eine Änderung des Systemverhaltens möglich ist.

Das Modellieren wiederverwendbarer Software gestaltet sich in der Softwareentwicklung aber als äußerst schwieriger und komplexer Vorgang [CE99, CE00]. Beim Erstellen einzelner Implementationen sind eine Reihe von Entwurfsentscheidungen zu treffen. Wird beispielsweise ein Planungswesen für Prozesse betrachtet, muß entschieden werden, welcher Art die Prozesse sind, mit Hilfe welcher Strategie diese verarbeitet werden, wie die Verwaltung geschieht (Liste, Queue, Array,...) und wie die Ausführung bewerkstelligt wird. Soll hingegen Wiederverwendbarkeit angestrebt werden, dann sind die Entwurfsentscheidungen veränderbar auszulegen, weil in unterschiedlichen Kontexten unterschiedliche Anforderungen gestellt werden. Es wird ein flexibles und veränderbares Design erzwungen, das Raum für Konfiguration läßt. Jede Designentscheidung entspricht einem Fea-

ture[9], welches im jeweiligen Kontext eine wiederverwendbare konfigurierbare Anforderung repräsentiert.

Die Merkmale eines Systembereiches werden in sogenannten Merkmalsdiagrammen oder auch Merkmalsmodellen angeordnet. Erstmals wurde diese Methode in der Feature-Oriented Domain Analyse (FODA) eingeführt, welche eine Vorgehensweise vorschlägt, um allgemeine und variable Bestandteile eines Systembereiches zu beschreiben [KCH+90]. Das Modell stellt dabei den konfigurierbaren Aspekt eines Konzept auf einem hohen Abstraktionsniveau heraus. So kann auf einer abstrakten Ebene die Modellierung des Systembereiches erfolgen. Die Modelle zeigen dabei die Art der Variabilität oder die Allgemeinheit der Eigenschaften eines Teilbereiches an. So kann mit Hilfe eines solchen Modelles auch eine Softwarefamilie beschrieben und gleichzeitig aufgezeigt werden, wie sie konfiguriert werden kann.

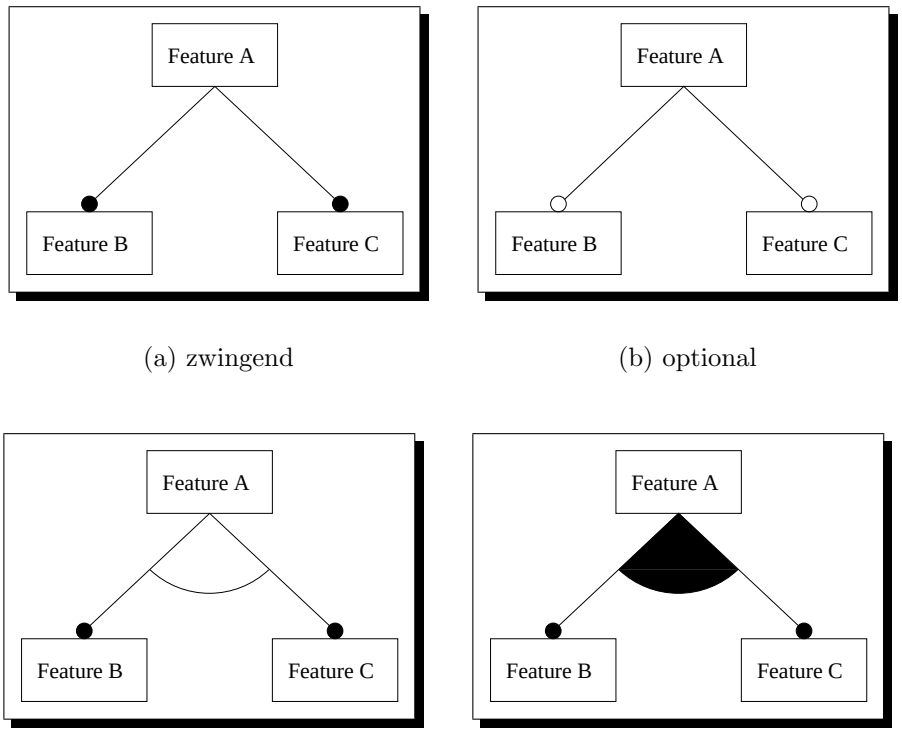
Verschiedene Arten der Merkmale sind zu unterscheiden, da Eigenschaften von Systembereichen zwingend, optional, alternativ bzw. oder sein können. Im Folgenden werden die einzelnen Erscheinungsformen kurz in Bild und Wort erläutert.

Zwingende Merkmale (engl. mandatory features): sind immer Bestandteil des Systems, wenn das Elternmerkmal ausgewählt wurde. Das zwingende Vorhandensein wird im Modell durch einen ausgefüllten Kreis an den Anforderungen gekennzeichnet. Als Beispiel sei die Abbildung 2.5(a) angegeben, in der ersichtlich ist, daß die Eigenschaften `B` und `C` immer vorhanden sein müssen, sobald das Merkmal `A` enthalten ist.

Optionale Merkmale (engl. optinal features): können, wie der Name schon ausdrückt optional, also zusätzlich im System erscheinen. Diese Merkmalsart kann somit im Zielsystem vorhanden sein, wobei es dann zusätzlich ausgewählt werden muß. Prinzipiell sind die Merkmale `B` und `C` nicht erforderlich, um die grundsätzliche Aufgabe der Anforderung `A` zu bewerkstelligen, aber sie zeigen eine mögliche Erweiterung auf, wodurch `A` andere Eigenschaften oder Verfahrensweisen annehmen kann. Das graphische Pendant 2.5(b) wird in Merkmalsdiagrammen mit Hilfe eines unausgefüllten Kreises an den Mermalen dargestellt.

Alternative Merkmale (engl. alternativ features): sind im Bild 2.5(c) illustriert. Gekennzeichnet werden sie durch einen ausgefüllten Kreis an den Anforderungen sowie einem Bogen zwischen den Kanten, die zu ihnen führen. Bei alternativen Merkmalen ist eine Auswahl aus einer Menge von möglichen Optionen zu treffen, da im Zielsystem immer eines der Mermale vorhanden sein muß. Daraus folgt: Ist Merkmal `A` Teil des System, muß entweder `B` oder `C` gewählt werden, da ein Erbringen der Aufgabe von

[9] feature(engl.) = Merkmal bzw. Eigenschaft.



(a) zwingend (b) optional

(c) alternativ (d) oder

Abbildung 2.5: Merkmalsformen

`A` ohne die zu Hilfenahme eines der alternativen Merkmale nicht möglich ist.

Oder Merkmale (engl. or features): besagen, daß mindestens eine Anforderung Bestandteil des Systembereiches sein muß, wenn die Elterneigenschaft vorhanden ist. Das schließt nicht aus, daß mehrere Anforderungen ausgewählt werden können. Im System ist somit immer wenigstens eine Komponente vorhanden. Die Darstellung 2.5(d) zeigt, daß die Anforderungen ausgefüllte Kreise tragen und zwischen den Verbindungslinien ein ausgefüllter Bogen zu erkennen ist. Die Anforderung `A` als Elterneigenschaft erbringt ihre Aufgabe unter Nutzung eines der Merkmale `B` oder `C`, oder es gebraucht gar beide für die Lösungsbewältigung.

Durch zusätzliche semantische Bedingungen (constrains, restrictions, dependency rules) kann Einfluß auf die Zusammenhänge von Anforderungen genommen werden. Sind z.B. Eigenschaften unvereinbar, wird eine Bedingung formuliert, die ein gemeinsames Auftreten verhindert. Für tiefergehende Betrachtungen auf dem Gebiet der Merkmalsmodelle wird auf einschlägige Literatur [CE99, CE00, KCH+90] verwiesen.

Die vorangegangenen Betrachtungen geben dem Entwickler ein Hilfsmittel für die Visualisierung von Programmfamilien an die Hand. Dadurch kann dem Nutzer auf einer allgemeinen verständlichen Ebene das System vermittelt werden, ohne tiefe Einblicke in den jeweiligen Systembereich und dessen Verhalten freizugeben. Die Konfiguration kann auf diesem Niveau erfolgen, wodurch keine spezifischen Kenntnisse über die Implementierung, sowie die Zusammenhänge von Systemteilen vorhanden sein müssen.

Kapitel 3

Realisierung

In den folgenden Abschnitten wird ein Überblick über die Realisierung des konfigurierbaren, maßschneiderbaren Planungswesen für Betriebssystemfamilien gegeben. Insbesondere sollen Details des Entwurfes beleuchtet werden, wobei aus verschiedenen Sichtweisen das Planungswesen eine eingehende Betrachtung erfährt. Nachdem der Entwurf als abgeschlossen erachtet werden kann, schließt sich ein Klassendesign sowie die Implementierung nahtlos an. Hier werden einige Besonderheiten, sowie ebenfalls alternative Herangehensweisen angesprochen. In dem Abschnitt über die Konfiguration wird einmal grob gezeigt, wie das Planungswesen konfiguriert werden kann. Nachfolgend werden zwei Szenarien von Beispielanwendungen betrachtet, wodurch noch einmal ersichtlich wird, wie variabel anpaßbar das System sich darstellt.

3.1 Entwurf

Der Entwurf eines Softwaresystems ist kein trivialer Vorgang, wenn dieses sich noch dazu flexibel verschiedenen Situationen anpassen können muß. Das Spektrum der erzeugbaren Systeme soll dabei vom Anwendungsbereich der tiefsten eingebetteten Systeme bis hin zu Stapelverarbeitungssystemen reichen. Daraus ergibt sich, daß die Systemsoftware skalierbar und konfigurierbar ausgelegt sein muß, um den teilweise restriktiven Anforderungen der Einsatzbereiche Rechnung zu tragen. Funktionalität, die nicht tatsächlich eine Benutzung erfährt, sollte auch nicht Bestandteil des endgültigen Systems sein. Die Anwendungsebene ist optimal zu unterstützen.

Beim Entwurf des Planungswesen soll der Grundgedanke der Wiederverwendbarkeit im Vordergrund stehen. Daher wird es als Familie entworfen, bei der die gleiche Idee verfolgt wird. Sukzessive werden von den Grundfunktionen ausgehend immer neue Funktionen Schicht für Schicht hinzugefügt. So werden Basisfunktionen, die als Grundbausteine in der Softwarehierarchie stehen, immer wieder eine Verwendung finden. Die verschiedenen Familienmitglieder entstehen

durch Spezialisierungen einzelner Funktionen, wobei auch mehrere Spezialisierungen für eine Basisfunktion existieren können. Gerade an diesen Punkten kann eine Konfiguration durch Selektion der favorisierten Implementierung erfolgen.

Beginnend mit Merkmalsmodellen sollen im Entwurf die Konfigurationsmöglichkeiten des Systems aus Sicht der Anwendung gezeigt werden. Eine Umsetzung der betrachteten Eigenschaften sowie die Bestimmung der Schnittstelle zum Planungswesen schließt sich an. Darauf aufbauend werden im weiteren Verlauf die funktionalen Einheiten sowie Konfigurationspunkte vorgestellt, um einen tieferen Einblick über das Zusammenspiel der Funktionen zu geben. Abschließend wird die Umsetzung mit Hilfe der Objektorientierung vorgenommen, welche in einem Klassendesign mündet.

3.1.1 Merkmalsmodelle der Systembereiche

Durch Merkmalsmodelle werden die variablen und die allgemeinen Merkmale eines Systems beschrieben. Das Planungswesen und seine einzelnen Bestandteile können nun als Teil eines solchen Systems verstanden werden. In den folgenden Abschnitten wird vom Standpunkt der Anwendung ausgehend das Prozeßbild modelliert. Dabei werden in Merkmalsdiagrammen die verschiedenen Ausprägungen eines Prozesses sowie Strategien zu deren Koordinierung aufgezeigt.

3.1.1.1 Merkmalsmodell Prozeß

Durch unterschiedlichste Anforderungen der Anwendungsebene an das System ist die Erscheinungsform des Prozesses von Fall zu Fall sehr differierend. So kann es für die Anwendung bzw. für die Erbringung der Lösung einer Aufgabe ausreichend sein, ein Einprozeßsystem zur Verfügung zu stellen. In einer anderen Situation mit gesteigerten Anforderungen an das System ist hingegen eine Mehrprozeßvariante vorzuziehen, um die Anwendung in ihrer Aufgabenerfüllung optimal zu unterstützen.

Die Anzahl der Prozesse ist ein Grundmerkmal in einem System. Braucht die Anwendung nur einen Prozeßfaden, so sind keine weiteren Information bezüglich des Prozeßmodells nötig. Die Abbildung 3.1 zeigt, daß bei der Eigenschaft *single* keine anderen Merkmale mehr auszuwählen sind. Wird von der Anwendungsebene jedoch ein Mehrprozeßsystem verlangt, sind zusätzliche Eigenschaften der Prozesse zu spezifizieren. Dem Diagramm kann entnommen werden, daß die nächste Ebene zwei zwingende Eigenschaften (*scheduling*, *process count*) und eine optionale Eigenschaft (*idle loop*) aufweist. Das bedeutet, hier ist eine Variabilität gegeben, so daß bereits unterschiedliche Systeme bzw. Konfigurationen erstellt werden können. Beginnend mit der optionalen Eigenschaft ist zu entscheiden, ob immer ein Prozeß im System rechenbereit ist. Kann die Frage positiv beantwortet werden, ist dieses Merkmal nicht auszuwählen. Ist dies jedoch nicht der Fall, muß eine Leerlaufeigenschaft (*idle loop*) vorhanden sein. Daran anschließend

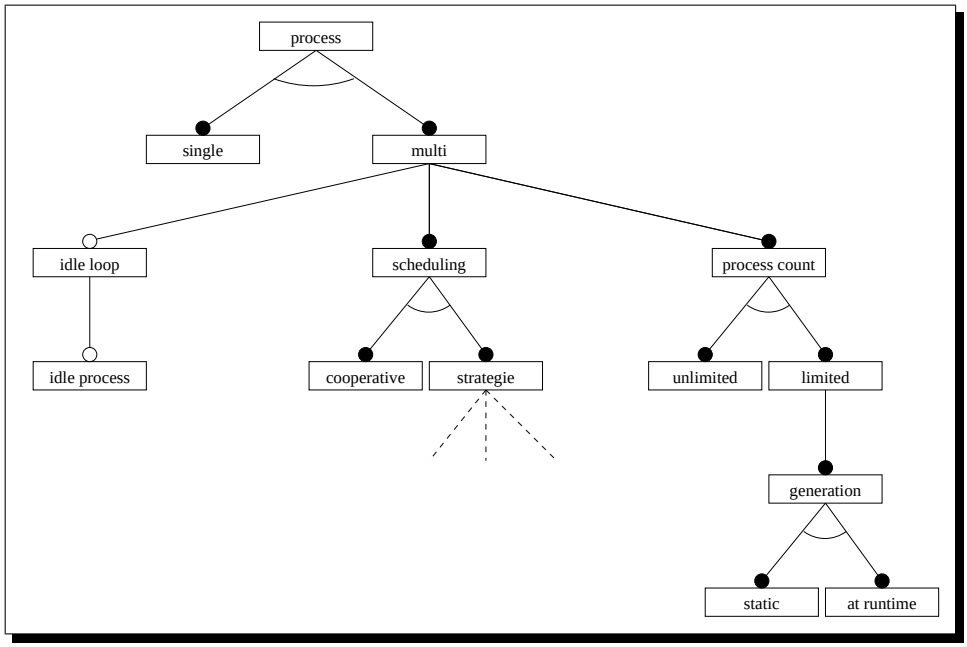


Abbildung 3.1: Das Merkmalsmodell des Planungswesens

kann wiederum eine optionale Eigenschaft (*idle process*) angegeben werden, um die Leerlaufzeit des Prozessors durch einen eigens dafür spezialisierten Prozeß zu nutzen. Das Verhalten der Prozesse zueinander muß ebenfalls dem System bekannt gemacht werden, welches mit der zwingenden Anforderung *scheduling* zum Ausdruck gebracht wird. Hier ist anzuführen, ob Prozesse die Verwaltung der Ressource Prozessor selbst in die eigene Verantwortung nehmen oder ob sie mit Hilfe einer Strategie der Ressource zugeteilt werden. Im Bild spiegeln die alternativen Eigenschaften *cooperative* und *strategie* den Sachverhalt wieder. Zum alternativen Merkmal *strategie* sei noch anzumerken, daß dieses wiederum ein Konzept darstellt, welches im Abschnitt 3.1.1.2 einer eingehenden Betrachtung unterzogen wird.

Aus Sicht der Anwendung muß bei der Betrachtung der Eigenschaften eines Systems auch das zwingende Merkmal der Anzahl der Prozesse angegeben werden. Der Konzeptknoten *process count* spiegelt den genannten Aspekt im Diagramm wieder. Wird der Untergraph etwas genauer untersucht, ist festzuhalten, daß Anwendungen die Prozeßanzahl limitieren können oder aber keine Aussagen über die Höchstgrenze treffen müssen. Als alternative Merkmale sind die Eigenschaften *limited* bzw. *unlimited* anzugeben. Für den Fall der unbegrenzten Anzahl von Prozessen im Endsystem ist per Definition die Generierung der Prozesse immer zur Programmlaufzeit vorzunehmen. Bei der begrenzten Anzahl kann die Art der Erzeugung der Prozesse angegeben werden, weil bereits zur Konstruktionszeit des Systems bekannt ist, wie viele Prozesse maximal im System eingesetzt werden. Die zwingende Eigenschaft *generation* verlangt deshalb die Einstellung

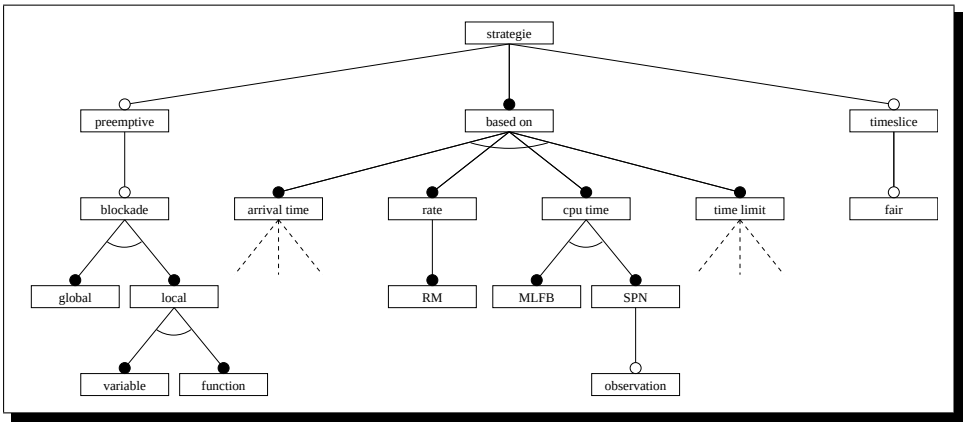


Abbildung 3.2: Das Merkmalsmodell des Bereiches Strategie

der Art der Generierung der Prozesse, wobei durch die alternativen Merkmale *static* und *at runtime* zwei Möglichkeiten zur Auswahl stehen. Wird der Punkt *at runtime* ausgewält, erfolgt die Generierung der Prozesse zur Programmlaufzeit. Fällt die Wahl hingegen zugunsten einer statische Erzeugung aus, so werden Prozesse schon zur Konstruktionszeit des endgültigen Systems kreiert.

3.1.1.2 Merkmalsmodell Planungsstrategien

Nachdem grundlegende Eigenschaften des Systems auf der konzeptmäßig höherliegenden Ebene beleuchtet sind, wird auf das Zusammenspiel der Prozesse näher eingegangen. Wie aus den Abschnitten 2.2.1 und 2.2.2 des Kapitels 2 zu entnehmen, ist das Gebiet der Planungsstrategien ein weites Feld. Viele verschiedene Verfahren sind dort aus den unterschiedlichsten Bereichen anzutreffen. Oftmals unterscheiden sich die Strategien in ihrem Verhalten, weisen aber bei genauerer Betrachtung meist eine große Anzahl an Gemeinsamkeiten auf. So können bei einigen Verfahren die gleichen Parameter für die Steuerung eingesetzt werden.

Die optionalen Eigenschaften *preemptive* und *timeslice* etwa sind für einige Planungsalgorithmen gleich einstellbar. Aus diesem Grund sind sie auf der ersten Ebene als Merkmale verzeichnet, wie dies auch die Abbildung 3.2 des Merkmalsmodelles der Strategien anzeigt. Die Anforderung *preemptive* beschreibt hierbei das Unterbrechungsverhalten der Prozesse und die Eigenschaft *timeslice* ordnet jeder Prozeßausführung eine Zeitscheibe zu. Der Abbildung ist zu entnehmen, daß weitere Einstellungen zu den optionalen Eigenschaften vorgenommen werden können. Beginnend mit der Zeitscheibe kann als zusätzliches optionales Merkmal *fair* angegeben werden. Diese Einstellung erlaubt es jedem Prozeß eine volle Zeitscheibe zu nutzen, sobald *fair* selektiert wurde. Wenn die Option nicht verwendet wird, hat nicht unbedingt jeder Prozeß seinen ihm zustehenden Zeitanteil. Folgendes Szenario beschreibt diesen Fall: Ein Prozeß verbraucht seinen Zeitanteil fast vollständig und wechselt dann freiwillig zu einem anderen. Der neue Prozeß

beginnt mit seiner Ausführung und die Zeitscheibe läuft ab, wodurch sich ein erneuter Prozeßwechsel anschließt. Bei der Einstellung *fair* wird der Zeitgeber jedoch immer nach einem Prozeßwechsel auf den initialen Wert gesetzt, wodurch jeder Prozeß eine komplette Zeitscheibe ausschöpfen kann. Wird von der Anwendung jedoch dieser Mechanismus nicht benötigt, kann bei Prozeßwechseln Zeit eingespart werden und auch ein geringerer Speicherverbrauch ist die Folge. Dadurch kann das System möglicherweise auf noch restriktiveren Plattformen eingesetzt werden, wobei sich meist ein Kostenersparnis ergibt.

Die zweite optionale Eigenschaft *preemptive* beschreibt wie schon erwähnt das Unterbrechungsverhalten der Prozesse. Von der Voreinstellung her werden Prozesse immer unteilbar ausgeführt. Daher kann als optionales Merkmal die Unterbrechbarkeit hinzugefügt werden. Aus der Sicht der Anwendung kann es verschiedene Arten von Prozessen geben, bei denen die Ausführung beispielsweise komplett oder zumindest zeitweise unteilbar ablaufen muß. Infolge solcher Anforderungen ist das optionale Konzept *blockade* mit in das Modell aufgenommen worden. Es läßt Spielraum für die Konfiguration von Prozessen, bei denen Unteilbarkeit benötigt wird. So ist zwischen den beiden alternativen Eigenschaften *global* und *local* zu wählen, wenn eine Blockadeeigenschaft vorhanden sein muß. Die Blockadenkontrolle in der Variante *global* sperrt das gesamte Planungswesen, wodurch kein Prozeßwechsel durch ein externes Ereignis initiiert werden kann. Der unteilbare Prozeß muß freiwillig die Kontrolle über die Ressource Prozessor abgeben. Danach ist die Blockade weiterhin für alle Prozesse gültig, weshalb explizit die Sperrung aufgehoben werden muß.

In der Version der lokalen Sperrung (*local*) wird das Planungswesen nur für die Zeit der Ausführung des unteilbaren Prozesses blockiert. Es bestehen wiederum zwei Möglichkeiten die Anforderungen der lokalen Unteilbarkeit zu gewährleisten. Einerseits kann mit Hilfe einer prozeßlokalen Variablen (*variable*) angezeigt werden, ob Teilbarkeit vorliegt, anderseits kann auch eine spezialisierte Funktion (*function*) diese Aufgabe übernehmen. Ein unteilbarer Prozeß hemmt das Ausführen anderer Prozesse, solange bis er selbstständig den Prozessor abgibt oder die Sperrung des Schedulings zurücknimmt. Nach dem Umschalten ist die Blockade des nun aktiven Prozesses von Bedeutung, wodurch in Abhängigkeit von ihr das Scheduling vorangebracht wird.

Unabhängig von den zuvor beschriebenen Eigenschaften muß weiterhin festgelegt werden, mit Hilfe welcher Planungsstrategie die Prozesse zu verarbeiten sind. Die Verfahren beruhen auf teilweise verschiedenen Grundkriterien für die Planungsentscheidung, wodurch eine Einteilung nach der Art des Kriteriums möglich ist. Unterhalb des Konzeptknotens *based on*[1] (Abb. 3.2) kann die Aufteilung in vier Hauptgruppen betrachtet werden. Als alternative Eigenschaften

[1] based on(engl.) = beruht auf

treten *arrival time*[2], *rate*[3], *cpu time*[4] und *time limit*[5] im Merkmalsmodell auf, wobei sie gleichzeitig die Basiskriterien der Strategien symbolisieren. Die Anwendung bestimmt den Einsatz eines Verfahrens, indem sie aus den alternativen Merkmalen auswählt. Die Konzeptknoten *arrival time* und *time limit* werden in gesonderten Merkmalsmodellen (Abbildungen 3.3 und 3.4) behandelt, weil beide ein sehr umfangreicheres Konfigurationsangebot besitzen. Zu den raten- und prozessorzeitbasierten Verfahren sind in der Abbildung 3.2 bereits die Strategien namentlich aufgeführt. In den folgenden Unterabschnitten wird auf die unterschiedlichen Verfahren gesondert eingegangen. Es sei darauf hingewiesen, daß die oben beschriebenen Eigenschaften wie *preemption*, *timeslice* und andere ebenfalls Merkmale der darzustellenden Verfahren sein können.

Planungsverfahren basierend auf Raten

Das Planungsverfahren basierend auf Raten teilt einem Prozeß eine Priorität gemäß seiner Periode zu. Lange Perioden bedeuten hier kleine Prioritäten und kurze Intervalle dementsprechend hohe Prioritäten. Hieraus ist ablesbar, daß das Verfahren vorwiegend für periodische Prozesse mit fester Frequenz gedacht ist. Soll eine Anwendung diese Verfahren verwenden, wird im Merkmalsmodell die rate-monotonic[6] Strategie (Merkmal *RM*) ausgewählt. Weitere bzw. zusätzliche Eigenschaften sind hier nicht mehr einzustellen.

Planungsverfahren basierend auf Prozessorzeiten

Bei prozessorzeitgesteuerten Verfahrens kann zwischen den Strategien *MLFB*[7] und *SPN*[8] gewählt werden. In beiden Verfahren wird der Prozeß ausgeführt, der die kürzeste Laufzeit bzw. Ausführungsdauer aufzuweisen hat. Der Unterschied besteht in der Tatsache, wie die Laufzeit der Prozesse dem Planungswesen bekannt gemacht wird. Bei der Strategie *SPN* bekommt das Planungswesen die Prozeßlaufzeit von der Anwendung als Parameter mit übergeben, wohingegen die Methode *MLFB* die Zeiten der Prozesse protokolliert und eine Mittelwertbildung durchführt. Anhand dieser wird der momentan kürzeste Prozeß bestimmt und aktiviert. Der Prozeß gibt den Prozessor später infolge von Rechenzeitverbrauch ab, weil sein Mittelwert größere Werte annimmt als die anderer Prozesse. Die Abbildung 3.2 zeigt weiterhin ein optionales Merkmal *observation*[9] an dem Strategieknoten *SPN*. Dies ermöglicht bei Auswahl ein Überwachen der durch die

[2] arrival time(engl.) = Ankunftszeit
[3] rate(engl.) = Rate, Frequenz
[4] cpu time(engl.) = Prozessorzeit
[5] time limit(engl.) = Frist
[6] rate-monotonic(engl.) = monotone, gleichbleibende Frequenz
[7] MLFB = Multi Level Feedback = Mehrebenen Rückkopplung
[8] SPN = Shortest Process Next = kürzester Prozeß als nächstes
[9] observation(engl) = Beobachtung

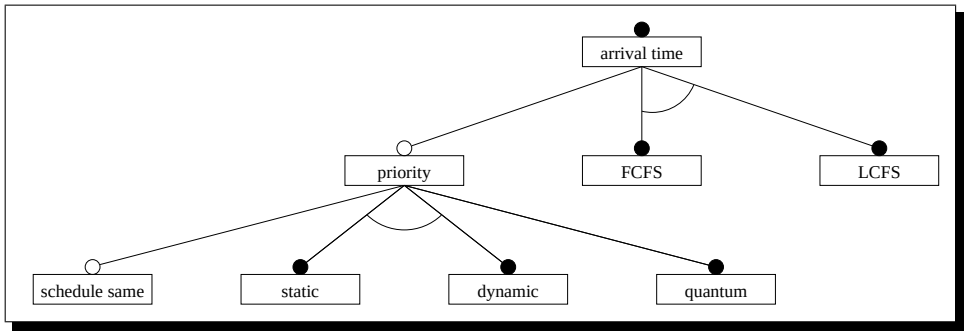


Abbildung 3.3: Merkmalsmodell der ankunftszeitbasierten Strategien

Anwendung angegebenen Prozeßlaufzeiten und ein Reagieren auf mögliche Überschreitungen bei Falschangabe. Die Reaktion kann von Fall zu Fall unterschiedlich ausfallen und ist durch die Anwendung geeignet zu implementieren. Aus diesem Grund ist dies auch nicht Bestandteil der formalen Merkmalsbeschreibung.

Planungsverfahren basierend auf Ankunftszeiten

Verfahren die als Planungskriterium die Ankunftszeit der Prozesse beim Planungswesen verwenden, sind sehr verbreitet und sie lassen sich hervorragend mit anderen Methoden kombinieren. So kann als weiteres Merkmal für die Planung etwa eine Priorität dienen, mit Hilfe dieser die Verarbeitungsreihenfolge beeinflußt wird.

Allgemein betrachtet gibt es zwei Formen der ankunftsbasierten Planungsverfahren. Unter ihnen kann die Anwendung wählen, um beispielsweise ein frühes (*FCFS*) bzw. ein spätes (*LCFS*) Ankommen zu favorisieren. Die Abbildung 3.3 zeigt die beiden alternativen Merkmale nebst der optionalen Eigenschaft Priorität (*priority*), welcher sich weitere Merkmale zuweisen lassen.

So sind etwa in der Darstellung eine optionale, zwei alternative und eine zwingende Eigenschaft der Priorität zugeordnet. Bisher war der entscheidende Faktor für einen möglichen Prozeßwechsel immer die Ankunftszeit beim Planungswesen. Wird jedoch zusätzlich die Priorität gewählt, hat diese einen gewichtigen Einfluß auf die Zuteilung der Ressource Prozessor. Einhergehend mit der Auswahl des optionalen Merkmals *schedule same*[10] ändert sich diese Steuerung dahingehend, daß ein Prozeßwechsel auch immer dann stattfindet, wenn zumindest die gleiche Prioritätsklasse vorliegt.

Als Alternativen stellen sich die beiden Merkmale *statisch* und *dynamisch* dar. Sie geben Auskunft über die Beschaffenheit der Prioritäten zur Laufzeit des Systems. Auf der einen Seite kann die Priorität statisch, also konstant über die gesamte Prozeßlaufzeit sein, andernfalls stellt sie sich veränderbar bzw. dynamisch dar. Dadurch entsteht aber oftmals ein Mehraufwand im Systembereich, weil sich

[10] schedule same(engl.) = plane gleiche

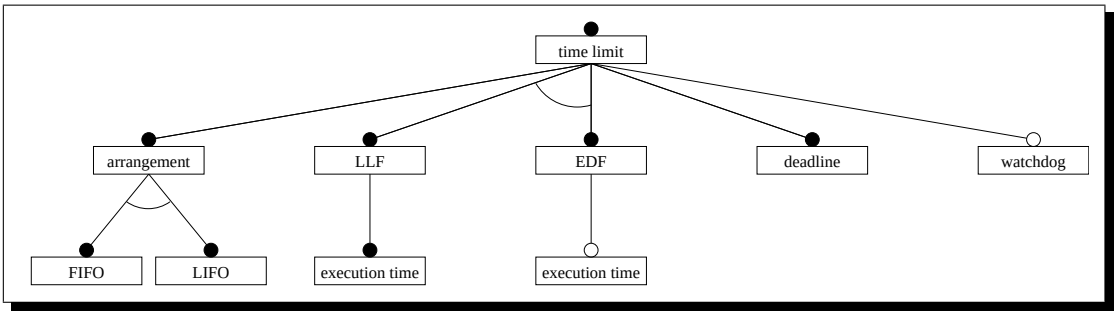


Abbildung 3.4: Merkmalsmodell der fristenbasierten Strategien

Ordnungsrelationen innerhalb von Warteschlangen verändern können. Als zwingende Eigenschaft gekennzeichnet, tritt das Merkmal *quantum* auf. Während der Konfigurationsphase ist dort die Anzahl der unterschiedlichen Prioritäten anzugeben, damit dem System der richtige Typ bereitgestellt werden kann.

Planungsverfahren basierend auf Fristen

Ein weiteres Gebiet in denen Planungsverfahren für Prozesse eingesetzt werden, ist der Bereich der Echtzeitsysteme. Prozesse müssen hier im allgemeinen Ergebnisse innerhalb bestimmter Fristen liefern, weil durch Zeitschrankenüberschreitungen diese ihre Gültigkeit bzw. Bedeutung verlieren. Schlimmer noch kann durch ein Verfehlen der Frist Schaden an Mensch und Material entstehen.

Das konfigurierbare Planungswesen verfügt über zwei verschiedene Verfahren speziell für den Echtzeitbetrieb. In der Abbildung 3.4 können die beiden alternativen Strategien *LLF*[11] und *EDF*[12] betrachtet werden. Beiden ist gemein, daß die *deadline*[13] immer eine zwingende Eigenschaft ist. Daher ist sie auf gleicher Ebene unterhalb des Konzeptknotens *time limit* aufgeführt. Zu sehen ist weiterhin, daß die Strategie *LLF* die Eigenschaft *execution time*[14] benötigt, wohingegen beim Verfahren *EDF* dieses Merkmal optional ist. Dies rührt aus den unterschiedlichen Herangehensweisen der Strategien zur Lösung ihrer Aufgabe her. Im Falle des Spielraumverfahrens wird die Berechnungszeit eines Prozesses benutzt, um mit Hilfe der Frist den verbleibenden Spielraum zu berechnen, wobei dem Prozeß mit dem kleinsten Spielraum der Prozessor zugeteilt wird. Bei der Behandlung der Prozesse nach *EDF* wird die Ausführungszeit an und für sich nicht benötig, weil allein die Frist über die Vergabe der Ressource CPU entscheidet. Jedoch kann bei Angabe der Berechnungszeit eine mögliche Fristüberschreitung von Prozes-

[11] LLF = Least Laxity First = kleinster Spielraum zuerst
[12] EDF = Earliest Deadline First = frühste Frist zuerst
[13] deadline(engl.) = letzter Termin, Frist
[14] execution time(engl.) = Ausführungszeit, Berechnungszeit

sen zur Planerstellungszeit errechnet werden und gegebenenfalls rechtzeitig eine Reaktion erfolgen.

Können Fristüberschreitung nicht zur Erzeugungszeit des Planes erkannt werden, weil die Ausführungszeiten nicht bekannt oder variabel sind, kann mit Hilfe der optionalen Eigenschaft *watchdog*[15] auf die Verfehlung einer Zeitschranke reagiert werden. Die Anwendung schreibt durch eine geeignete Implementierung der *Watchdog*funktion das Verhalten des Systems in solchen Fällen vor.

Als eine weitere zwingende Eigenschaft ist die Anordnung (*arrangement*) der Prozesse innerhalb der Warteschlange einzustellen, wodurch die Verarbeitungsreihenfolge zusätzlich beeinflußt wird. Folglich kann unter zwei Prinzipien *FIFO* und *LIFO* gewählt werden. Die Eigenschaft *arrangement* ist nur in diesem Ast des Merkmalsmodelles vorhanden, da bei allen anderen Planungsstrategien die Reihenfolge der Verarbeitung bereits festgelegt ist. Eine Änderung wäre natürlich möglich, jedoch hätte das Verfahren dann nicht das Verhalten, welches mit dem Namen assoziiert wird.

3.1.1.3 Kombinationsvielfalt

Bezugnehmend auf die vorangegangenen Abschnitte kann resümierend gesagt werden, daß in Merkmalsmodellen die allgemeinen sowie auch die variablen Bestandteile eines System grafisch zusammengefaßt werden. Bei der Betrachtung der Diagramme dem Anwender fällt die Kombinationsvielfalt im allgemeinen und die Konfigurationenmöglichkeiten im besonderen auf. So können Eigenschaften auf beliebige Art und Weise kombiniert werden, es sei denn, es liegen Restriktionen oder gesonderte Bedingungen vor.

Durch Verwendung der beschriebenen Merkmale lassen sich verschiedenen Arten von Systemen bauen, wobei auch durch einige Kombinationen eher zwecklose Zusammenstellungen entstehen. Dies heißt jedoch nicht, daß diese Systeme nicht lauffähig wären. Als Beispiel sei hier ein System angegeben, welches mit Hilfe der Echtzeitstrategie *EDF* arbeitet und eine Zeitscheibe (*timeslice*) verwendet. Die Benutzung der Zeitscheibe als Auslöser für einen Kontextwechsel ist hier völlig ohne Bedeutung, da die Strategie allein anhand der Frist den Wechsel initiiert.

Die meisten der möglichen Kombinationen ergeben dennoch nutzbare und sinnvolle Varianten. So können unter anderem Systeme konfiguriert werden, die zeitscheibengesteuert priorisierte Prozesse nach FCFS verarbeiten. Viele weitere Beispiele lassen sich finden und im Kapitel 3.2 wird auf einige näher eingegangen.

3.1.2 Funktionale Hierarchien

Nach den weitreichenden Betrachtungen über allgemeine und variable Systembestandteile gilt es nun die gewonnenen Erkenntnisse aus den Merkmalsmodellen

[15] watchdog(engl.) = Wachhund

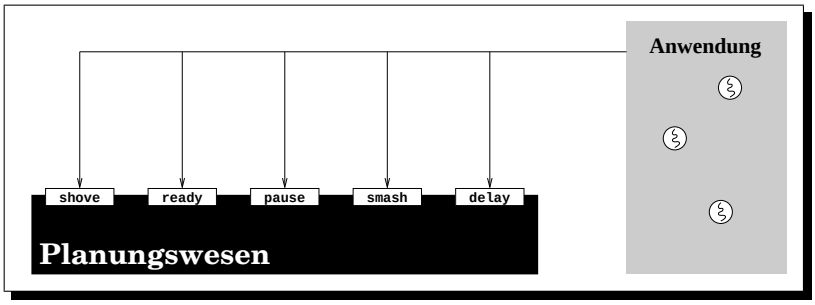


Abbildung 3.5: Blackboxdarstellung des Planungswesens

in eine funktionale Sicht zu überführen. Dadurch soll ein Übergang von der theoretischen auf die programmiertechnische Ebene erfolgen, damit die in Aussicht gestellten Eigenschaften für den Anwendungsprogrammierer greif- und verwendbar werden.

Zunächst soll erst einmal die Schnittstelle des Planungswesens aus der Perspektive der Applikation beleuchtet werden. Anschließend wird diese Blackboxdarstellung einer eingehenden Untersuchung unterzogen, wodurch die funktionalen Zusammenhänge der verschiedenen Ebenen zur Geltung kommen. Hierbei zeigt sich auch die Verwendung einer Prozeßebene, als Basis des Planungswesens. Darauf aufbauend wird die Verwaltung der wartenden Prozesse mit Hilfe eines Speichersystem gezeigt.

3.1.2.1 Bestimmung des Funktionumfanges

Den Funktionsumfang für eine allgemeine Schnittstelle zum Planungswesen zu entwerfen, stellt keinen trivialen Vorgang dar. Die Definition sollte auf eine minimale aber dennoch ausreichende Schnittstelle hinauslaufen, so daß ein breites Spektrum an Anwendungen ohne Zusatzmethoden damit auskommt. Das Planungswesen soll sich als eine Blackbox für die Applikation darstellen, wobei nur der Experte den tieferen Einblick in die Materie hat. Der Anwendungsprogrammierer kennt die Schnittstelle und das Verhalten des Planungswesen, aber was im Inneren an Funktionalität vorhanden ist, braucht ihn nicht zu beschäftigen. In der Abbildung 3.5 kann das Rechnersystem mit der Blackbox Planungswesen betrachtet werden. Der dargestellte Funktionsumfang beläuft sich auf fünf Methoden, die jeweils unterschiedliche Aufgaben wahrnehmen. Die Prozesse der Anwendung können das Planungswesen über die verschiedenen Methoden betreten, um sich bereit zu setzen, zu blockieren, zu beenden, zu pausieren oder um sich abhängig von Bedingungen das Vorrecht auf den Prozessor zu sichern. Nachfolgend ist eine ausführlichere Beschreibung der aufgeführten Funktionen zu finden:

`ready() :` benutzen Prozesse, um sich beim Planungswesen als bereit anzumelden, damit sie in der Zukunft den Prozessor zugeteilt bekommen.

`pause() :` rufen Prozesse auf, um anderen ebenfalls rechenbereiten Prozessen die Möglichkeit der Abarbeitung eigener Aufgaben zu geben.

`delay() :` kann der Prozeß aufrufen, beispielsweise wenn er auf ein Ereignis warten möchte, oder er seine Aufgabe beendet hat, aber für eine erneute Aktivierung bereit stehen möchte.

`smash() :` wird vom Prozeß verwendet, um sich entgültig aus dem System zurück zu ziehen, wodurch er für keine erneute Aktivierung zur Verfügung steht.

`shove() :` ist zum unmittelbaren Starten eines Prozesse ohne den Umweg über die Warteschlange des Planungswesens. Abhängig von der Planungsstrategie und den dabei vorhandenen Bedingungen wird der Prozeß gestartet oder gegebenenfalls doch auf der Warteliste platziert.

Die gezeigte Funktionalität kann als minimal betrachtet werden, weil sie für kooperative sowie präemptive Systeme ausreichend ist. In einem kooperativen System sind alle Methoden bis auf `shove()` vollständig ausreichend, da die Prozesse freiwillig die Kontrolle über die CPU abgeben und keine gesonderten Bedingungen zum Start von Prozessen vorliegen.

Hingegen nimmt die Methode `shove()` in einem präemptiven System eine besondere Stellung ein, da Prozesse ausdrücklich über diesen Funktionsaufruf gestartet werden sollten, da hier Prozesse defiziellen Bedingungen unterliegen können, denen Beachtung geschenkt werden muß. Das Bereitsetzen durch `ready()` umgeht diesen Mechanismus, weshalb in System mit speziellen Anforderungen (z.B. Prioritäten) jene Methode nicht direkt Verwendung finden sollte.

Durch Konfiguration können die Verhaltensweisen der einzelnen Funktionen beeinflußt werden, wodurch sich eine große Anzahl an unterschiedlichsten Strategien und Vorgehensweisen erstellen lassen.

3.1.2.2 Voraussetzungen auf Prozeßebene

Um die Funktionalität des Planungswesens umzusetzen, benötigt dieses als Voraussetzung eine Prozeßebene. Allgemein muß nur eine einzige Methode bereit gestellt werden, die das Umschalten von einem Prozeßkontext zu einem anderen bewerkstelligt. Daher ist es möglich das Planungswesen auf eine große Anzahl an Prozeßbibliotheken aufzusetzen. Es ist gegebenenfalls eine Adapterklasse zu implementieren, die eine Umsetzung von Funktionsnamen vornimmt.

In der vorliegenden Arbeit wurde der **T**hread **A**bstraction **L**ayer[16] (TAL [SP01]) aus der Vorlesungsreihe Betriebssystementwurf verwendet. Die Threadebene stellt dabei als Schnittstelle die Methoden `latch()`[17] und `yield()`[18] für

[16] **T**hread **A**bstraction **L**ayer(engl.) = Faden Abstraktions Schicht
[17] latch(engl.) = schließen
[18] yield(engl.) = nachgeben, weichen

Prozeßwechsel bereit. Die Funktionen unterscheiden sich in soweit, als daß bei der Methode `yield()` der Prozessorstatus des abgebenden Prozesses beim Kontextwechsel gesichert wird, wo hingegen `latch()` direkt den Registersatz des zu aktivierenden Fadens lädt. Hierdurch kann der abgebende Faden nicht erneut aktiviert werden, was beim Beenden von Prozessen aber unerheblich ist.

Wie bereits angemerkt, reicht jedoch eine Funktion für die Durchführung des Prozeßwechsels aus. Es kann beim Beenden von Prozessen durchaus eine Sicherung erfolgen, selbst wenn diese unnötig ist. Daher können auch Prozeßbibliotheken verwendet werden, die keine eigene Beendigungsmethode besitzen, indem die beiden Methoden `latch()` und `yield()` auf eine Prozeßwechselfunktion abgebildet werden.

3.1.2.3 Planungswesen

Ausgehend von der Prozeßebene soll darauf aufbauend das Planungswesen entstehen. Die Funktionalität nimmt sukzessive durch kleine Erweiterungen von Ebene zu Ebene zu. Den Grundbaustein und somit die Ebene null stellen die schon beschriebenen Methoden `latch()` und `yield()` dar. Dadurch ist das Planungswesen in der Lage Prozeßwechsel und Prozeßbeendigungen durchzuführen, weshalb die Ebene auch als Kontextwechselebene bezeichnet wird. Hierauf setzt die Ebene eins auf, welche Funktionen zum Vermerken und Abfragen des aktiven Prozesses bereit stellt. Als Methodenname wurde in beiden Fällen `label()`[19] gewählt, wobei einmal die Funktion mit Parameter existiert, um den Wert zu setzen und einmal ohne Parameter, um den gesetzten Wert auszulesen.

In der Darstellung 3.6 können unter anderem die beschriebenen Funktionen betrachtet werden. Es fällt auf, daß einige Funktionskästen mit gestrichelten Umrandungen versehen sind und wiederum andere einen grauen Hintergrund aufweisen, sowie Kombinationen aus beiden Varianten existieren. Die Erklärung hierfür liegt darin, daß die grau hinterlegten Funktionen die bereits beschriebene Schnittstelle des Planungswesens darstellen. Die gestrichelt umrandeten Funktionen können konfiguriert werden, liegen ergo zu meist in mehreren Implementierungen vor. Alle Funktionen ohne diese Merkmale sind nicht veränderbar ausgelegt.

Aus der Skizze entnehmbar ist also eine Konfigurierbarkeit der Methode `label()`. Einmal kann ein Vermerken oder Auslesen des aktiven Prozesses mit Hilfe eines Pointers[20] geschehen, aber auch durch eine weitere Methode, dem Ausnutzen von ausgerichteten Stacks[21]. Hierzu sei auf [Sch00] verwiesen.

Weiterführend kann in der Abbildung die nächste Ebene betrachtet werden. Unter Ausnutzung der Funktionen der tiefer gelegenen Schichten erbringen die

[19] label(engl.) = kennzeichnen, Marke
[20] Pointer(engl.) = Zeiger,Verweis
[21] Stack(engl.) = Stapel, Kellerspeicher

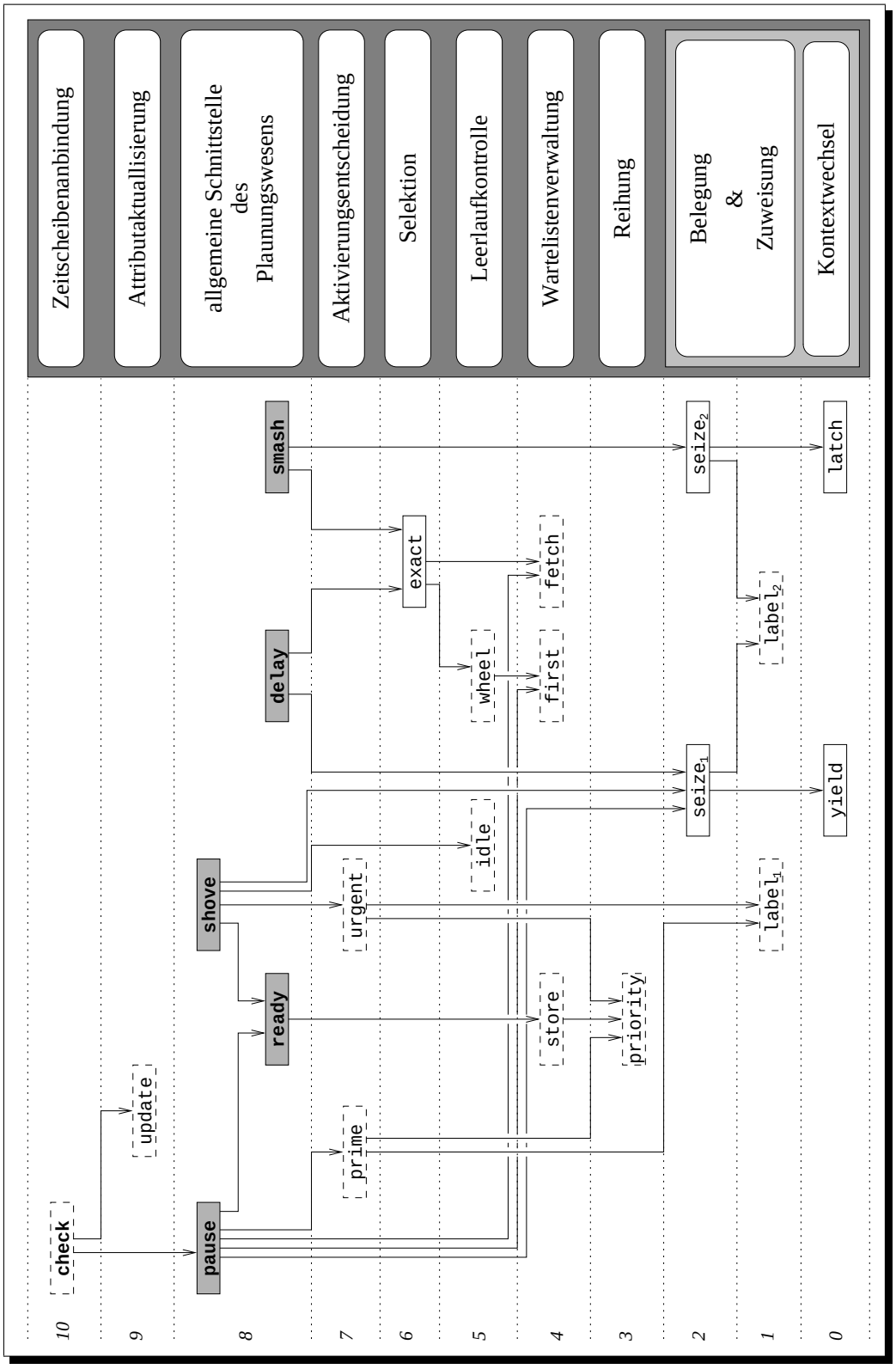


Abbildung 3.6: Funktionale Einheiten und Hierarchie des Planungswesens

Methoden `seize`$_1$`()`[22] und `seize`$_2$`()` ihre Aufgabe. Beide vermerken zunächst mit Hilfe des Aufrufes von `label`$_2$`()` den nun aktiv zu setzenden Prozeß.

`seize`$_1$`()` erbringt danach das Umschalten durch `yield()`, wobei der Prozessorstatus des abgebenden Prozesses gesichert und der des neuen installiert wird. Diese Funktion wird von den Methoden `pause()`, `shove()` und `delay()` der allgemeinen Schnittstelle verwendet.

`seize`$_2$`()` wird verwendet, um Prozesse umzuschalten ohne den alten Prozessorzustand zu sichern. Diese Funktion wird nur von der Beendigungsmethode `smash()` (siehe Abbildung) benutzt.

Die unteren drei Schichten bilden den Grundstein für das Planungswesen und seine jeglichen Ausprägungen, weil hier das allgemeine Umschalten von Prozessen sowie das Herausfinden des momentan aktiven Prozesses erbracht wird. Die hier anzutreffenden Methoden sind grundsätzlich Bestandteil des Endsystems. Bei den Funktionen auf höher gelegenen Ebenen ist dies nicht immer der Fall. Dort ist es eine Frage der Konfigurationsentscheidung, ob und in welcher Ausführung Methoden zum Einsatz gelangen.

So ist etwa die Methode `priority()`[23] der Ebene drei nicht per Definition im System enthalten. Wird beispielsweise ein System ohne Prioritäten benötigt, entfällt diese Funktion oder ist als leere Implementierung vorhanden. Deshalb ist sie durch eine gestrichelte Umrandung als konfigurierbar gekennzeichnet. In Systemen mit Prioritäten oder Parametern, aus denen sich prioritätsähnliche Werte errechnen lassen, ist der Rückgabewert der Methode `priority()` eben genau diese Größe, mit deren Hilfe Entscheidungen über Prozeßaktivierungen oder Sortierreihenfolgen getroffen werden können. So kann die Methode einfach eine Zahl zurückgeben, wie es bei statischen Prioritäten der Fall ist oder das Ergebnis einer Berechnung liefern, welches bei Verfahren mit dynamischen Wertigkeiten einer ständigen Veränderung unterliegt.

Die vierte Schicht der `Wartelistenverwaltung` stellt drei konfigurierbare Funktionen zur Verfügung. Die Verwaltung der rechenbereiten Prozesse ist naturgemäß stark abhängig von der Strategie des gewählten Planungsverfahrens. Deshalb ist für eine optimale Unterstützung der Verfahren für eine geeignete Handhabung der ablauffähigen Prozesse zu sorgen. Die Funktionen erfüllen dabei unterschiedliche Aufgaben. Grob kann folgendes Verhalten der Funktionen vorausgesetzt werden:

`store()`[24] ordnet die rechenbereiten Prozesse in einer dafür vorgesehenen Datenstruktur an. Die Datenstruktur kann dabei abhängig von der Sortierstrategie des Verfahrens eine sortierte Liste, eine Warteschlange, ein Stapel oder ein Feld sein. Die Skizze zeigt eine Benutzung der Methode `priority()` an, wenn diese die Ordnungsreihenfolge beeinflussen soll.

[22] seize(engl.) = belegen
[23] priority(engl.) = Priorität

`fetch()`[25] entfernt und liefert das erste Element der Datenstruktur.

`first()`[26] liefert das erste Element (Prozeß) in der Datenstruktur ohne es aus dieser zu entfernen. Die Funktion wird verwendet, um Vergleiche durchzuführen und um vor einem Entfernen von Elementen sicherzustellen, daß überhaupt ein Element existiert.

Auf dieser Funktionalität aufbauend, werden die weiteren Schichten in kleinen Schritten aufgebaut. So wird in der Ebene der Leerlaufverwaltung die Methode `first()` von der Methode `wheel()`[27] verwendet. Dabei erbringen die Funktionen der Ebene fünf die Leerlaufeigenschaft des Systems, sobald diese nötig ist. Daß heißt, wenn kein ablauffähiger Prozeß vorhanden ist, wird in einer Endlosschleife ständig die Warteliste abgefragt, ob rechenbereite Prozesse aufgetaucht sind. Die Methode `idle()`[28] liefert dabei als Resultat den Status des Systems, sei es momentan im Leerlauf oder nicht. Der Skizze ist wiederum zu entnehmen, daß es sich um eine konfigurierbare Einheit des Planungswesens handelt. Für den Fall, daß immer mindestens ein rechenbereiter Prozeß zur Verfügung steht oder ein Leerlaufprozeß vorhanden ist, kann die Leerlaufeigenschaft auch aus dem System entfernt werden, andernfalls muß sie existieren.

Die Ebene sechs stellt im Grunde genommen keine neue Funktionalität bereit, denn sie verbindet bereits vorhandene Methoden so, daß eine leichte Nutzung durch übergeordnete Schichten ermöglicht wird. Die als `Selektionsebene` bezeichnete Schicht sichert durch den Aufruf der Methode `whell()` ein Vorhanden sein eines lauffähigen Prozesses innerhalb der Warteliste ab und erbringt im folgenden durch den Aufruf von `fetch()` das Entfernen aus der Datenstruktur. Als Rückgabewert liefert die Methode `exact()`[29] dann diesen Prozeß entweder an die Funktion `delay()` oder `smash()` der allgemeinen Schnittstelle des Planungswesens. Die Funktion `exact()` braucht nicht veränderbar ausgelegt werden, da sie in sämtlichen möglichen Szenarien die gleiche Funktionalität und Verhaltensweise aufweist.

Der folgenden siebenten Ebene kommt eine absolut strategische Bedeutung zu, da hier in der Aktivierungsentscheidungsschicht über den Start von Prozessen befunden wird. Zwei konfigurierbare Funktionen `prime()`[30] und `urgent()`[31] stehen den aufbauenden Schichten zur Verfügung. Die Methoden sind abhängig von der Strategie der Planungsverfahrens unterschiedlich ausgelegt. Sie verwenden jeweils die Methode `priority()` (Ebene 3) und $\texttt{label}_1\texttt{()}$ (Ebene 1) um die Aufgabe der Entscheidungsfindung zu bewerkstelligen. Dabei wird mit Hilfe der Methode `label()` vom aktiven Prozeß die `priority()` Funktion aufgerufen und

[27] wheel(engl.) = Drehung, Rad
[28] idle(engl.) = untätig, Leerlauf
[29] exact(engl.) = bestimmt, genau
[30] prime(engl.) = wichtigste, primär
[31] urgent(engl.) = dringend, vorrangig

deren Rückgabewert mit dem Rückgabewert der `priority()` Methode des aufkommenden Prozesses verglichen. Das Ergebnis gibt über die Dringlichkeit des einen oder anderen Prozesses Auskunft. Den oberen Schichten obliegt nun die weitere Vorgehensweise in Bezug auf das Initiieren des Umschaltvorganges.

Der Kontextwechsel wird durch die beiden Funktionen `shove()` und `pause()` vorangebracht, indem sie die Funktion $\texttt{seize}_1\texttt{()}$ aufrufen, welche das Belegen und Zuteilen der Ressource Prozessor vornimmt. Vorher wird jedoch der verdrängte Prozeß in die Datenstruktur der rechenbereiten Prozesse aufgenommen, was über den Mechanismus des Bereitsetzens (`ready()`) erreicht wird. Als allgemeine Schnittstelle sind die anderen Funktionen dieser Ebene schon bekannt, weshalb sie hier nicht erneut beschrieben werden sollen.

Die Ebenen neun und zehn arbeiten sehr eng miteinander zusammen. Dies läßt sich der Abbildung ebenfalls entnehmen, da die Methode `update()`[32] einzig und allein von der Methode `check()`[33] verwendet wird. Die Funktion `check()` ist als Unterbrechungsanbindung für den `Zeitscheibenmechanismus` angedacht. Beiden Funktionen ist eine Anpaßbarkeit an unterschiedliche Strategien gemein. Das liegt zum einen in der Tatsache begründet, daß nicht alle Planungsstrategien eine Zeitscheibe verwenden und zum anderen an den unterschiedlichen Parametern und deren Anpassungen an die verschiedenen Verfahren. In einigen Planungsverfahren muß nach dem Ablauf einer Zeitscheibe eine Neuberechnung der Planungsparameter erfolgen, in andern wird einfach zum nächsten Prozeß umgeschaltet. Die Methode `update()` kapselt deshalb den Mechanismus der eventuellen Attributanpassung und einer möglichen damit verbundenen Neuausrichtung der rechenbereiten Prozesse.

Werden einmal alle Ebenen als gesamtes System betrachtet, kann eine Vielzahl an Konfigurationspunkten entdeckt werden, wodurch unter Ausnutzung verschiedener Implementierungen für die anpaßbaren Methoden ein große Anzahl an Endsystemen erstellbar ist. Gemäß der Merkmalsmodelle ist ein maßschneiderbares System zu entwerfen. Ein Schritt in diese Richtung stellt der allgemeine Entwurf der funktionalen Hierarchie des Planungswesens dar. Jedoch ist noch nicht die gesamte Palette der Konfigurationsmöglichkeiten der Merkmalsdiagramme vorhanden. Deshalb wird sich ein weiterer Teil einer funktionalen Anreicherung im Bereich der Prozeßerweiterung niederschlagen. Im folgenden Abschnitt wird dieser Punkt einer genaueren Betrachtung unterzogen.

3.1.2.4 Prozeßerweiterungen

Der Bereich der Prozeßerweiterungen beschäftigt sich mit dem Aufbau einer konfigurierbaren Prozeßabstraktion, welche sich nach dem Vorbild des Planungswesens ebenfalls maßschneidern lassen soll. Die Basis bildet eine Prozeßbibliothek, die Funktionen insbesondere für die Erzeugung, den Kontextwechsel und den

[32] update(engl.) = aktualisieren
[33] check(engl.) = prüfen, kontrollieren

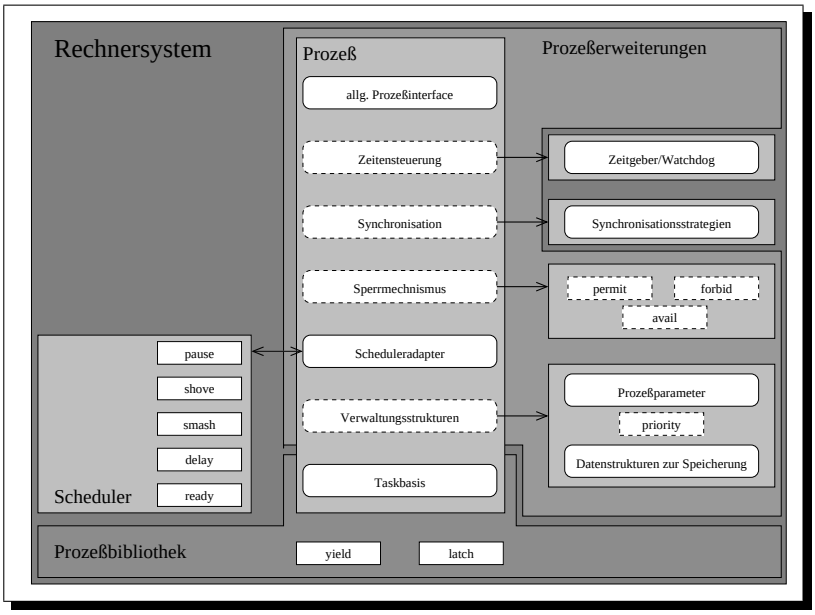


Abbildung 3.7: Funktionale Einheiten des Prozesses

Abschluß von Prozessen bereit stellt. Die Generierung von neuen Prozessen soll jedoch hier nicht betrachtet werden. Es wird vorausgesetzt, daß eine Erzeugung von Prozessen möglich ist. Als Grundbaustein der Prozeßabstraktion findet die schon erwähte TAL-Bibliothek ihre Anwendung, welche die Methoden `yield()` (Prozeßwechsel) und `latch()` (Prozeßwechsel ohne Zustandssicherung ⇒ Prozeßabschluß) anbietet. Beide Funktionen bilden das Fundament für den weiteren Ausbau der Funktionalität, aber sie allein sind natürlich nicht ausreichend, um eine Verwaltung durch ein Planungswesen zu ermöglichen. Eigenschaften der Prozesse wie etwa Sperrmechanismen bzw. Blockadenkontrollen, Watchdogüberwachung und Prioritäten, die in den Merkmalsdiagrammen beschrieben wurden, sind noch kein Bestandteil des Prozeßmodells. Schritt für Schritt sind diese Eigenschaften dem Prozeßbild hinzuzufügen, aber immer mit der Maßgabe anpaßbare und austauschbare Einheiten zu schaffen.

Im ersten Schritt wird die Prozeßstruktur um Daten für die Verwaltung durch das Planungswesen erweitert. Diese Daten können sehr unterschiedlich aufgebaut und in ihrer Anzahl variieren, da es von dem Planungsverfahren abhängt, welche Prozeßparameter benötigt werden. Als Beispiel kann eine Priorität und/oder Verkettungseigenschaft hinzukommen. Die Möglichkeit einer Verkettung wird im Falle der `Wartelistenverwaltung` des Planungswesen benötig. In der Abbildung 3.7 ist der behandelte Aspekt in dem Bereich `Verwaltungsstrukturen` zusammen gefaßt. Es ist zu erkennen, daß aus den Datenstrukturen des Prozesses die bereits bekannte Funktion `priority()` stammt, welche beim Scheduling verwendet wird, um über die Aktivierung von Prozessen zu befinden. Die gestrichelte Umrandung von `Verwaltungsstrukturen` sowie der Methode `priority()` verdeutlicht noch

einmal die Konfigurierbarkeit, wobei der Anwendungsfall die Art der Prozeßparameter (Prioritäten, Fristen, ...) vorschreibt.

Darauf aufbauend ist die nächste Stufe der Abstraktion die Verbindung der Prozesse mit dem Scheduler (`Scheduleradapter`), weil nun die Datensturkturen für das Planen bereitstehen. Durch die Bindung Prozeß⇔Planungswesen braucht der Anwendungsprogrammierer nicht direkt die Instanz des Planers kennen oder ansprechen, sondern er ruft einfach auf den Prozessen die Funktionen `pause()`, `shove()`, `smash()`, `delay()` und `ready()` auf. Intern erfolgt eine Umsetzung auf die Schedulerfunktionen mit dem Prozeß als Parameter. Der `Scheduleradapter` stellt im Grunde genommen keine wirklich neue Funktionalität bereit, lediglich zur besseren Handhabbarkeit des Prozeßmodelles ist diese Schicht eingebaut.

In der folgenden Ebene sind die `Sperrmechanismen` der Prozesse angeordnet, welche bereits in dem Merkmalsdiagramm in Abschnitt 3.1.1.2 beschrieben wurden. Als Systemeigenschaft wurden sie unter dem Begriff *blockade* eingeführt. Falls eine Sperrung des Planungswesen benötigt wird, kann sie auf unterschiedliche Weise (local vs. global) verwirklicht werden. Aus diesem Grund wurde wiederum die gestrichelte Darstellung der konfigurierbaren Einheit gewählt. Der Abbildung der Prozeßabstraktion kann entnommen werden, daß drei Funktionen zur Schnittstelle des Prozesses hinzukommen. Es handelt sich hierbei einerseits um die Sperrmethode `forbid()`[34] und andererseits um die Methode `permit()`[35], welche das Scheduling wieder freigibt. Weiterhin steht die Methode `avail()`[36] zur Verfügung, über die der Zustand der Sperre abgefragt wird. Ebenfalls als anpaßbare Funktionen ausgelegt, kann der Anwender entscheiden, welche Art der `Blockadensteuerung` er einsetzen möchte. Zur Auswahl stehen die im Merkmalsmodell beschriebenen Möglichkeiten der globalen oder lokalen Sperrung.

Aufsetzend auf die unteren Ebenen wird die Prozeßabstraktion um eine `Synchronisationseigenschaft` erweitert. Ebenfalls anpaßbar kann die Schicht fehlen, falls es keiner Synchronisation bedarf, andernfalls sind mehrere verschiedene Mechanismen zur Synchronisierung denkbar. Da der Aspekt der Synchronisationsstrategien nicht zum eigentlichen Prozeßbild gehört, ist er außerhalb der Abstraktion abgebildet. Allgemein müssen Prozesse beim Zugriff auf ein gemeinsam genutztes Betriebsmittel synchronisiert werden, weil oftmals kritische Datenstrukturen vorliegen. Im Falle des Planungswesens handelt es sich um gerade so eine Ressource, mit der Warteliste als kritischer Datenbestand. Da es nicht Gegenstand der Arbeit ist, Synchronisationsmechanismen zu entwickeln, wird auf [HH89] verwiesen.

Auf der Ebene der Synchronisation setzt die Schicht der `Zeitensteuerung` auf. Ebenso wie die vorangegangenen Schichten ist auch diese Ebene konfigurierbar ausgelegt. Es stehen drei Möglichkeiten der Nutzung bereit. Als erstes kann

[34] forbid(engl.) = untersagen, verbieten
[35] permit(engl.) = erlauben, zulassen
[36] avail(engl.) = nutzen, helfen

die Ebene vollständig fehlen, falls kein Zeitverhalten von Prozessen gefragt ist. Zweitens kann sie genutzt werden, um einen fairen `Zeitscheibenmechanismus` zu erzwingen, und drittens kann sie eingesetzt werden, um ein Überwachen von zeitkritischen Prozessen zu erbringen. In der Abbildung ist die Ansteuerung und Verwendungen eines Zeitgebers oder Watchdogs zu sehen, welcher sich außerhalb der Prozeßerweiterungen im allgemeinen System befindet.

Nachdem nun die Prozeßabstraktion aus vielen konfigurierbaren Schichten besteht und diese aber auch gegebenenfalls fehlen können, bildet ein `allgemeines Prozeßinterface` die abschließende Ebene. Hier sind noch einmal alle Methoden des Ebenenmodells zusammengefaßt. Durch diese Maßnahme können Anwendungen völlig unabhängig vom Prozeßmodell bzw. dessen Ausprägung programmiert werden. So braucht bei einer Änderung der Konfiguration nur eine Neuübersetzung erfolgen und nicht eine Anpassung der Anwendung. Im Bereich der Implementierung spielt diese Schicht zusätzlich noch eine besondere Rolle für die minimale Maschinencodeerzeugung im Bezug auf den Charakter des Inliningprozesses.

3.1.2.5 Prozeßparameter

In der Abbildung 3.7 sind neben der konfigurierbaren Prozeßabstraktion auch die Prozeßparameter und die `Verwaltungsstrukturen` abgebildet. Dort wird nur eine allgemeine Sicht auf diesen wichtigen Punkt der Verwaltung eingenommen. In diesem Abschnitt soll auf die Parameter der Prozesse etwas spezieller eingegangen werden. Zu den Strukturen zur Speicherung der Prozesse ist nur zu sagen, daß eine Verkettungseigenschaft zur Verfügung steht, sobald diese benötigt wird. Die funktionalen Zusammenhänge werden im Abschnitt 3.1.2.6 eingehender behandelt.

Als wichtige Prozeßparameter gelten die im Abschnitt 2.1.3 beschriebenen Größen, wobei sich aus ihnen ein Wert ermitteln läßt, welcher als Priorität gewertet wird. Da die verschiedenen Verfahren zumeist unterschiedliche Größen verwenden, muß eine anpaßbare Parameterabstraktion geschaffen werden. Über die konfigurierbare Funktion `priority()` ist das Planungswesen in der Lage die Prozeßprioritäten abzufragen und mit Hilfe des gelieferten Wertes Entscheidungen über Prozeßwechsel zu fällen.

Die Darstellung 3.8 gibt einen Einblick in die Methode `priority()`, wodurch die Zusammensetzung der verschiedenen Prioritäten ersichtlich wird. Zu sehen ist, daß aufbauend auf einer allgemeinen Priorität (`general valence`) die weiteren Wertigkeiten Schicht um Schicht aufgebaut werden. Durch das Hinzufügen von zusätzlichen Informationen und Parametern entstehen sukzessive die benötigten Prioritäten. Die `general valence` wird in Systemen verwendet, in denen die Prioritäten statischer Natur sind, was einen konstanten Wert über die gesamte Laufzeit eines Prozesses bedeutet.

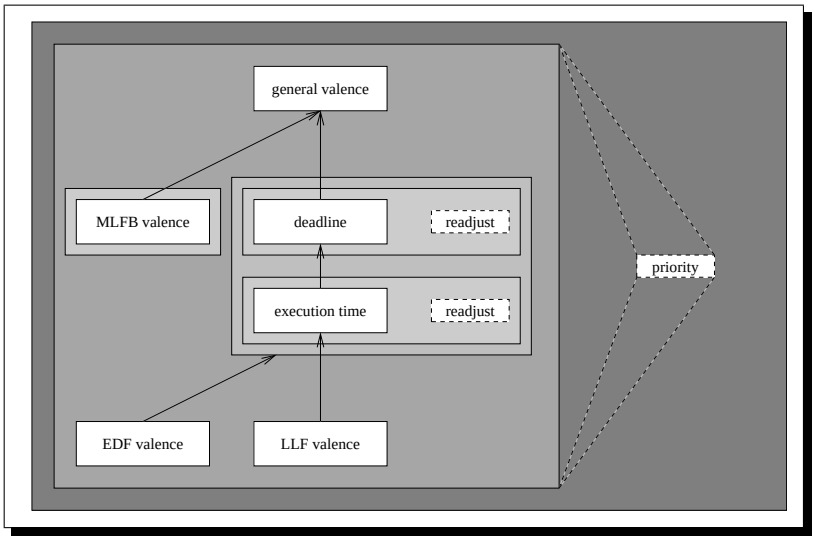


Abbildung 3.8: Funktioneller Aufbau der Prozeßparameter

Für dynamische Verfahren mit wechselnden Prioritätswerten müssen neue Funktionalitäten und Informationen hinzugefügt werden. So ist etwa im Falle der `deadline` nur Funktionalität hinzugebracht, jedoch die Speicherung des eigentlichen Wertes erfolgt in `general valence`. Bei der zusätzlichen Funktionalität (`readjust()`[37]) handelt es sich um die Veränderung des Prioritätswertes im Laufe des Arbeitszyklusses des Prozesses. Denn die verbleibende Arbeitszeit eines Prozesses wird immer kleiner, je weiter er sich seiner eigenen Frist nähert. Somit ist die Wertigkeit eines Prozesses innerhalb dieser Klasse ständig den Gegebenheiten anzupassen.

Ebenso verhält es sich mit der `execution time`, welche mit zunehmender Laufzeit immer geringer werden muß. Hier ist aber ein zusätzlicher Wert zu speichern, der die Zeit aufnimmt. Ansonsten gilt das soeben bei `deadline` Gesagte.

Für das Verfahren *EDF* ist die `deadline` essentiell, da sie allein ausschlaggebend für Prozeßwechselentscheidungen und Fristverfehlungen ist. Wenn die `execution time` bekannt ist, kann sie genutzt werden, um Konflikte und Fristverfehlungen schon im Vorfeld zu entdecken. Es ist im Bild angedeutet, daß die `EDF valence` konfiguriert werden kann, abhängig von der Kenntnis über die `execution time`. Bei dem Planungsalgorithmus nach *LLF* wird hingegen die `execution time` unbedingt benötigt, um die verbleibenden Spielräume der Prozesse zu errechnen. Hier ist ebenfalls eine Vorabüberprüfung auf Erreichbarkeit des Planungsziels möglich.

In der Darstellung kann auch eine extra Prioritätsbeschreibung für das Verfahren *MLFB* betrachtet werden. Dieses nutzt die `general valence` um die Prio-

[37] readjust(engl.) = nachregeln, nachstellen

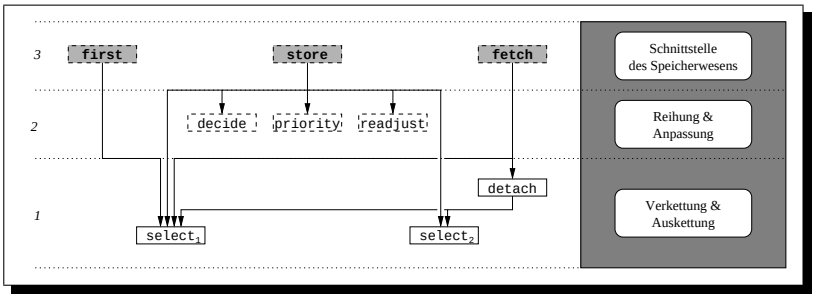


Abbildung 3.9: Funktionale Zusammenhänge im Speicherwesen

ritätswerte zu speichern, implementiert jedoch ein eigenes Verhalten, um die Werte den Ansprüchen entsprechend anzupassen.

3.1.2.6 Wartelistenverwaltung und Speicherwesen

Nachdem die funktionalen Zusammenhänge der Prozeßparameter untersucht wurden, soll in diesem Abschnitt die Verwaltung der rechenbereiten Prozesse einer Betrachtung unterzogen werden. Die Prozeßgrößen spielen hier ebenfalls eine entscheidene Rolle, da sie einen Einfluß auf die Position innerhalb der Speicherstrukturen der wartenden Prozesse haben. Unter der Berücksichtigung der unterschiedlichen Planungsverfahren muß das Speicherwesen gleichermaßen wie die anderen Systemteile konfigurierbar aufgebaut sein. Daher wird das Teilsystem wieder als Familie entworfen, wodurch eine feingranulare Struktur entsteht, die hochgradig anpaßbar ist.

Im Bild 3.9 kann der grundsätzliche Aufbau des Speicherwesens im Schichtenmodell betrachtet werden. Als wesentliche Funktionen erbringen `select()`$_1$[38], `select()`$_2$ sowie `detach()`[39] das Einfügen bzw. das Entfernen von Elementen, wodurch eine Verwaltung in Form von Warteschlangen, Listen und Stacks ermöglicht wird. Die Funktionen weisen dabei folgendes charakteristisches Verhalten auf:

`select()`$_1$ liefert als Rückgabewert den Nachfolger eines Elementes.

`select()`$_2$ definiert einen neuen Nachfolger eines Elementes, wobei dieser beim Aufruf der Funktion als Parameter mit übergeben werden muß.

`detach()` erbringt das Entfernen eines Elementes unter der zu Hilfenahme der beiden `select()` Funktionen.

Die beschriebene Funktionalität stellt die minimale Voraussetzung für die Verkettung der rechenbereiten Prozesse dar.

[38] select(engl.) = auslesen, auswählen, selektieren
[39] detach(engl.) = abtrennen, lösen

In der folgenden zweiten Ebene werden die betrachteten Funktionen jedoch noch nicht verwendet. In der Schicht `Reihung & Anpassung` sind die Funktionen `priority()` und `readjust()` aus dem Block der Prozeßparameter wiederzufinden. Sie bilden zusammen mit der Methode `decide()`[40] die Grundlage für die Entscheidungsfindung über die Sortierreihenfolge sowie einer möglichen Anpassung von Prozeßprioritäten. Durch die gestrichelte Umrandung der Funktionen wird gezeigt, daß alle drei Funktionen anpaßbar sind. Im Falle von `decide()` kann zwischen einer absteigenden oder aufsteigenden Sortierreihenfolge gewählt werden. Die zwei anderen Funktionen wurden in vorangegangenen Abschnitt schon beleuchtet, wo die maßgerechte Adaptierbarkeit bereits gezeigt wurde.

Die Vielfalt der möglichen `Verwaltungsstrukturen` (Warteschlage, sortierte Liste, Stack, ...) verlangt, daß die Funktionen der allgemeinen `Schnittstelle des Speichersystems` flexibel auf spezielle Bedürfnisse hin zugeschnitten werden können. Dies wird noch einmal durch die gestrichelte Umrandung der Funktionen `store()`, `fetch()` und `first()` ausgedrückt. So ist die Methode `store()` beispielsweise in einer Warteschlange anders verwirklicht, als etwa in einer sortierten Liste und wiederum unterschiedlich im Falle eines Stapelspeichers. Hingegen ist die Funktion `first()` für alle listenorientierten Verfahren gleich implementiert. Dies zeigt die gute Umsetzbarkeit des Systems durch den Familiengedanken und der sich daraus ergebenen Wiederverwendbarkeit von Softwarebausteinen.

3.1.3 Klassenentwurf und Implementierung

Ein weiterer Schritt in der Verwirklichung des Systems in tatsächlich verwendbare Software stellt der Entwurf einer objektorientierten Klassenhierarchie dar. Wie bereits im Abschnitt 1.2 angedeutet, läßt sich der Familiengedanke sehr gut durch eine objektorientierte Implementierung umsetzen. In den nun folgenden Abschnitten sollen zuerst einmal die unterschiedlichen Möglichkeiten der Umsetzung der Flexibilität mit Hilfe von Sprachelementen untersucht werden. Darauf aufbauend werden unter der Nutzung der gewonnenen Erkenntnisse die verschiedenen Systembestandteile objektorientiert entworfen und die Hierarchie der Klassen eingehend betrachtet. Als letzter Schritt erfolgt ein Ausblick auf mögliche Erweiterungen.

3.1.3.1 Implementierungsmöglichkeiten

Zunächst werden unter Implementierungsmöglichkeiten nicht nur allgemeine Konstrukte einer objektorientierten Sprache verstanden. Eine weitere Möglichkeit Flexibilität zu gewinnen, besteht in der Nutzungen eines spezielles Werkzeuges. Dieses kann Programmkodetransformationen durchführen, um beispielsweise das Softwaresystem den anwendungsspezifischen Anforderungen anzupassen.

[40] decide(engl.) = befinden, entscheiden

Mit Hilfe eines solchen zusätzlichen Programms (z.B. PUMA[41]) können komplexe Klassenstrukturen vereinfacht und dadurch meist ein besseres Laufzeit- sowie Speicherverhalten erreicht werden. Dieser Ansatz wurde in der Arbeit nicht weiter verfolgt, es sei auf [Urb01] verwiesen. Mehrere Punkte spielten hierfür eine Rolle, wobei nachfolgend die wichtigsten aufgeführt sind:

- Werkzeug noch im experimentellen Stadium der Entwicklung
- zusätzliche Werkzeugabhängigkeit des Systems
- Werkzeug unterstützt noch keine *Templates*

Eine weitere Möglichkeit flexible Strukturen zu erreichen, stellen verschiedene Sprachkonstrukte dar. Zu nennen sind *virtuelle Funktionen*, *Typedefs*[42] und *Templates*[43]. Das Wort flexibel hat bei jedem Sprachkonstrukt eine unterschiedliche Bedeutung. Im Falle der *Typedefs* und *Templates* ist die Anpaßbarkeit zur Konfigurierungs- und Generierungszeit des Systems gegeben, hingegen kann bei *virtuellen Funktionen* sogar eine Spezialisierung während der Laufzeit des Systems erbracht werden. Diese Funktionalität ist aber nicht ohne einen gewissen Mehraufwand im Bezug auf Laufzeit und Speicherverbrauch zu erreichen. Da im Bereich der eingebetteten Systeme meist restriktive Anforderungen vorliegen und eine Änderung zur Laufzeit oft nicht nötig ist, sollte das Einbringen von *virtuellen Funktionen* sparsam und an nur wirklich benötigten Stellen erfolgen.

Der Mechanismus der *Typedefs* und *Templates* ist statischer Natur, weshalb im endgültigen System von diesen Strukturelementen keine Rückstände vorhanden sind. Jedoch besteht bei der Verwendung von *Templates* die Gefahr der Codedopplung und einem damit verbundenen höheren Speicherverbrauch. Allgemein dienen sie dem flexiblen Programmieren auf der Ebene der Sprache, da mit Hilfe von Schablonen typenunabhängige Algorithmen entworfen werden können.

In der Arbeit werden *Templates* und *Typedefs* verwendet. Der Einsatz von *virtuellen Funktionen* war bisher nicht nötig, da momentan noch sämtliche Funktionalität des Systems ohne zu Hilfenahme dieser Sprachmöglichkeit erbracht werden konnte. Wenn jedoch Erweiterungen des gesamten Komplexes angestrebt werden, kann es aber durchaus zu einer Anwendung *virtuellen Funktionen* kommen. Der Mechanismus der *Templates* findet im Bereich des Planungswesen für die verschiedenen Prozeßabstraktionen seine Verwendung, um die Basisfunktionen des Planers unabhängig vom Prozeßmodell immer konstant zu halten. Weiterhin sind zu meist Vererbungsbeziehungen mit *Templates* ausgedrückt, das heißt die Klasse erbt von einem *Template*-Typen, damit Basisklassen flexibel ausgetauscht werden können. Dadurch besteht die Möglichkeit unterschiedliche Funktionalität in die Klassen einzubringen, jedoch unter der Voraussetzung gleicher Schnittstellen.

[41] PUMA = `Pure Manipulator`
[42] Typedef = `type def`inition(engl.) = Typendefinition
[43] Template(engl.) = Schablone

Im Bereich der Konfiguration ist ein Zusammenspiel von *Typedefs* und *Templates* anzutreffen, wobei die *Typedefs* der Übersichtlichkeit des Konfigurationsvorganges dienen und die neu generierten Typen darstellen. Die *Templates* dienen – wie schon erwähnt – dem einfachen Austauschen der Basisklassen, wodurch sich verschiedene Ableitungsfolgen in der Klassenhierarchie erstellen lassen. Dadurch können auf simple Weise unterschiedliche Systeme gebaut werden, ohne den eigentlichen Quellkode direkt zu manipulieren, da der Compiler[44] diese Arbeit übernimmt. Ein weiterer Vorteil der Nutzung von *Templates* ist die Typenüberprüfung durch den Compiler zur Konstruktionszeit und der daraus resultierenden Sicherheit auf Typenverträglichkeit.

3.1.3.2 Planungswesen

Während des Entwurfes der Klassenbibliothek des Planungswesens wurden viele kleine Schritte zum nun vorliegenden System gemacht. Am Anfang wurden Vererbungsbeziehungen direkt formuliert, wodurch sich eine immense Anzahl an Klassen ergab, wobei nur ein Bruchteil der jetzigen Funktionalität gegeben war. Um einer weiteren Klassenexplosion vorzubeugen sowie aus Gründen der sinnvollen Namensfindung sind nun die Vererbungsbeziehungen in den Klassenhierarchien mit Hilfe von *Templates* ausgedrückt. Es entstand ein flexibles System, welches auch offen für Erweiterungen und Anpassungen ist.

Die Strukturierung des Systems ergab sich oftmals direkt aus den funktionalen Hierarchien. Zurückblickend auf die Abschnitte 3.1.2.3 kann in der Abbildung 3.6 bereits eine Einteilung in funktionale Einheiten betrachtet werden. Von den gemachten Erkenntnissen ausgehend, wurden die Einheiten zumeist sofort als Klassen übernommen. Nur in einzelnen Fällen sind Ebenen noch weitere aufgeteilt und in mehrere Klassen aufgespaltet, damit eine bessere Konfigurierung und Übersichtlichkeit erreicht wird. Ebenso ist auch ein Zusammenfassen von Ebenen möglich. Als Beispiele seien die Ebenen sieben und acht für ersteres und die Schichten neuen und zehn für zweiteres genannt.

In der Abbildung 3.10 ist der Ableitungsgraph des Planungswesens zu sehen, wobei zum allgemeinen Verständnis der Darstellung folgendes anzumerken ist. Die hervorgehobenen Namen stellen Klassen und Typenbezeichnungen dar und ein einfaches `T` mit gestrichelter Umrandung bezeichnet an Klassen einen *Template*-Parameter. Verweist ein Pfeil auf ein `T` wird von einem *Template* geerbt. Welche Klassen sich in solchen Fällen anbieten, ist dem Bild ebenfalls zu entnehmen, da die gesondert unterlegten Bereiche mehrere Klassenoptionen enthalten. Die Klassen ihrerseits beinhalten neben dem eigenen Namen weiterhin die enthaltenen Methoden und falls vorhanden auch die Beschreibung von Attributen. Ein weiteres Merkmal der Klassen- und Typennamen ist das voranstellen der

[44] compiler(engl.) = Übersetzer

drei Buchstaben SAD[45]. Dadurch wird die Zugehörigkeit aller Namen zu einem System ausgedrückt.

Nach den Erklärungen zur grafischen Darstellung der unterschiedlichen programmiersprachlichen Konstrukte soll im Verlauf der folgenden Abschnitte die Erläuterung der Abbildung erfolgen. Eine vorausschauende Anmerkung soll dennoch zum Verständnis der Abbildung abgegeben werden. Überall wo in der Darstellung die Bezeichnung Thread in irgendeiner Form auftaucht, handelt es sich gleichermaßen um ein *Template*, welches die verschiedenen Erscheinungformen des Prozesses repräsentiert. Es wurde nicht an jeglichen Klassen eine Annotation dieses *Template*-Typs vorgenommen, da die Abbildung sonst schnell an Übersicht verlieren würde.

Beginnend mit dem Typ sadActivity, welcher eine konfigurierbare Einheit und zugleich der Grundstein der Klassenhierarchie ist, wird das Systems Schicht für Schicht aufgebaut. Mit Hilfe der beiden Klassen *sadSample*[46] und *sadCompute*[47] kann der momentan laufende Prozeß abgefragt oder ein neuer Prozeß als aktiv vermerkt werden. Die Klasse *sadSample* erbringt die Aufgabe durch das Abfragen eines Zeigers, wobei *sadCompute* den aktuellen Prozeß aus dem Stack errechnen kann. Aus der funktionalen Hierarchie bekannt, setzt hier die Ebene der Belegung auf. Als Klassenrepräsentation ist *sadAssignment*[48] in der Darstellung abgebildet, wobei sie von einer der beiden Klassen des Typs sadActivity erbt. Hier besteht also erstmals eine Möglichkeit der Konfiguration. Momentan ist der Typ sadActivity fest auf *sadSample* eingestellt, da für die Version mit *sadCompute* zwar eine Implementierung vorliegt, aber das Erzeugen ausgerichteter Stacks nicht Bestandteil dieser Arbeit ist.

Die in der funktionalen Hierarchie sich anschließende Ebene der `Reihung` ist nicht im Bereich der Schedulerfamilie anzutreffen. Sie befindet sich im Prozeßbild in Abschnitt 3.1.3.3 wieder, da die Funktionalität der Errechnung und des Abfragens von Prioritäten in den Sektor des Prozesses fällt. Somit wird in der Schicht vier fortgesetzt, in der sich die `Wartelistenverwaltung` befindet. In der Abbildung ist die Klasse *Warehouse*[49] als anpaßbare Einheit vertreten, welche als *Template*-Parameter den Typ einer Speicherklasse (*Store*[50]) erhält. Sie fungiert im Grunde genommen nur als eine Art Adapter oder auch Wrapper[51], der die Anfragen an das Speichersystem weiterleitet. Dadurch kann die allgemeine Schnittstelle des Speichersystems direkt in die Hierarchie der Klassen des Planungswesen aufgenommen werden, wodurch eine einfachere und übersichtlichere Ableitungsfolge entsteht. Weiterhin kann durch diese Maßnahme das Speicher-

[45] SAD = schedule and dispatch = plane und teile zu
[46] sample(engl.) = abfragen
[47] compute(engl.) = berechnen
[48] assignment(engl.) = Zuweisung, Zuteilung
[49] warehouse(engl.) = Lager, Lagerhaus
[50] to store(engl.) = verwahren, lagern, speichern
[51] wrapper(engl.) = Hülle, Umschalg

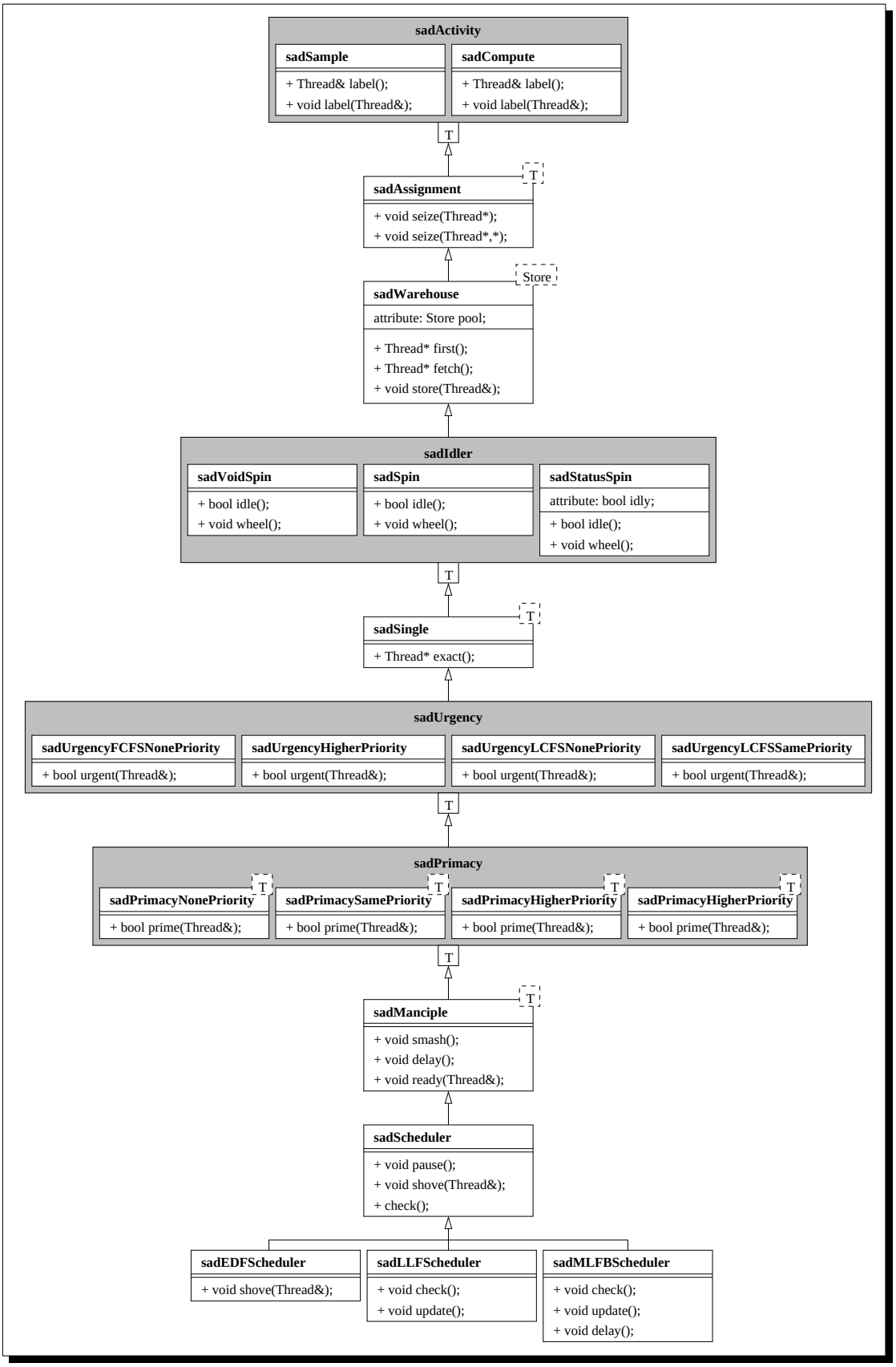


Abbildung 3.10: Klassenhierarchie des konfigurierbaren Planungswesens

system unabhängig vom Scheduler entworfen werden, da es keine Funktionalitäten der unteren Schichen in höhergelegene Ableitungsebene übertragen muß. Die Aufgabe des Mitbringens von Basisfunktionalität übernimmt somit ebenfalls die Klasse *sadWarehouse*.

Fortfahrend mit der Ebene der `Leerlaufkontrolle` ist in dem Bild ein grau unterlegter Bereich mit der Typenbezeichnung sadIdler[52] zu erkennen. Hierbei handelt es sich um eine konfigurierbare Entität des Planungswesens, welche durch drei unterschiedliche Klassen repräsentiert werden kann. In der Abbildung ist die Ableitungsfolge der drei Klassen nicht verdeutlicht, jedoch kann wenn die Darstellung betrachtet wird von links beginnend die Basisklasse *sadVoidSpin*[53] erkannt werden. Darauf aufbauend erbringt die Klasse *sadSpin*[54] ihr Funktionalität unter Nutzung der Basisklasse ein. Den Abschluß und somit die höchste Spezialisierung der `Leerlaufkontrolle` stellt die Abstraktion *sadStatusSpin*[55] dar. In der nachfolgenden Beschreibung der Klassen wird die Funktionsweise der jeweils implementierten oder erweiterten Methoden beschrieben:

sadVoidSpin ist wie der Name bereits ausdrückt eine leere Form einer `Leerlaufkontrolle`. Diese Variante wird in Systemen eingesetzt, die keine Leerlaufeigenschaft benötigen, da immer ein Prozeß auf dem Prozessor residiert. Somit ist die Methode `wheel()` leer implementiert und die Funktion `idle()` liefert immer den Wert *false*[56], was bedeutet, daß das System nie im Leerlaufmodus ist.

sadSpin stellt eine Variante der `Leerlaufkontrolle` zur Verfügung, bei der nicht immer ein Prozeß im System aktiv sein muß. Warten alle verfügbaren Prozesse auf Ereignisse oder sind sie aus anderen Gründen nicht ablaufbereit wird durch das Eintreten in die Methode `wheel()` der Leerlaufbetrieb initiiert. Daraus ergibt sich bereits die Implementierung der Routine `wheel()` als eine Schleife, welche als Abbruchbedingung das Vorhandensein eines Prozesses in der `Wartelistenverwaltung` vorsieht. Die Methode `wheel()` ist die einzige Spezialisierung dieser Klasse. Da die Methode `idle()` schon in der Basisklasse *VoidSpin* implementiert ist und diese hier keine Spezialisierung erfahren muß, erfährt sie ein Wiederverwendung. In Systemen in denen die Aktivierung von Prozessen nicht notwendigerweise durch die Konfiguration des Systems erzwungen über die `Wartelistenverwaltung` läuft, kann das beschriebene Verhalten zu inkonsitenten Prozeß- und Systemzuständen führen. Das Problem entsteht, wenn ein Prozeß der sich im *idle-loop* befindet verdrängt wird, ohne den *loop* zu verlassen. Bei einer späteren Aktivierung eben dieses Prozesses ergibt sich der inkorrekte

[52] idler(engl.) = Faulenzer, Müßiggänger
[53] void(engl.) = leer, ungültig
[54] to spin(engl.) = spulen, drehen
[55] status(engl.) = Status, Zustand
[56] false(engl.) = falsch, unwahr

Zustand, daß der Prozeß immer noch im Leerlauf ist, obwohl er eigentlich arbeiten sollte. Wenn zu diesem Zeitpunkt ein Prozeß in der `Wartelistenverwaltung` ausgemacht werden kann, erfolgt ein Kontextwechsel hin zu diesem und der vormals aktvierte aber dennoch untätige (idle) Prozeß muß erneut aktviert werden, um schlußendlich zum Rechnen zu kommen. Steht kein Prozeß in der `Wartelistenverwaltung` kann der Kreislauf von Anfang wieder beginnen. Um diesen Situationen entgegen zu wirken, wurde eine weitere Abstraktion geschaffen, die eine Zustandsabfrage des Systems gestattet. In Systemen in denen präemptive Prozeßumschaltungen direkt vorzunehmen sind, muß die Klasse *sadStatusSpin* für die `Leerlaufkontrolle` verwendent werden.

sadStatusSpin bringt die Möglichkeit einer Überprüfbarkeit der Leerlaufsituation mit sich, um das oben beschriebene falsche Systemverhalten zu unterbinden. Die Implementierung der Methoden ist aber dennoch recht einfach gehalten, indem die Methode `idle()` den Wert des Attributes *idly*[57] (siehe Abbildung 3.10) liefert. Dieser wird durch das Betreten sowie Verlassen der spezialisierten Methode `wheel()` beeinflußt. Die Spezialisierung der Funktion ist wiederum unter Verwendung der Basisklassenfunktionalität erbracht, indem das Setzen des Attributes vor und nach dem Aufruf der Methode `wheel()` der Basisklasse *sadSpin* durchgeführt wird.

Die Klassen *sadSpin* und *sadStatusSpin* verwenden zum Erbringen der Leerlaufeigenschaft den Kontext des zuletzt aktiven Prozesses. Wird ein spezieller Leerlaufprozeß benötigt, um beispielsweise die Zeit zu nutzen, in der keine anderen Prozesse aktiv sind, muß dieser als Prozeß angelegt und dem Planungswesen zugeführt werden. Da in solch einem Fall immer ein Prozeß zur Verfügung steht, kann die Klasse *sadVoidSpin* zum Einsatz kommen.

Durch den konfigurierbaren Ansatz der Ebene der `Leerlaufkontrolle` wird eine flexible Einerbung von Basisklassen in die Schicht der `Selektion` impliziert. So ergibt sich ebenfalls ein Erben von einem *Template*-Parameter bei der Klasse *sadSingle*[58], wie es bereits als Vorbild im Bereich der `Zuweisung und Belegung` geschah. Als einfache Klasse hervorgehend aus der siebenten Ebene des Schichtenmodells der funktionalen Hierarchie beinhaltet sie die Methode `exact()`, welche über die Absicherung durch die `Leerlaufkontrolle` den ersten Prozeß aus der `Wartelisteverwaltung` entfernt und diesen als Rückgabewert liefert.

Darauf aufbauend wird die Funktionalität der Ebene der `Aktivierungsentscheidung` in zwei Schichten in der Klassenhierarchie eingeteilt. Zu nennen sind die beiden entstandenen Typen sadUrgency[59] und sadPrimacy[60], wobei der erstere

[57] idly(engl.) = untätig
[58] single(engl.) = einfach, allein stehend
[59] urgency(engl.) = Dringlichkeit
[60] primacy(engl.) = Vorrang

die Funktion `urgent()` und der zweite die Methode `prime()` der funktionalen Hierarchie beinhaltet. Wie der Abbildung 3.10 zu entnehmen ist, erben sämtliche Klassen des Blockes sadUrgency direkt von der Klasse *sadSingle*. Es spannt sich hier wiederum ein Raum auf, der eine Vielzahl an Möglichenkeiten zur Konfiguration bietet. In der Ebene der `Aktivierungsentscheidung` wird nämlich darüber befunden, in wie weit Prozesse zu starten oder aufzuschieben sind. Da dies abhängig vom verwendeten Planungsverfahren unterschiedlich gehandhabt wird, muß eine Variationsvielfalt gegeben sein, die das breite Spektrum der Planungsalgorithmen abdeckt. Im folgenden werden die Unterschiede der einzelnen Klassen des Bereiches sadUrgency erklärt, wobei die Funktion `urgent()` immer einen Wahrheitswert als Ergebnis liefert, mit Hilfe dessen die `Aktivierungsentscheidung` getroffen wird. Ihre Anwendung findet die Funktion in der allgemeinen Schnittstellenfunktion des Planungswesens `shove()`, wodurch die Konfiguration der Methode `urgent()` direkten Einfluß auf Prozeßaktivierung hat. Es stehen vier Klassen für die Konfiguration des Typs sadUrgency zur Verfügung:

sadUrgencyFCFSNonePriority liefert als Rückgabewert immer einen *false* Wert. Dies hat zur Folge, daß ein rechnender Prozeß niemals durch einen anderen Prozeß der bereit gesetzt wird, verdrängt werden kann, da kein Vergleich der Prioritäten stattfindet.

sadUrgencyHigherPriority vergleicht den Prioritätswert des laufenden Prozesses mit dem bereitgesetzten Prozeß. Hat der aufkommende Prozeß eine höher Priorität als der momentan aktive Prozeß erfolgt ein Kontextwechsel hin zum neuen Prozeß. Besitzen hingegen beide Prozesse gleiche Prioritäten wird nach dem FCFS-Prinzip fortgesetzt, was gleichbedeutend mit dem Einsortieren des bereitgesetzten Prozesse in der Warteliste ist.

sadUrgencyLCFSNonePriority liefert den Wahrheitswert *true*[61] als Ergebnis zurück. Dadurch wird der Prozeß, der nach Abarbeitung verlangt, unmittelbar gestartet und der vormals aktive auf die Liste der wartenden Prozesse verbracht. Es erfolgt somit immer eine Verdrängung des laufenden Prozesses.

sadUrgencyLCFSSamePriority erbringt im Grunde genommen die gleiche Funktionalität wie *sadUrgencyHigherUrgency*, nur daß hier nach dem LCFS-Prinzip fortgesetzt wird, wenn Prioritätsgleichheit vorliegt. In dem Fall einer höheren Priorität des ankommenden Prozesses, wird er natürlich ebenfalls bevorzugt und somit zur Ausführung gebracht.

Durch die Möglichkeit der Konfiguration der Funktion `urgent()` kann der Anwender Systeme bauen, die eine Steuerung der `Aktivierungsentscheidung` mit und ohne Prioritäten unterstützen. So ist es ebenfalls denkbar, daß ein System

[61] true(engl.) = wahr, richtig

mit Prioritäten arbeitet, aber eine Unterbrechung der laufenden Berechnung nicht zulassen möchte. In solchen Fällen wird einfach eine entsprechende Auswahl aus den vier sadUrgency-Klassen vorgenommen, welche dem Anwendungsszenario am besten entspricht.

Der zweite Punkt in der Ebene der `Aktivierungsentscheidung` betrifft die Funktion `prime()`, die über die freiwillige Abgabe des Prozessors (Schnittstellenfunktion `pause()`) durch den Prozeß entscheidet. So ist nicht in allen Situationen eine Abgabe von Rechenzeit an andere Prozesse erlaubt, weil beispielsweise der nachfolgende Prozeß eine geringere Priorität aufweist. Wenn doch ein Kontextwechsel zu dem niederpriorisierten Prozeß erfolgen würde, ist eine Prioritätsumkehr entstanden, welches ein Fehlverhalten des Systems darstellt. Um solche kritischen Lagen zu vermeiden, kann der Typ sadPrimacy ebenfalls angepaßt werden. Es stehen vier verschiedene Ausführungen des Typs zur Verfügung, wobei auch hier immer ein Wahrheitswert als Ergebnis der Funktion geliefert wird. Die Klassen zeigen folgendes funktionales Verhalten:

sadPrimacyNever wird in Systemen eingesetzt, die kein freiwilliges Abgeben des Prozessors (Beispiel EDF) unterstützen dürfen. Die Funktion `prime()` liefert als Ergebnis immer den Wahrheitswert *false* zurück, wodurch die Funktion `pause()` eine leere Implementierung erhält.

sadPrimacyNonePriority ist wie der Name schon sagt, unabhängig von dem Vorhandensein von Prioritäten. Es wird immer der Wahrheitswert *true* als Ergebnis der Funktion `prime()` geliefert, wodurch ein Umschalten zum nächsten Prozeß in der Warteliste erfolgt, wenn dort Prozesse warten.

sadPrimacySamePriority verwendet den Mechanismus der Prioritäten. Die Methode `prime()` ihrerseits gibt der Wert *true* zurück, sobald eine Prioritätsgleichheit zwischen dem laufenden und dem ersten Prozeß der Warteliste besteht oder der Prozeß auf der Warteliste eine höhere Priorität hat. Infolgedessen wird ein Kontextwechsel durchgeführt.

sadPrimacyHigherPriority ist ähnlich der Klasse *sadPrimacySamePriority* implementiert, jedoch mit der Maßgabe einen Umschaltvorgang nur dann zu initiieren, wenn der Prozeß auf der Warteliste eine höhere Priorität besitzt. Somit liefert die Funktion `prime()` nur den Wert *true*, wenn der laufende Prozeß nur eine kleinere Wertigkeit vorweisen kann.

Die Adaptierbarkeit der beiden Typen sadUrgency sowie sadPrimacy entspricht dem angedachten flexiblen Design der Familie der Scheduler, weil allein durch die Kombinationsvielfalt der beschriebenen Klassen der zwei Typen sich zwölf Varianten der Schicht `Aktivierungsentscheidung` erstellen lassen. Durch den Mechanismus der austauschbaren Basisklassen ist ein einfaches Zusammensetzen der Klassenbausteine gegeben.

Weiter fortführend setzt hierauf die Ebene der allgemeinen Schnittstelle auf, welche gleichermaßen in zwei Klassen aufgespaltet wurde. Es sind die bekannten Funktionen in den Klassen *sadManciple* und *sadScheduler* zu betrachten. Ebenso ist bereits die einfache Version der `Zeitscheibenanbindung` (`check()`) in der Klasse *sadScheduler* untergebracht, da sie in dieser Variante nur einen Wrapper darstellt. Es erfolgt nur eine Weiterleitung des Aufrufes an die Methode `pause()` der selben Klasse.

In den nachfolgenden Ableitungen der speziellen Scheduler sind die Funktionen vermerkt, welche eine Erweiterung oder auch eine neue Implementierung oftmals unter Nutzung der Funktionalität der Basisklasse erfahren. Der Strategiename ist Bestandteil des Klassennamens, wodurch die Anwendbarkeit des Schedulers klar erkennbar ist.

Durch den anpaßbaren Aufbau der gesamten Klassenhierarchie des Planungswesens ist eine immense Anzahl an Kombinationen gegeben. Somit kann dem Anwender ein Scheduler zur Verfügung gestellt werden, der auf seine Problemstellung hin zugeschnitten ist. Sollte ein Verfahren oder eine bestimmte Vorgehensweise beim Planen noch nicht Bestandteil sein, so kann eine Erweiterung des flexiblen Systems geschehen. Die Variationspunkte sind bereits im Verlaufe der Erklärungen bekannt geworden. Der Abschnitt 3.1.3.5 beschäftigt sich dann aber noch einmal etwas ausführlicher mit dem Gebiet.

3.1.3.3 Prozeßbild

Ebenso wie das Planungswesen ist der Aufbau des Prozeßbereiches möglichst flexibel zu halten. Als Ausgangspunkt dient eine Prozeßbibliothek, welche zwei Grundfunktionen zum Beenden und zum Wechseln von Prozessen anbieten muß. Darauf aufsetzend, kann mit der schichtweisen Erweiterung des Prozeßbildes fortgesetzt werden.

Der Entwurf der Klassenhierarchie der Prozeßabstraktion wird gleichermaßen mit Hilfe der anpassungsfähigen Ableitung durch den Mechanismus der *Templates* erbracht. Daraus hervorgehend ist eine individuelle Anpaßung an die Gegebenheiten der Problemstellung möglich, wodurch eine optimale Unterstützung der Anwendungsstruktur gewährleistet wird.

Bei der Betrachtung die Klassenhierarchie der Prozeßabstraktion kann erkannt werden, daß sie sich unmittelbar aus dem Modell der funktionalen Einheiten ergibt. Auf der Basis der Prozeßbibliothek aufsetzend sind zunächst die nötigen Daten und Strukturen für die Prozeßverwaltung zur Verfügung zu stellen. In der Abbildung 3.11 können die Klassen *teLinkage*[62] und *tePriority*[63] betrachtet werden, welche die `Verwaltungsstrukturen` bereitstellen. Das Namenskürzel *te* vor sämtlichen Klassenbezeichnungen ist ausgeschrieben **t**hread **e**xtension, was in die deutsche Sprache übersetzt, Fadenerweiterung bedeutet. In der Fachsprache

[62] Linkage(engl.) = Verknüpfung
[63] priority(engl.) = Priorität

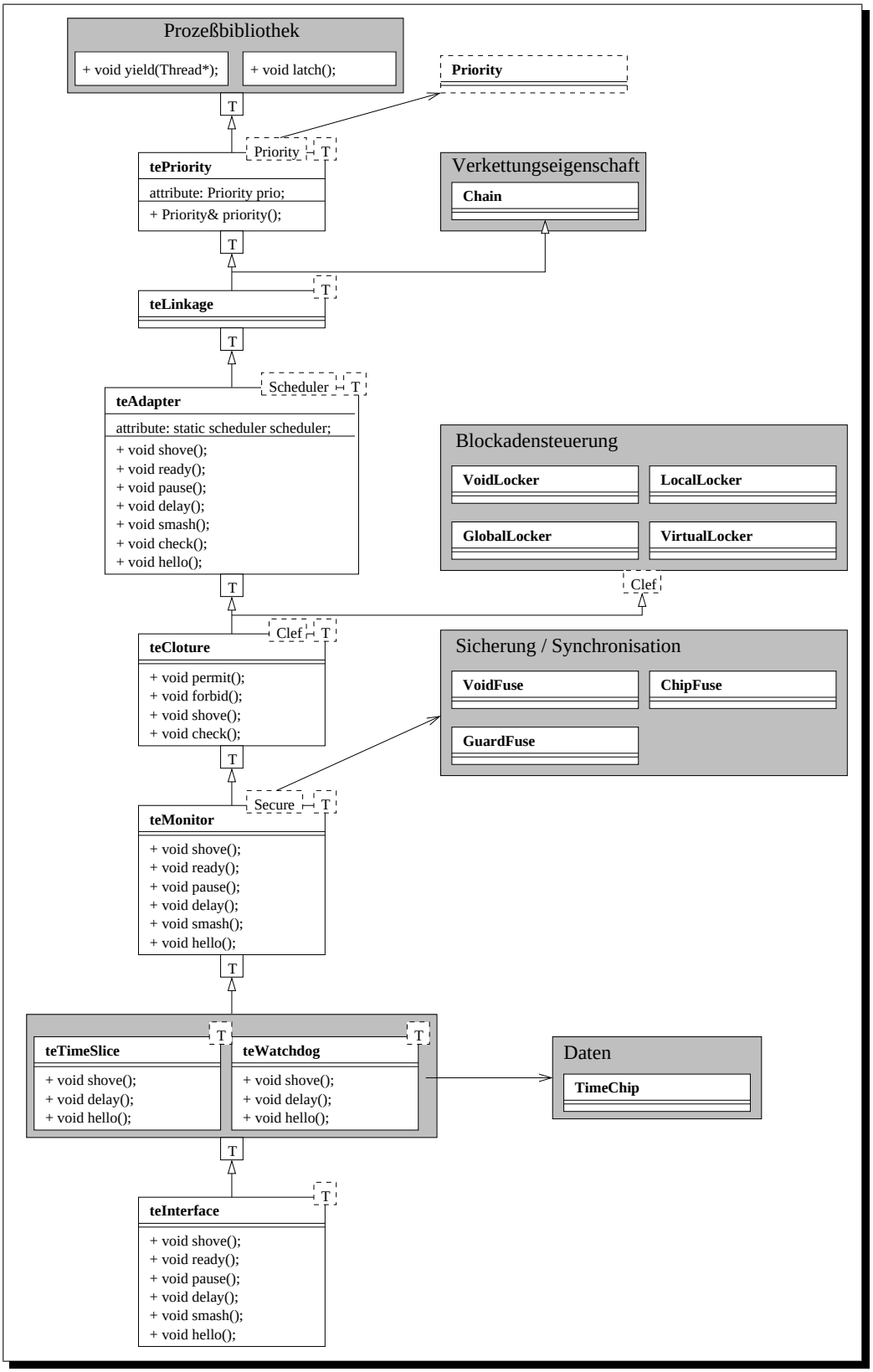


Abbildung 3.11: Klassenhierarchie der konfigurierbaren Prozeßabstraktion

wird oft von Fäden bzw. Prozeßfäden gesprochen, weshalb das Wort Faden auch als Synonym für einen Prozeß verwendet wird. Der Darstellung ist zu entnehmen, daß jede Ableitung ein Erben von einem *Template* bedeutet, so daß dies im folgenden nicht immer zusätzlich erwähnt werden muß. Weiterhin bedeutet dies ebenfalls, daß auch einzelne Ebenen fehlen dürfen, wenn kein Bedarf an ihnen besteht. Zu den Strukturen für die Verwaltung ist hinzuzufügen, daß die Klasse *teLinkage* einzig und allein die Aufgabe hat, die Verkettungseigenschaft der Prozeßabstraktion zugänglich zu machen. Sie erbt aus diesem Grund von einer Basisklasse (*Template*: Prozeßbasistyp aus der Prozeßbibliothek) und von der Klasse *Chain*[64], welche eben dieses Merkmal aufweist. Der zweite Bestandteil der Verwaltungsdaten ist die Priorität, wobei sie natürlich gleichermaßen nur im Bedarfsfall vorhanden ist. Wird sie benötigt, erbringt *tePriority* das Bereitstellen einer Priorität. Durch den Mechanismus der Schablonen können beliebige Wertigkeiten verwendet werden, wobei als Typen die schon beschriebenen Prozeßparameter aus Abschnitt 3.1.2.5 in Frage kommen. Die Abbildung 3.8 im erwähnten Abschnitt spiegelt nicht nur die funktionalen Zusammenhänge wieder, sondern stellt ebenfalls gleichzeitig die Vererbungsbeziehungen der Prioritäten dar.

Nachdem die Verwaltungsdaten in die Prozeßabstraktion integriet wurden, setzt auf diesen die Verbindungsschicht zum Scheduler auf. Die Klasse *teAdapter* enthält einen statischen Planer, welcher mit Hilfe eines *Template*-Parameters (*Scheduler*) spezifiziert wird. Eine weitere Möglichkeit wäre hier eine dynamische Variante, die einen Verweis auf den entsprechenden eigenen Planer hält. Wird ein solches Verhalten benötigt, so ist eine dahingehende Klassenimplementierung mit gleicher Schnittstelle zu entwickeln. Dieser Punkt zeigt sehr anschaulich die Erweiterungsfähigkeit des Systems. Durch die Kopplung des Planungswesens zum Prozeß kann ein einfaches Aufrufen der Planungsfunktionen durch den Prozeß stattfinden, wobei der Adapter die richtige Übergabe der Parameter an die Funktionen des Schedulers regelt.

In der nächsten Ebene der funktionalen Einheiten schließt sich die `Blockadensteuerung` bzw. die Steuerung des `Sperrmechanismus` an. Die verwendete Klasse *teCloture*[65] erbt einerseits von der Basis des Prozesses und anderseits von einer weiteren Klasse *Clef*[66], die als *Template*-Parameter von der Sperreigenschaft abstrahiert. In der Abbildung sind die Funktionen zu erkennen, welche im Zuge der Erweiterung des Prozeßmodells angepaßt werden. Es handelt sich hierbei um die beiden Funktionen `shove()` und `check()`, die Umschaltvorgänge von außerhalb des Prozesses initiieren können, und um die Funktionen `permit()` und `forbid()`, welche die Blockade setzen oder wieder aufheben. Über die ebenfalls vorhandene Funktion `avail()` kann der Zustand der Blockade abgefragt werden. Das Erbringen der Blockadeeigenschaft kann auf vielfältige Weise erbracht werden, weshalb

[64] chain(engl.) = Kette
[65] cloture(engl.) = Verschluss
[66] clef(engl.) = Schlüssel

mehrere Varianten zur Verfügung stehen. Aus dem Bereich der Merkmalsmodelle sind vier verschiedene Verfahren bekannt. Zu nennen sind folgende:

VoidLocker [67] stellt dem System leere Implementierungen der beiden Sperrfunktionen `permit()` und `forbid()` zur Verfügung.

LocalLocker ist – wie der Name bereits sagt – die Form, in der die Prozesse eine lokale Sperrung erbringen. Die Sperre hat nur für die Ausführungszeit des Prozesses Auswirkung auf das Planungswesen. Sobald ein freiwilliger Prozeßwechsel durchgeführt wurde, ist die Sperre des dann aktiven Prozesse von Gültigkeit.

GlobalLocker erbringt ein Sperren auf globaler Ebene zustande. Die Auswirkung der Sperre ist auch nach Prozeßwechseln unverändert. Somit muß explizit die Sperre gesetzt oder aufgehoben werden.

VirtualLocker stellt die Blockadesteuerung der den Prozessen in Form einer virtuellen Funktion zur Verfügung. Dabei ist die Methode `avail()` virtuell ausgelegt, damit jeder Prozeß eine auf sich angepaßte Variante des Sperrens implementieren kann, indem er eine Spezialisierung der virtuellen Funktion vornimmt. So wird es dem Prozeß überlassen, ob eine Berechnung den Ausschlag für eine Sperrung gibt oder ob immer feste Werte geliefert werden. Mit dieser Methodik können sogar zeitabhängige Sperren realisiert werden, weshalb sie die flexibelste Form darstellt.

Eigentlich ist die Klasse *VoidLocker* nicht unbedingt nötig, da die Schicht des Sperrens ebenso komplett fehlen darf, wenn es beispielsweise keinen Bedarf an Sperrmechanismen auf Prozeßebene gibt. Jedoch kann durch die Möglichkeit einer leeren Implementierung des Sperrens (*VoidLocker*) die Anwendung ohne die Veränderung des Quelltextes weiterhin getestet werden, ob die Sperrung des Planungswesens überhaupt von Bedeutung ist. Wenn der Test ergibt, daß die `Blockadesteuerung` nicht gebraucht wird, kann das System Speicher und Laufzeit durch das Entfernen der Ebene einsparen, weil das Abfragen sowie Aktualisieren vollständig entfällt.

In der sich anschließende Ebene sind die `Synchronisierungsmechanismen` der Prozesse untergebracht. Da Prozesse auf ein und demselben Scheduler operieren, welcher ein Betriebsmittel mit Datenstrukturen darstellt, müssen Prozesse aufgrund von möglichen Nebenläufigkeitserscheinungen synchronisiert werden. Die Klasse *teMonitor*[68] übernimmt die Aufgabe der Synchronisierung, wobei durch den *Template*-Parameter *Secure*[69] die Art der Synchronisation anzugeben ist. Wiederum werden hier verschiedene Herangehensweisen unterstützt. So ist ein

[67] locker(engl.) = Schrank, Schließfach
[68] monitor(engl.) = Überwachungsgerät, Bildschirm
[69] secure(engl.) = sicher, sicherstellen

allgemeines Aussperren von anderen Prozessen durch Abschalten von Unterbrechungen möglich oder es kommt eine andere Form der Synchronisierung zum Einsatz. Da jedoch der Schwerpunkt der Arbeit nicht im Bereich der Synchronisierung liegt, ist auf [HH89] verwiesen. Es stehen aber Möglichkeiten der Synchronisierung zur Verfügung (siehe Abbildung). Zu den dargestellten Varianten ist folgendes zu erwähnen:

VoidFuse [70] findet Anwendung, wenn im System keine nebenläufigen Aktionen vorhanden sind oder die Integrität der Ressource Planungswesen zu jedem Zeitpunkt sichgestellt ist, jedoch die Anwendung bereits mit Synchronisationsanweisungen implementiert wurde. Weiterhin kann sie zu Testzwecken eingesetzt werden.

ChipFuse [71] erbringt das Synchronisieren durch Abschalten von Unterbrechnungen, wodurch infolgedessen Unterbrechungen verloren gehen können. Dieses Verfahren wird in vielen Systemen eingesetzt, da es keinen zusätzlichen Aufwand im Systembereich verursacht.

GuardFuse [72] ermöglicht eine Synchronisation von Prozessen, jedoch ohne das Unterbinden von Unterbrechnungen. Ein gesteigerter Verwaltungsaufwand auf Systemebene ist aber die Folge. Die Funktionsweise der Synchronisation ist in [SSPSS00] beschrieben.

Gemäß der Abbildung 3.11 sowie dem funktionalen Einheitenmodell folgend, schließt sich die Ebene der `Zeitensteuerung` an. Im Klassendiagramm sind die Klassen *teWatchdog*[73] und *teTimeSlice*[74] verzeichnet. Es ist erkennbar, daß sie einen Timerbaustein (*TimerChip*) verwenden, um ihre Aufgabe wahrzunehmen. Weiterhin soll durch die Darstellung ausgedrückt werden, daß beide Module nicht gleichzeitig im System vorhanden sein können. Die Klasse *teWatchdog* ist in Systemen mit Prozeßzeitüberwachung einzusetzen, wohingegen die Klasse *teTimeSlice* verwendet wird, wenn jedem Prozeß eine volle Zeitscheibe zugeteilt und garantiert werden soll.

Die Klasse *teInterface*[75] schließt die Prozeßabstraktion auf Systemseite ab. Hier sind nocheinmal alle Funktionen zusammengetragen, die garantiert immer Bestandteil der Klassenhierarchie sind. Es wurde ein allgemeines Interface als Abschluß der Abstraktion genommen, um eine einheitliche Schnittstelle dem Anwendungsprogrammierer zu offerieren. Intern erfüllt es die Aufgabe der Abgrenzung von der Anwendung bzw. den Aufrufsemantiken der Planungsfunktionen

[70] fuse(engl.) = Sicherung
[71] chip(engl.) = integrierter Schaltkreis
[72] guard(engl.) = Schutz, Wache, Wächter
[73] watchdog(engl.) = Wachhund
[74] timeslice(engl.) = Zeitscheibe
[75] interface(engl.) = Schnittstelle, Verbindung

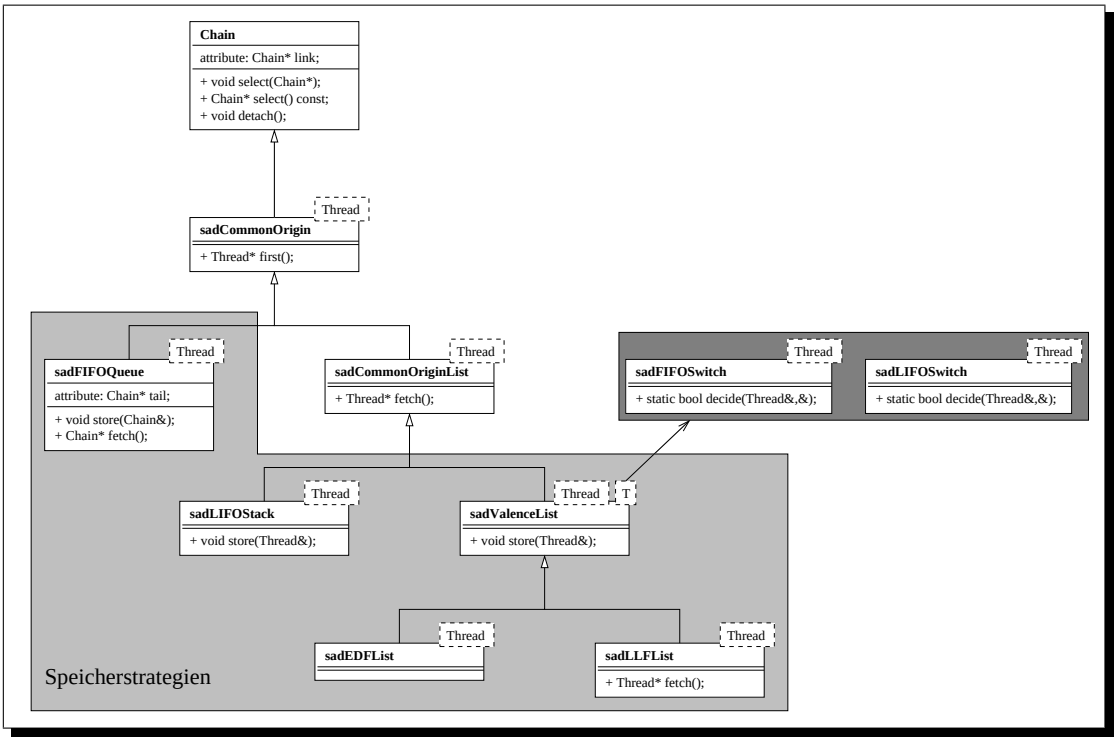


Abbildung 3.12: Klassenhierarchie des Speicherwesens

auf Anwendungsebene. Dadurch kann Einfluß auf die Strategie bzw. den Prozeß der Inline-Expansion (Optimierung durch den Übersetzer/Compiler) innerhalb des SAD-Systems genommen werden.

Der Anwender des Prozeßmodells nimmt die Klasse *teInterface* als Basis für seine weiteren Ableitungen. Er füllt damit den Prozeß mit Leben, indem er die zu erbringenden Aufgaben auf programmiersprachlicher Ebene formuliert. Eine gute Erweiterung stellt die Schnittstelle *tie*[76] [SP01] der Prozeßbibliothek TAL dar, weil sie mehrere Varianten für die Aktivierung des Anwendercodes bietet.

Zusammenfassend kann gesagt werden, daß eine hoch flexible Prozeßabstraktion entstanden ist, welche sich sehr vielen unterschiedlichen Gegebenheiten anpassen kann. Dadurch kann eine große Anzahl an System bereits unterstützt werden. Einer Erweiterung des Prozeßbildes steht ebenfalls nichts im Wege, wodurch auch momentan nicht angebotene Fähigkeiten hinzugefügt werden können.

3.1.3.4 Speicherverwaltung

Nachdem die Klassenhierarchien des Planungswesens und der Prozeßabstraktion behandelt wurden, ist im folgenden die Sturktur und der Aufbau des Spei-

[76] **t**hread **i**nterface **e**xtension(engl.) = Fadeninterfaceerweiterung

chersubsystems zu betrachten. Ausgehend von der Klasse *Chain*, welche bereits im Prozeßbild ihre Anwendung fand, wird das System schichtenweise aufgebaut. Hierbei stellt die Klasse *Chain* die allgemeine Verkettungsfähigkeit sowie Operationen darauf bereit. So gehören auch die bekannten Funktionen `select()` jeweils zum Auslesen und Setzen des Nachfolgers sowie die Methode `detach()` zu dem Tätigkeitsfeld dieser Abstraktion.

Allen weiteren Klassen im Ableitungsgraphen (Bild 3.12) ist der *Template*-Parameter *Thread* gemein. Durch diese Maßnahme kann jegliche Prozeßabstraktion aus dem vorangegangenen Abschnitt direkt im Bereich des Speicherwesens verwendet und eingetragen werden, sobald sie selber die Eigenschaft der Verkettung (*teLinkage*) trägt. Überdies sind deshalb die Algorithmen für die Organisation der verschiedenen Speicherstrategien völlig unabhängig vom Modell des Prozesses.

Ausgehend von der für alle listen- oder schlangenorientierten Verfahren allgemeingültigen Funktion `first()` wird im weiteren Vererbungsbaum eine Anreicherung der Funktionalität stattfinden. Die Klasse *sadCommonOrigin*[77] erbt die Eigenschaften der Verkettung sowie den Funktionsumfang der Klasse *Chain* und reichert diesen mit der Methode `first()` an, wobei jene unter Nutzung der ererbten Methoden ihre Aufgabe erbringt.

sadCommonOrigin realisiert die Funktion `first()`, welche als Ergebnis den ersten Eintrag aus der Speicherstruktur liefert. Da dies für alle Verfahren gleich ist, die mit Verkettungsbeziehungen arbeiten, wurde die Methode in einer eigenen Klasse angesiedelt, damit keine Mehrfachimplementierung erfolgen muß.

In der Abbildung 3.12 ist das gesamte Klassendigramm des Speichersubsystems auf einen Blick zu sehen. Daraus ist zu entnehmen, daß in der nächsten Schicht eine Aufspaltung in ein schlangen- und ein listenbasiertes Speicherverfahren geschieht. Unter einer Schlange wird im allgemeinen eine Datenstruktur mit zwei Zeigern verstanden, wobei der eine Zeiger auf das erste und der andere auf das letzte Element verweist. Die listenorientierten Verfahren verwenden hingegen nur einen Zeiger, welcher das Anfangselement identifiziert. In der Darstellung sind auf der einen Seite die Klasse *sadFIFOQueue*[78] und auf der anderen die Klasse *sadCommonOriginList*[79] zu erkennen. Eine Aufteilung in zwei Klassen für die Verfahren bringt den Vorteil einer effizienteren Implementierung speziell für die schlangenorientierte Strategie.

Es zeigt sich, daß dabei beide Klassen die Funktion `fetch()` implementieren, jedoch arbeiten diese intern mit unterschiedlichen Aufrufsequenzen der Basisfunktionen. Die Klasse *sadFIFOQueue* beinhaltet zusätzlich noch die Platzierungs- und Speicherfunktion `store()`. In späteren Ableitungen erfolgt auch im Teilbe-

[77] common(engl.) = allgein; origin(engl.) = Ursprung
[78] queue(engl.) = Warteschlange, Reihe
[79] list(engl.) = Liste, Verzeichnis, Aufzählung

reich der listenorientierten Strategien eine Verwirklichung der Einsortierfunktion. Die beiden Klassen weisen folgendes Verhalten auf:

sadCommonOriginList implementiert die Funktion `fetch()` einfach durch ein Auslesen des ersten Listeneintrages, falls überhaupt ein Element Vorhanden ist und dem darauf folgenden Weitersetzen des Anfangszeigers der Struktur. Dadurch ist bereits das Entfernen geschehen.

sadFIFOQueue erbringt das Entfernen ähnlich dem eben beschriebenen Verhalten, jedoch mit der zusätzlichen Neuausrichtung des *tail*[80]-Zeigers, falls das Warteschlangenende erreicht wurde.

Die Funktion `store()` als zweite Methode der Klasse fügt ein Element in die Warteschlange mit Hilfe des *tail*-Zeigers ein. Durch die Nutzung des zusätzlichen Zeigers kann ein Durchlaufen der gesamten Warteschlange verhindert werden, da das Ende ständig über den Verweis erreichbar ist.

Die Klasse *sadFIFOQueue* stellt eine der verwendbaren Einheiten für das im Planungswesen angesiedelte Warenlager (Klasse *sadWarehouse*) dar. So sind alle Klassen in dem grau unterlegten Bereich `Speicherstrategien` mögliche *Template*-Parameter eben dieser Abstraktion.

Damit nicht nur Verfahren Verwendung finden können, die auf Datenstrukturen mit Schlangenverwaltungen beruhen, muß eine Erweiterung der Klasse *sadCommonOriginList* vorangebracht werden, um auch auf andere Betriebsarten eingestellt zu sein. Allerdings sind zwei Möglichkeiten der weiteren Herangehensweise in Bezug auf Speicherstratgien denkbar. So ist der Abbildung einmal die Klasse *sadLIFOStack* sowie ein andermal die Klasse *sadValenceList*[81] zu sehen, welche jeweils eine Spezialisierung der Klasse *sadCommonOriginList* aufzeigen. Das Spezialisieren bzw. Erweitern der Funktionalität wird durch die Methode `store()` erreicht. Es entstehen Klassenabstraktionen, die direkt als Parameter im Bereich der `Wartelistenverwaltung` des Planungswesens eingesetzt werden können. Folgendes Verhalten wird mit den beiden Klassen assoziert:

sadLIFOStack entspricht einem Kellerspeicher, der nach dem Prinzip `LIFO` aufgebaut ist. Somit sortiert die Methode `store()` das Prozeßelement an der ersten Stelle der Liste ein. Ein Entfernen wird durch die Methode `fetch()` durchgeführt, so daß immer das zuletzt eingefügte Element entfernt wird, was die richtige Verarbeitungsreihenfolge nach `LIFO` auch verlangt.

sadValenceList benutzt, wie der Name bereits ausdrückt, Wertigkeiten zum Auffinden der richtigen Einfügeposition. Die Funktion `store()` durchsucht die Liste, wobei die Priorität des einzufügenden Elementes mit dem jeweiligen Element der Suchposition mit Hilfe der Funktion `decide()` verglichen

[80] tail(engl.) = Schwanz, Ende
[81] valence(engl.) = Valenz, Wertigkeit

wird. Beim Durchwandern der Liste erfolgen gegebenenfalls Anpassungen der Wertigkeiten, sobald dies ein Verfahren (z.B. EDF) verlangt. Ist die richtige Prosition gefunden, erfolgt der einfache Einfügevorgang.

Die Klasse *sadValenceList* kann konfiguriert werden, was in der Abbildung durch den *Template*-Parameter *T* angedeutet ist. Es kann die Sortierreihenfolge der Prioritätenliste beeinflußt werden, indem einer der beiden möglichen Parameter *sadFIFOSwitch*[82] und *sadLIFOSwitch* Verwendung findet. Die Semantik hinter den Klassennamen offeriert bereits sehr gut die Auswirkung auf die Reihenfolge der Sortierung. Die Klassen bieten die Funktion `decide()` an, welche einen Wahrheitswert in Abhängigkeit der Ausprägung sowie der Prozeßprioritäten liefert. Die Funktion ist statischer Natur, damit kein konkretes Objekt zur Laufzeit für den Aufruf existieren muß, aber dennoch eine Kapselung als Klasse möglich ist.

Den letzten Schritt in der Ableitungsfolge stellt die Spezialisierung der Klasse *sadValenceList* dar. Durch den Einsatz besonderer Planungsstrategien wird auch in Einzelfällen eine Erweiterung der Speicherfunktionalität nötig. So wird im Falle der Klasse *sadLLFList* eine spezifische Implementierung der Funktion `fetch()` vorgenommen, die unter Verwendung der Basisfunktion `fetch()` das Ausketten aus der Liste erbringt und im folgenden eine Prioritätsanpassung des nun ersten Elementes der Warteliste vornimmt. Im Gegensatz hierzu ist eine Verfeinerung der Basisfunktionalität bei anderen momentan verfügbaren Planungsverfahren nicht nötig. Als Beispiel hierfür wurde in der Abbildung die Klasse *sadEDFList* nur als neuer Name für *sadValenceList* eingefügt, um auf der Ebene der späteren Konfigurationsbeschreibung ein einheitliches Bild im Bezug auf Namensbeziehungen zu haben. Ebenso ist auch ein weiterer Name für das Speicherinterface des Planungsverfahren *MLFB* vorhanden, aber aus Gründen des besseren Verständnisses der Darstellung vernachlässigt worden.

3.1.3.5 Erweiterungsmöglichkeiten

Das entstandene Softwaresystem bietet eine immense Auswahl an verschiedenen Verfahren des Planens sowie eine Unterstützung für unterschiedlichste Ausprägungen auf Prozeßebene. Da ein System sowie ein lebendiges Design niemals als vollständig abgeschlossen erachtet werden kann, sollte immer die Möglichkeit einer Erweiterung gegeben sein. Jedoch differiert die Machbarkeit sowie die Einfachheit zusätzliche Mechanismen einzubauen von Softwaresystem zu Softwaresystem sehr stark. Das vorliegende System ist durch die hochgradige Strukturierung und der damit verbundenen feingranularen Klassenbibliothek offen für Erweiterungen in jeglichem Subsystem. Desweiteren sind viele Bestandteile des Systems konfigurierbar ausgelegt und nicht durch feste Vererbungsbeziehungen

[82] switch(engl.) = Schalter

ausgedrückt, wodurch bei Einhaltung der Schnittstellendefinition eine Erweiterung stattfinden kann.

So ist etwa in den unterschiedlichen Teilbereichen des Planungswesens, der Prozeßabstraktion sowie den Prozeßparametern und dem Speichersubsystem sicherlich weiterhin Entwicklungsarbeit zu leisten. Durch die *Template*-Programmierung in den verschiedenen Subsystemen kann neue Funktionalität in bereits vorhandenen Systemteile ohne größeren Aufwand eingebracht werden. Ebenso ist eine Aufspaltung einzelner Klassen möglich, um neue Konfigurationspunkte im System zu schaffen.

Eine Aufstellung von Erweiterungsmöglichkeiten der verschiedenen Teilgebiete ist nachfolgend zu sehen. Beginnend mit dem Bereich Planungswesen sind zu nennen:

Planungswesen :

- zusätzliche Planungsverfahren einbetten
- Nutzung andere Prozeßbibliotheken
- Unterstützung von Prozeßabhängigkeiten bzw. -beziehungen
- zeitgesteuertes Aktivieren von Prozessen in Verbindung mit einem differenzierteren und angepaßten Leerlaufverhalten
- Reihenfolgeaktivierungen (OSEK) von Prozessen zulassen
- mehrere Scheduler im System vs. Anwendungsscheduler
- Multiprozessorscheduling
  - ein vs. mehrere Scheduler in einem System
  - per Prozessor Scheduling
  - feste Prozeß-Prozessor Zuordnung
  - ...
- Optimierungen im Detail (Inliningstrategie)
- ...

An das Teilgebiet der Prozeßabstraktion können ebenfalls zusätzliche Anforderung gestellt werden, welche infolge von Erweiterungen des Modelles zu befriedigen sind. Einige der nachstehend erwähnten Zusätze müssen in Einklang mit dem Planungswesen erbracht werden, sonst wird eine Umsetzung der Funktionalität nicht gegeben sein. So sind auch einige bereits beim Planungswesen vermerkte Punkte hier wieder zu finden. Das Modell bzw. die Abstraktion des Prozesses kann somit durch nachstehende Eigenschaften angereichert werden:

Prozeßabstraktion :

- Nutzung andere Prozeßbibliotheken

- Unterstützung von Prozeßabhängigkeiten bzw. -beziehungen
- Mechanismen zur Reihenfolgeaktivierung (OSEK) von Prozessen
- Möglichkeiten der Betriebsmittelüberwachung speziell Ressourcenverbrauchsmessung für die Abrechnung von Leistungen
  - System-, Benutzer-, Prozessor-, Warte- vs. Realzeiten
  - Speicherverbrauch
  - ...
- Mehrfachaktivierungen von Prozessen (OSEK)
- Optimierungen im Detail (Inliningstrategie)
- ...

Die beiden Gebiete Planungswesen und Prozeßabstraktion gehören thematisch sehr eng zusammen und müssen deshalb in sehr vielen Fällen gleichzeitig weiterentwickelt werden. Für das dritte klar abgegrenzte Subsystem, dem Speichersystem, trifft dies nicht zu, da die Bindung und die Kommunikation des Teilsystems vom und zu dem Restsystem über eine fest definierte Schnittstelle abläuft, weshalb es relativ unabhängig vom Restsystem weiterentwickelt werden kann. Mögliche Zusätze zu den vorhandenen Abstraktionen wären folgende:

Speichersubsystem :

- andere Speicherstrategien, unter der Bedingung gleicher Schnittstelle
  - Bäume
  - Felder
  - Hashtabellen
  - ...
- Entwicklung unterbrechungstransparenter Speicherstrategien und Strukturen
- multiprozessorsichere Vorgehensweisen
- ...

Zusammenfassend kann gesagt werden, daß durch die Struktur der Software sowie dem allgemein konfigurierbaren Ansatz des Designs einer Weiterentwicklung unter Verwendung der Basisabstraktionen nichts im Wege steht. So können sämtliche Basisklassen als Ausgangspunkt für die Erweiterung und Anreicherung des Systems um neue Funktionalität dienen. Aus diesem Grund kann das System eine ständige Erweiterung erfahren.

3.2 Konfiguration

Nachdem der allgemeine Entwurf in all seinen zahlreichen Fassetten beleuchtet wurde, muß im folgenden die Konfiguration der Teilbereiche gezeigt werden. Zunächst ist im weiteren Verlauf die Konfigurations- und Kombinationsvielfalt als markantes Merkmal des Softwaresystems beschrieben, wodurch noch einmal die hohe Flexibilität des Design zum Ausdruck kommt. Der sich anschließende Abschnitt zeigt die allgemeine Vorgehensweise der Konfiguration, welche nachfolgend an zwei Beispielszenarien gezeigt und validiert wird.

3.2.1 Konfigurations- und Kombinationsvielfalt

Bereits während der Begriffserklärung sowie dem Aufbau der Merkmalsmodelle konnte die extreme Vielfalt der Kombinationsmöglichkeiten erahnt werden. Im Bezug auf die Modelle sind im Ganzen weit über 400 Varianten erstellbar. Das Problem hierbei ist, daß nicht alle Einstellung sich wirklich bis auf die Implementierungsebene durchziehen. So ist etwa bei dem Merkmal *process count* nicht notwendigerweise ein anderes Planungssystem zu erstellen, nur weil ein Wechsel der Eigenschaft von *unlimited* zu *limited* erfolgt. Es handelt sich somit nicht um eine funktionale Eigenschaft des Systems, sondern um ein nicht funktionales Merkmal, welches nicht bis zur Ebene der Klassen und Implementierung reicht. Davon abgesehen, sind ebenfalls nicht alle Kombinationen der Merkmale in diesen Modell sinnvoll (Vergleiche hierzu 3.1.1.3). Prinzipiell erzeugen dennoch sehr viele Kombinationen eine durchaus verwendbare Konfigurationsbeschreibung.

Die Vielfalt der Möglichkeiten auf der Ebene der Merkmalsmodellierung ist durch das Design und die anschließende Implementierung umgesetzt, wodurch ein immenser Konfigurationsraum entstand. So können allein im Bereich des Planungswesen 384 verschiedene Scheduler gebaut werden, welche wiederum einerseits keine und andererseits teilweise spezielle Anforderungen an die Prozeßabstraktion stellen. Durch die im Prozeßbereich ebenfalls gegebenen Vielfalt von 144 Varianten der Abstraktion kann das Planungswesen in seinem Tätigkeitsfeld eine optimale Unterstützung erfahren. Eine weitere Anreicherung an Möglichkeiten bietet das Gebiet der Prioritäten, wobei die Vielfalt der Varianten noch nicht einmal in der Zahl 144 enthalten ist. Somit vergrößert sich der Konfigurationsspielraum weiterhin.

Werden die Bereiche Planungswesen und Prozeß als gemeinsamer Komplex betrachtet, was der natürlichen Sicht entspricht, sind die Möglichkeiten mit 55296 Variationen enrom. Die 384 Planungsvarianten können mit 144 Prozeßabstraktionen kombiniert werden, wobei angemerkt sei, daß nicht alle Kombinationen sinnvoll einsetzbar sind. Jedoch ist hierdurch eine Anzahl an verschiedenen Konfigurationen möglich, welche selbst die aus den Merkmalsmodellen bekannte Dimension übersteigt. Dadurch kann das gesamte Softwaresystem weitreichender konfiguriert werden, als es durch die Modellbeschreibung der Eigenschaftsdia-

gramme dargestellt werden kann. Dies ist auch von Vorteil, da bei besonderer Kenntnis des Systems eine Konfiguration erstellt werden kann, ohne über die Merkmalsbeschreibung eine Auswahl von Eigenschaften vorzunehmen.

Da das System nicht als abgeschlossen gilt, kann immer eine Erweiterung erfolgen, infolge dessen sich die Möglichkeiten der Konfiguration weiterhin vervielfältigen. Dem Aufbau zusätzlicher Funktionalität und der damit einhergehenden Kombinationsvermehrung steht nichts im Wege.

3.2.2 Allgemeine Vorgehensweise

Durch die immensen Kombinationsmöglichkeiten ist es notwendig, den Vorgang der Konfiguration nach einem gewissen Schema durchzuführen, um nicht die Übersicht beim Prozeß der Konfiguration zu verlieren. Aus diesem Grund wird folgende modulare Herangehensweise vorgeschlagen, bei der die Konfiguration in drei mehr oder minder einzelne Teile aufgespaltet ist. Zunächst ist eine Basisabstraktion für eine planbare Einheit zu schaffen, welche alle wichtigen Informationen für die Planung bereitstellt. Daran anschließend muß der Planer zusammengestellt werden, der diese Einheiten verwaltet und zur Ausführung bringen kann. Im Anschluß an diesen Schritt wird die Basisabstraktion des Prozesses um weitere Eigenschaften erweitert, um gemäß der Modellbeschreibung zusätzliche Merkmale und Fähigkeiten zu erhalten.

Aus den vorangegangenen Abschnitten ist bekannt, daß die Konfiguration mit Hilfe von *Templates* und *Typedefs* erbracht wird. Um die Konfigurationsbeschreibung leichter verständlich zu halten, ist eine allgemeine Klassensicht beibehalten. Es werden jedoch nur Typendefinitionen vorgenommen und keine neue Funktionalität hinzugefügt.

3.2.2.1 Konfiguration einer planbaren Einheit

Beginnend mit der Basisabstraktion für Prozesse kann diese in der Abbildung 3.13 betrachtet werden, wobei zu erkennen ist, daß die Klasse *TBD*[83] ebenfalls einen *Template*-Parameter (*Base*) erhält. Bei jenem handelt es sich um die Basisklasse des Prozesses aus der Prozeßbibliothek, die die Grundfunktionalität für Prozeßwechsel und -beendigungen bereit stellt. Wird die Spezifikation der Klasse *TBD* genauer untersucht, ist festzustellen, daß hier die Klassen für die Verwaltung durch das Planungswesen konfiguriert werden. Dabei sei angemerkt, daß die Reihenfolge der Strukturen austauschbar ist, weil sie unabhängig von einander sind und nicht aufeinander aufbauen. Ergo wird eine Einheit zusammengestellt, die durch einen Planer verarbeitet werden kann. Für den entstehenden Typ ThreadBase innerhalb von *TBD* sind vier mögliche Konfigurationen zulässig, wenn die Vielfalt der Prioritätsparameter einmal unbeachtet bleibt. Damit ein Prozeß

[83] **T**hread **B**ase **D**efinition = Faden-Basis-Definition

```
template< class Base>
class TBD{                                          // Thread Base Definition  TBD
  typedef teLinkage <Base> Link;
  typedef tePriority <Link, priority> Priority;
public:
  typedef Priority ThreadBase;
};

typedef typename TBD<theObject>::ThreadBase ThreadBase;
```

Abbildung 3.13: Konfiguration der planbaren Threadeinheit

überhaupt benutzt werden kann, ist es zwingend notwendig, daß mindestens die Basis *Base* vorhanden ist, welches die erste Möglichkeit darstellt. Die verbleibenden drei Kombinationen ergeben sich durch die beiden Klassen *teLinkage* und *tePriority*, wobei sie jeweils einzeln oder auch zusammen auftreten können.

Der Abbildung ist weiterhin zu entnehmen, daß der Typ ThreadBase ein weiteres Mal außerhalb von *TBD* eine Definition erfährt, wobei als *Template*-Parameter für die Klasse *TBD* die Basisklasse des Prozesses eingesetzt wird. Es handelt sich um die Basisprozeßabstraktion *theObject* aus der Prozeßbibliothek TAL.

3.2.2.2 Konfiguration des Planers

Nachdem die planbare Einheit konfiguriert ist, muß als nächster logischer Schritt die Zusammenstellung des Planungswesens erfolgen. Dem gleichen Schema folgend ist die Definition der einzelnen Bestandteile des Planers ebenfalls mit Hilfe von *Templates* vorgenommen.

Aus den Abschnitten 3.1.3.2 und 3.1.3.4 über die Klassenentwürfe des Planungswesen sowie der Wartelistenabstraktion sind die in der Abbildung 3.14 ersichtlichen Namenskürzel der konfigurierbaren Einheiten bereits bekannt. Die zu erkennenden Auslassungen in den Namen, durch `XXXX` dargestellt, beschreiben dabei die Austauschbarkeit der Einheit durch schnittstellenkompatible Varianten. Dem Klassendiagramm Abbildung 3.10 können die verschiedenen Ausprägungen entnommen werden. Weiterhin hat jeder Bestandteil seine spezielle Typenbezeichnung (Store, Idler, ...), wodurch auch das Schichtenmodell der funktionalen Einheiten wiederzuerkennen ist.

Eine kleine Besonderheit stellt die Klasse *sadXXXXSpin* dar, weil sie einen *Template*-Parameter *XXXXFuse* erhält. Dieser ist maßgeblich für das Funktionieren des gesamten Systems von Bedeutung, da im Bereich der `Leerlaufkontrolle` die Synchronisierungsmechanismen benötigt werden, falls diese überhaupt eine Benutzung erfahren. Denn für den Fall, daß ein Prozeß blockieren möchte und in der Warteliste kein weiterer steht, muß die Synchronisierung aufgehoben

```
template< class TBD>
class SD{                                          // Scheduler Definition     SD
  typedef sadXXXX <TBD> Store;
  typedef sadXXXXSpin <Store, XXXXFuse, TBD> Idler;
  typedef sadUrgencyXXXXPriority <Idler, TBD> Urgency;
  typedef sadPrimacyXXXXPriority <Urgency, TBD> Primacy;
public:
  typedef sadScheduler <Primacy, TBD> Scheduler;
};

typedef typename SD<ThreadBase>::Scheduler Scheduler;
```

Abbildung 3.14: Konfiguration des Planungswesens

werden. Ansonsten kann niemals wieder ein Prozeß bereit gesetzt werden. Damit das gesamte System mit den selben Synchronisationsmechanismen arbeitet, muß *XXXXFuse* gleichermaßen im erweiterten Prozeß definiert werden.

Als Schlußpunkt steht die Zuordnung der Schedulerdefinition (*SD*) zu dem verwendbaren Typen Scheduler, wobei der Typ der planbaren Prozeßbasis ThreadBase als *Template*-Parameter Verwendung findet. Dadurch ist gewährleistet, daß sämtliche Abstraktionen innerhalb der Planerdefinition mit ein und derselben Prozeßbasis arbeiten, womit die Typenverträglichkeit im System garantiert wird.

3.2.2.3 Konfiguration zum erweiterten Thread

Im dritten Schritt der Konfiguration erfolgt der Aufbau des erweiterten Prozeßfadens. Er kann nun an dieser Stelle klar bestimmt werden, da erst zu diesem Zeitpunkt die tatsächliche Ausprägung des Planers bekannt ist. Denn durch den *teAdapter* wird der Prozeßfaden mit den Planungswesen verbunden, weshalb er natürlich auch den genauen Typ des Schedulers kennen muß.

Die Abbildung 3.15 beschreibt die Konfiguration auf programmtechnischer Ebene. Es zeigt sich wiederum das bekannte Bild, daß eine Spezifikation der Abstraktion mit Hilfe von Schablonen erbracht wird. Dabei stellt die Klasse *TED*[84] den Rahmen für die Deklaration bereit, welche ihrerseits zwei *Template*-Parameter bekommt. Zunächst ist die Basisabstraktion TBD des Prozesses zu nennen und als zweites die Beschreibung SD des Schedulers.

Dem Aufbau des Klassendesign folgend wird die Konfiguration ebenfalls Schicht für Schicht etabliert. Somit bekommt der *teAdapter* als erster Baustein die beiden grundlegenden Parameter TBD und SD zugewiesen. Daran ansetzend werden die weiteren benötigten Klassen hinzugefügt, wobei noch einmal betont

[84] **T**hread **E**xtended **D**efinition = Fadenerweiterungsdefinition

```
template< class TBD, class SD>
class TED{                                          // Thread Extended Definition  TED
public:
  typedef teAdapter <TBD, SD> Adapter;
  typedef teCloture <Adapter, XXXXLocker> Cloture;
  typedef teMonitor <Cloture, XXXXFuse> Monitor;
  typedef teWatchdog <Monitor> Watchdog;
  typedef teInterface <Monitor> Interface;
};

typedef typename TED<ThreadBase, Scheduler>::Interface Thread;
```

Abbildung 3.15: Konfiguration zum erweiterten Thread

sei, daß es sich hier immer um Vererbungsbeziehungen zum vorhergehenden Bestandteil handelt.

Herauszunehmen ist als wichtiger Teil der Konfiguration das Modul *teMonitor*, welches für die Synchronisation von Prozessen zuständig ist. Der zweite Parameter *XXXXFuse* beschreibt hierbei die Vorgehensweise bzw. die Strategie der Synchronisation. Als grundlegend muß hier angesehen werden, daß dieser *Template*-Parameter in Übereinstimmung mit dem Planer gleich zu setzen ist, damit das bereits angedeutete Stillstehen des Systems nicht vorkommen kann. Um die Situation ausschließen zu können, wird im Abschnitt 3.2.2.4 eine Möglichkeit der Vereinheitlichung des gesamten Konfigurationsvorganges aufgezeigt.

Abschließend ist zu erwähnen, daß die Teilabstraktionen des Prozeßbildes oberhalb der Klasse *teAdapter* kombinierbar sind. Die sich daraus ergebenen Möglichkeiten sind sehr komplex und vielfältig (24 Varianten), weshalb sie hier nicht alle aufgezählt werden. Es können beliebige Kombinationen der Klassen sein, darunter auch Auslassungen in dem Ableitungsschema. Jedoch sollte die hier vorgeschlagene Reihenfolge aus Gründen der korrekten Funktionweise des Systems sowie aus Effizenzgründen eingehalten werden.

3.2.2.4 Konfiguration des gesamten Komplexes

Aus den bereits erläuterten Gründen über gleiche Parameter bei der Synchronisation in beiden Abstraktionen SD und TED wird die Konfiguration des gesamten Komplexes noch einmal durch eine umschließende Klasse gekapselt. Dadurch ist es mit Hilfe eines globalen *Template*-Parameters (*Secure*) möglich, für alle eingebetteten Klassen die Synchronisationsmethode gleichermaßen festzulegen. Hiermit ist sichergestellt, daß ein Stillstand im System, verursacht durch falsches Synchronisationsmanagement, nicht auftreten kann.

Die Abbildung 3.16 zeigt den gesamten Konfigurationsprozeß, wobei nur die wichtigen Details dargestellt sind, um den Blick auf das Wesentliche zu lenken. Das Bild zeigt die Definition des Prozeßfadens (*Thread*) und des Planers

```
template<class Secure>
class conf{
  template< class Base>
  class TBD{                                          // Thread Base Definition   TBD
    ...
  };
  typedef typename TBD<theObject>::ThreadBase ThreadBase;

  template< class TBD>
  class SD{                                           // Scheduler Definition     SD
    ...
    typedef sadXXXXSpin <Store, Secure, TBD> Idler;
    ...
  };
  typedef typename SD<ThreadBase>::Scheduler Scheduler;

  template< class TBD, class SD>
  class TED{                                          // Thread Extended Definition   TED
    ...
    typedef teMonitor <Cloture, Secure> Monitor;
    ...
  };

public:
  typedef typename TED<ThreadBase, Scheduler>::Interface Thread;
  typedef typename TED<ThreadBase, Scheduler>::Cloture Schemer;
};

typedef conf<XXXXFuse> Mode;

typedef Mode::Thread Thread;
typedef Mode::Schemer Schemer;
```

Abbildung 3.16: allgemeine Konfigurationsbeschreibung

(*Schemer*) innerhalb der Klasse *conf*. Der *Schemer* stellt die Schedulerfunktionalität ohne Synchronisationsmechanismen bereit. Verwendet wird diese Abstraktion im Bereich der Unterbrechungsanbindung des Planers. Im Falle einer Synchronisierung mit Hilfe des Unterbindens der Unterbrechungen spielt diese keine Rolle. Jedoch muß bei der unterbrechungstransparenten Methode *GuardFuse* der Zustand der Synchronisation bzw. der Zustand des kritischen Abschnittes abgefragt und entsprechend dieses Zustandes gehandelt werden, weil ein mehrfaches Betretten eines kritischen Abschnittes nicht erlaubt ist.

Im Anschluß daran ist die Deklaration des Verfahren Mode zu sehen, wobei als globaler *Template*-Parameter *Secure* eine Klasse vom Typ XXXXFuse benötigt wird. Welche Möglichkeiten sich hier anbieten, ist der Abbildung 3.11 zu entnehmen. Wird ein spezielles Verfahren konfiguriert, ist der Name Mode durch den Namen des Plaungsverfahrens z.B. LCFS zu ersetzen, damit eine eindeutige Beschreibung gewährleistet ist.

Nachdem die Konfiguration vollständig abgeschlossen ist, steht der Typ der Prozeßfäden für die Anwendung fest. Die Prozeßfäden sind nun von dem Typ

Thread abzuleiten, damit sie durch das zusammengestellte Verfahren geplant werden können. Weiterhin ist für die Anbindung des Planers an die Unterbrechungsmechanismen des Systems gesorgt, weil der Typ Schemer ebenfalls verfügbar ist.

Die gesamte Konfigurationsbeschreibung muß dem System global zugänglich sein, damit in allen Teilbereichen mit den selben Abstraktionen gearbeitet wird. Hierdurch ist die ordnungsgemäße Funktionsweise sichergestellt.

3.2.3 Szenario einfacher LCFS-Scheduler

Das allgemeine Vorgehen beim Konfigurieren wurde durch den vorherigen Abschnitt deutlich herausgearbeitet. Nun wird die Konfiguration eines einfachen LCFS-Planers gezeigt, wobei die oben beschriebene Herangehensweise verwendet wird.

Von der Anwendung ausgehend muß zunächst einmal geklärt werden, welche Anforderungen diese an das System oder besser an die Planung der Prozeßfäden stellt. Der Anwendungsprogrammierer bzw. der Konfigurator wählt dann anhand einer Anforderungsbeschreibung der Applikation die Eigenschaften aus der Merkmalsbeschreibung des Planungswesen, welche für die Umsetzung die geeignetsten Methoden bereitstellen. Danach erfolgt eine Abbildung der gewonnenen Merkmalsauswahl hin zu den Klassen, die die Eigenschaften entsprechend implementieren. Darauf aufbauend werden die Klassen zu der schon beschriebenen allgemeinen Konfigurationsbeschreibung zusammengefaßt, wodurch ein System entsteht, welches die erwarteten Eigenschaften zeigt.

Die Abbildung von *Merkmalen* zu *Klassen* wieter zur *Konfigurationsbeschreibung* übernimmt im Moment der Konfigurator noch selbst, jedoch könnte auch ein Werkzeug diese Aufgabe übernehmen, wodurch eine Automatisierung der Konfiguration möglich wäre.

In dem vorliegenden Beispielszenario eines einfachen LCFS-Planers sind die gestellten Anforderungen der Anwendung an das System in der folgenden Aufzählung zusammengefaßt, wobei die ausgewählten Eigenschaften aus den Merkmalsdiagrammen hinter den Anforderungen in Klammern vermerkt sind:

- mehrere Prozeßfäden (*multi*), nicht unterbrechbar
    1. kein Leerlaufprozeß, sondern Leerlaufschleife (*idle loop*)
    2. Planung mit Hilfe einer Strategie (*scheduling*, *strategie*, *based on*)
        (a) Ankunftszeitbasiertes Verfahren LCFS (*arrival time*, *LCFS*)
    3. Prozeßanzahl unbegrenzt (*process count*, *unlimited*)

Da es sich bei dem zu generierenden Planer um eine sehr einfache Variante des Planungswesens handelt, sind demzufolge nicht viele Eigenschaften aus den Merkmalsdiagrammen ausgewählt worden. Die verwendeten Klassen und die sich ergebene Konfigurationsbeschreibung ist dennoch sehr umfangreich, da das gesamte System sehr modular aufgebaut ist.

```
template <class Secure>
class conf
{
  template< class Base>
  class TBD{                                          // Thread Base Definition  TBD
    typedef teLinkage <Base> Link;
  public:
    typedef Link ThreadBase;
  };
  typedef typename TBD<theObject>::ThreadBase ThreadBase;

  template< class TBD>
  class SD{                                           // Scheduler Definition  SD
    typedef sadLIFOStack <TBD> Store;
    typedef sadSpin <Store, Secure, TBD> Idler;
    typedef sadUrgencyFCFSNonePriority <Idler, TBD> Urgency;
    typedef sadPrimacyNever <Urgency, TBD> Primacy;
  public:
    typedef sadScheduler <Primacy, TBD> Scheduler;
  };
  typedef typename SD<ThreadBase>::Scheduler Scheduler;

  template< class TBD, class SD>
  class TED{                                          // Thread Extended Definition  TED
  public:
    typedef teAdapter <TBD, SD> Adapter;
    typedef teInterface <Adapter> Interface;
  };
public:
  typedef typename TED<ThreadBase, Scheduler>::Interface Thread;
  typedef typename TED<ThreadBase, Scheduler>::Adapter Schemer;
};
typedef conf<VoidFuse> LCFS;

typedef LCFS::Thread Thread;
typedef LCFS::Schemer Schemer;
```

Abbildung 3.17: Konfigurationsbeschreibung eines LCFS-Planers

Aus der Analyse der Anforderungen ist zu erkennen, daß mehrere Prozeßfäden gleichzeitig im System vorhanden sein können, weshalb eine Prozeßbibliothek (TAL) die Voraussetzung für den Mehrprozeßbetrieb bildet. Weiterhin muß eine Verkettungseigenschaft vorhanden sein, damit wartende Prozesse in einer Struktur des Speicherwesens aufbewart werden können. Die Notwendigkeit eine Priorität mitzuführen, besteht laut Anforderungsbeschreibung nicht. Somit ist die ProzeßBasis TBD bereits vollständig beschrieben. Die Abbildung 3.17 zeigt neben der Definition der Basisprozeßfäden ebenfalls schon die Definition des Planers sowie die Beschreibung der erweiterten Prozeßfäden. Zu erkennen ist, daß unter dem Typennamen *ThreadBase* die Definition eines Prozeßfadens mit Verkettungseigenschaft zur Verfügung steht.

Der Beschreibung folgend schließt sich die Konfiguration des Planers beginnend mit dem Modul für die Speicherung wartender Prozesse an. Es erfolgt die

Auswahl einer Klasse aus dem Bereich der Speicherstrategien für Prozesse, wobei die am besten passendste Variante eine Benutzung erfährt. Für den vorliegenden Fall stellt die Klasse *sadLIFOStack* die geeigneteste Version dar, da mit minimalem Aufwand die Anforderungen auf dem Gebiet der `Wartelistenverwaltung` erbracht werden.

Aufbauend auf dieser Basis werden nun sukzessiv die weiteren Schichten des Planers zusammengefügt, so wie es im Klassenmodell und der Konfigurationsbeschreibung bereits beschrieben wurde. Als nächstes ist somit die Definition der `Leerlaufsteuerung` vorzunehmen. Die Anforderung schreibt hier die Benutzung einer *idle loop* vor, welche durch die Klasse *sadSpin* implementiert ist.

Da das System keine unterbrechbaren Prozesse benötigt, darf auch beim Bereitwerden und Setzen von Prozessen (Methode `shove()`) kein Kontextwechsel vorgenommen werden, obwohl die Strategie LCFS heißt. Aus diesem Grund ist der erste Teil der `Aktivierungsentscheidungsschicht` mit der Klasse *sadUrgencyFCFSNonePriority* konfiguriert, damit der momentan laufende Prozeß nie verdrängt wird. Hier zeigt sich, daß der Planer diese Konfigurationsmöglichkeit bietet, obgleich auch die Prozeßerweiterungen die Unteilbarkeit von Prozessen mit Hilfe der Klasse *teCloture* sicherstellen könnte. Da dieses der Planer bewerkstelligt, muß keine zusätzliche Schicht im Prozeß vorhanden sein und ein minimaleres System kann erzeugt werden. Ebenfalls ist dem Namen der gewählten Klasse zu entnehmen, daß keine Prioritäten im System Verwendung finden.

Der zweite Teil der Entscheidungschicht innerhalb des Schedulers betrifft die Konfiguration der Funktion `pause()`. Für die Konfiguration stehen drei Klassen zur Verfügung, jedoch kommt nur eine für den gegebenen Sachverhalt in Frage. Wie schon beschrieben, arbeitet das System ohne Prioritäten und deshalb darf nur die Klasse *sadPrimacyNonePriority* angewendet werden. Den Abschluß der Planer-Zusammenstellung bildet der allgemeine *sadScheduler*, der verbunden mit den Basisabstraktionen die gefragten Fähigkeiten zur Verfügung stellt. Dieser wird nur noch mit der planbaren Einheit *ThreadBase* versorgt, um die Typkonformität herzustellen.

Nachdem die Prozeßbasis und der Planer vollständig konfiguriert sind, muß nun die Definition des erweiterten Prozeßmodells stattfinden. Da keine zusätzlichen Anforderungen an die Prozeßfäden gerichtet werden, erfolgt nur das Zusammenbringen des Planers mit den Prozeßfäden unter Nutzung der Klasse *teAdapter* sowie das Aufstecken des *teInterface*, damit eine einheitliche Schnittstelle vorliegt.

Der weitere Konfigurationsprozeß gestaltet sich wie bereits erklärt mit dem Nach-Außen-Führen der Typen und der Kennzeichnung der eigentlichen Strategiebezeichnung. Die Anwendung kann nun ihre Aufgaben in Klassen modellieren, die von der Klasse *Thread* abgeleitet sind, wodurch die gesamte Funktionalität des Planungswesens Teil der Anwendungsfäden wird.

3.2.4 Szenario zeitscheiben- und prioritätsbasiertes FCFS

Ebenfalls wie beim vorherigen Beispiel liefert die Anwendung eine Anforderungsbeschreibung, mit deren Hilfe die Auswahl der Merkmal erfolgt. Da der Vorgang der Auswahl sowie der Abbildung von *Merkmale-Klassen-Konfigurationsbeschreibung* bereits erläutert wurde, wird dies als bekannt vorausgesetzt. Deshalb soll umgehend damit begonnen werden, welche Anforderungen die Applikation an das Planungswesen stellt. In der folgenden Aufzählung sind die Anforderung und wiederum dahinter in Klammern die Merkmale aufgelistet:

- mehrere Prozeßfäden (*multi*)
  1. kein Leerlaufprozeß, sondern Leerlaufschleife (*idle loop*)
  2. Planung mit Hilfe einer Strategie (*scheduling*, *strategie*)
     (a) unterbrechbare Prozesse (*preemptive*)
     (b) Strategie basiert auf Ankunftszeit (*based on*, *arrival time*)
        i. Verfahren FCFS (*FCFS*)
        ii. statische Prioritäten (*priority*, *static*)
        iii. 2^{32} Prioritäten (*quantum*)
        iv. plane gleiche Prioritäten (*schedule same*)
     (c) faire Zeitscheibe (*timeslice*, *fair*)
  3. Prozeßanzahl unbegrenzt (*process count*, *unlimited*)

Durch die umfangreichere Anforderungsbeschreibung gekennzeichnet, handelt es sich hierbei um ein aufwendigeres Verfahren. Einige bekannte Merkmale sind aber dennoch wiederzuerkennen. So ist es nicht verwunderlich, daß die Applikation mit mehreren Prozeßfäden arbeiten muß und weiterhin die Merkmale der *strategie*, *based on*, *arrival time idle loop* sowie *process count* und *unlimited* trägt. Bei der Ankunftszeitorientierung wurde ein Tausch von *LCFS* hin zu *FCFS* vorgenommen und als zusätzliche Eigenschaften sind *preemptive*, *priority*, *static*, *quantum*, *schedule same*, *timeslice* und *fair* hinzugekommen.

Die im einfachen Beispiel verwendeten Merkmale sowie deren Abbildung auf Klassen ist mit der Ausnahme von *idle loop* völlig gleich gehalten. Beim Merkmal der *idle loop* muß hingegen eine spezielle Variante aus Gründen der Prioritätensteuerung eingesetzt werden, welche den Status des Systems vermerkt und liefern kann. Die verwendete Klassenabstraktion ist *sadStatusSpin*.

Die Prozeßbasis besteht aus der Verkettungseigenschaft *teLinkage* und einer Prioritätsklasse *GeneralValence*, welche den *Template*-Parameter der *tePriority* bildet. Die Priorität *GeneralValence* erhält ebenfalls einen Parameter, der die Anzahl der verschiedenen Prioritäten angibt. Für den hier zu konfigurierenden Planer müssen 2^{32} Werte möglich sein. Daher wird als Parameter ein *long* gewählt.

```
template < class Secure>
class conf
{
  template< class Base>
  class TBD{                                          // Thread Base Definition  TBD
    typedef teLinkage<Base> Link;
    typedef GeneralValence<long> Prio;
  public:
    typedef tePriority<Link, Prio> ThreadBase;
  };
  typedef typename TBD<theObject>::ThreadBase ThreadBase;

  template< class TBD>
  class SD{                                           // Scheduler Definition  SD
    typedef sadFIFOSwitch<TBD> Switch;
    typedef sadValenceQueue <TBD, Switch> Store;
    typedef sadStatusSpin <Store, Secure, TBD> Idler;
    typedef sadUrgencyHigherPriority <Idler, TBD> Urgency;
    typedef sadPrimacySamePriority <Urgency, TBD> Primacy;
  public:
    typedef sadScheduler <Primacy, TBD> Scheduler;
  };
  typedef typename SD<ThreadBase>::Scheduler Scheduler;

  template< class TBD, class SD>
  class TED{                                      // Thread Extended Definition  TED
  public:
    typedef teAdapter <TBD, SD> Adapter;
    typedef teMonitor <Adapter, Secure> Monitor;
    typedef teInterface <Monitor> Interface;
  };
public:
  typedef typename TED<ThreadBase, Scheduler>::Interface Thread;
  typedef typename TED<ThreadBase, Scheduler>::Adapter Schemer;
};
typedef conf<ChipFuse> FCFS_RR_Same_Priority;

typedef FCFS_RR_Same_Priority::Thread Thread;
typedef FCFS_RR_Same_Priority::Schemer Schemer;
```

Abbildung 3.18: Konfigurationsbeschreibung eines FCFS-Planers

Mit der Definition des Typen ThreadBase (siehe Abbildung 3.18) ist die Konfiguration der planbaren Basisabstraktion abgeschlossen.

Sich daran anschließend wird der Planer selbst Gegenstand der Betrachtung bzw. Konfiguration. Die Beschreibung ist dem allgemeinen Konfigurationsmodell folgend sehr ähnlich, jedoch finden in einigen Bereichen andere Abstraktion Anwendung. Beginnend mit dem Teilgebiet der Speicherung wartender Prozesse muß hier eine Klasse gewählt werden, die das Sortieren nach Prioritäten unterstützt. Als Klasse kommt in diesem Fall *sadValenceList* zum Einsatz, wobei diese zusätzlich noch den Parameter der Reihenfolge erhält. Da das System nach FCFS arbeiten soll, muß die Folge in der Warteliste ebenso konfiguriert sein. Daher bietet sich die Klasse *sadFIFOSwitch* an.

Der nächste Punkt in der Konfiguration ist die Bereitstellung der Leerlaufsteuerung. Schon erwähnt wurde, daß hier ein spezielle Variante *sadStatusSpin* zum Einsatz gebracht wird. Der Grund für diese Maßnahme stellt die Verwendung der Prioritäten dar, weil sonst Prioritätsinversionen vorkommen können. Infolgedessen würde das System nicht mehr korrekt funktionieren.

Der weiteren Vorgehensweise folgend schließt sich die Bereitstellung der `Aktivierungsebene` an. Zu dieser zählen die beiden Typen Urgency und Primacy. Die Anforderung der Anwendung an das System besagen ein Verdrängen des aktiven Prozesses, wenn der aufkommende Prozeß eine höher Priorität aufweist als der laufende. Demzufolge muß für den Typ Urgency die Klasse *sadUrgencyHigherPirority* verwendet werden. Durch das Merkmal *schedule same* bestimmt, ist für den zweiten Teil der Aktivierungebene die Klasse *sadPrimacySamePriority* zu wählen. Den Abschluß bildet das Zusammenbringen mit dem allgemeinen Planer *sadScheduler* und der anschließenden Definition des Scheduler mit dem *Template*-Parameter *ThreadBase* als planbare Prozeßeinheit.

Nachdem die Prozeßbasis und der Planer vollständig zusammengefügt wurden, muß die Definition des erweiterten Prozeßfadens erfolgen. Den Anfang macht wie immer die Klasse *teAdapter*, die die Verbindung der Prozeßfäden zum Planer herstellt. Somit arbeiten alle von dieser Abstraktion abgeleiteten Prozesse mit ein und dem selben Scheduler. Die Anforderung der Anwendung war die Benutzung einer Zeitscheibe im System, um einen besseren Mehrprozeßbetrieb sowie damit Verbunden das gleichzeitige Voranschreiten der Aufgabenerbringung zu gewährleisten. Da aber durch den Mechanismus der Zeitscheibe ein nebenläufiges Ausführen von Planeraktivitäten nicht ausgeschlossen werden kann, muß eine Sicherungsschicht Bestandteil der Prozeßfäden sein. Um diese Aktionen serialisieren zu können, wird deshalb die Klasse *teMonitor* als Zwischenschicht dazugeschaltet. Sie erhält als Parameter die Art der Sicherung, welche als globaler *Template*-Parameter der gesamten Konfigurationsbeschreibung zugänglich ist. Die oberste und damit abschließende Ebene bildet das interface *teInterface*, welches die allgemeine und bereits bekannte Schnittstelle definiert.

Als letzter und finaler Schritt wird die Deklaration der Strategie vorgenommen, wobei die Konfigurationsbeschreibung *conf* den Sicherungsparameter *ChipFuse* erhält. Dadurch stehen die beiden Typen Thread und Schemer der Anwendung zur Verfügung, wobei die Prozeßfäden von Thread erben und die Abstraktion Schemer ihre Anwendung im Bereich Unterbrechungsanbindung des Planers finden wird. Es stehen im Schemer keine Synchronisierungsmaßnahmen bereit, weil eine Unterbrechung nur dann auftreten können, wenn momentan kein Prozeß im Planer residiert.

Kapitel 4

Ergebnisse

Dieses Kapitel beschäftigt sich mit der Bewertung und Diskussion der erreichten Ergebnisse. Ein besonderer Schwerpunkt liegt hierbei auf den gemessenen Systemgrößen. Zu diesen Größen zählen der Ressourcenverbrauch sowie die Laufzeit von Funktionen in den verschiedenen Konfigurationen. Weiterhin ist zu untersuchen, inwieweit die *Template*-Programmierung Einfluß auf die Codegröße des Systems und damit verbunden auf den Ressourcenverbrauch hat.

4.1 Größenmessungen

4.1.1 Die Messanordnung

Das Messen der Größen von verschiedenen Programmen bzw. unterschiedlichen Konfigurationen muß immer unter den gleichen Bedingungen erfolgen, um verlässliche und vergleichbare Ergebnisse zu erhalten. Deshalb sind vor Beginn der Messung die Parameter für die Messung festzulegen und diese in einer Messanordnung klar zu definieren.

Als erstes muß somit geklärt werden, mit welchem Compiler die Übersetzungen des Systems durchgeführt werden, sowie mit Hilfe welcher Optimierungseinstellungen dieser seine Arbeit verrichtet. Dadurch ist sichergestellt, daß nicht etwa Optimierungseinstellungen oder Versionsunterschiede beim Übersetzer ausschlaggebend für Größenveränderungen sind. Der verwendete Compiler ist der `g++` in der Version `2.96` aus dem GNU-Projekt[1]. Die Optimierungseinstellung ist auf `-O6` eingestellt, welches die höchste Einstellung darstellt.

Der zweite Punkt der Messanordnung betrifft das auszumessende Programm. Es werden immer ausführbare Programme erzeugt, die auf einem nativen System mit einem x86 Prozessor lauffähig sind. Als Vergleichsbasis wird zuerst ein Programm mit leerer Hauptfunktion sowie ein Programm mit einem TAL-Prozeß

[1] Der Name des GNU-Projektes leitet sich von dem rekursiven Akronym „GNU's Not Unix“, also „GNU ist nicht Unix“ ab.

System	Größen in Byte			
	Text	Data	BSS	Total
leeres `main()`	144	0	0	144
TAL-Prozeß	244	0	0	244

Tabelle 4.1: Speicherverbrauch eines Programms mit einem TAL-Prozeß

ausgemessen. Das zweite Testprogramm verwendet für den Prozeß den initialen Stack des Systems. Durch diese Vorgehensweise ergibt sich die Möglichkeit, den durch die verschiedenen Planer ins System eingebrachten zusätzlichen Code abzuschätzen.

Für das Ausmessen der Planergrößen wird ebenfalls jeweils ein Prozeß (verwendet Bootstack) erzeugt, der sich nach dem Start sofort blockiert. Infolgedessen befindet sich das System im Leerlauf. Die restlichen Planerfunktionalitäten werden ebenfalls referenziert, wodurch sie im ausführbaren Programmcode vorhanden sind.

4.1.2 Ressourcenverbrauch

Wie bereits beschrieben, wurden zwei Programme ausgemessen. In der Tabelle 4.1 sind die Codegrößen sowie die Anzahl der benötigten Daten aufgezeigt. Der Größenunterschied vom leeren Hauptprogramm zum Programm mit einem TAL-Prozeß fällt recht gering aus, wenn bedacht wird, daß die Funktionalität für Kontextwechselvorgänge sowie Prozeßerzeugungen bereits enthalten ist. Daten werden in der momentanen Konfiguration des System noch nicht gebraucht, da wie schon bekannt der Bootstack weiterbenutzt wird. Die Werte der Tabelle 4.1 sind als Vergleichsbasis zu denen in der Tabelle 4.2 zu sehen, weil dadurch zu erkennen ist, wie viel der Einsatz die Funktionalität der unterschiedlichen Planer kostet.

Die Tabelle 4.2 stellt die erreichten Größen für die verschiedenen Planungsstrategien zusammen dar. Mit dem kleinsten System beginnend ist absteigend eine Zunahme der Größe hin zum größten sichtbar. Abhängig von der Strategie bzw. den Aufwand den diese auf programmiertechnischer Ebene verlangt, ist der Ressourcenverbrauch unterschiedlich hoch. Wie demzufolge nicht anders zu erwarten war, stellt die Variante `LCFS` das kleinste System und das Verfahren `MLFB` das größte System dar. Dies zeigt sehr anschaulich, wie das System je nach Anforderung skaliert.

Weiterhin ist zu erkennen, daß alle Verfahren einen ähnlichen Datenaufwand erfordern. Eine Ausnahme bildet hierbei nur das FCFS-Verfahren, welches statt den sonst üblichen 12 Byte Daten 16 Byte braucht. Dabei setzt sich der Datenbereich folgendermaßen zusammen. Der *life*-Zeiger, über den der momentan aktive Prozeß identifiziert wird, benötigt 4 Byte und die verbleibenden Daten

Strategie	Größen in Byte			
	Text	Data	BSS	Total
LCFS	348	0	12	360
SPN	436	0	12	448
FCFS	480	0	16	496
RM	500	0	12	512
SRTF	500	0	12	512
EDF	532	0	12	544
LLF	732	0	12	744
MLFB	792	0	12	804

Tabelle 4.2: Code- und Datengrößen der verschiedenen Planer

sind auf die Planereinheit zurückzuführen. Die 4 Byte Daten zusätzlich beim FCFS-Verfahren resultieren aus der Konfiguration der `Wartelistenverwaltung` (Klasse *sadFIFOQueue*), weil diese einen Zeiger mehr verwendet als die anderen Speicherstrategien. Weitere Daten werden in dieser einfachen Konfiguration nicht gebraucht.

Die gemessenen Größen zeigen eindrucksvoll die Skalierbarkeit des entwickelten Softwaresystems. Von einer minimalen Basis ausgehend wird durch das Hinzufügen weiterer Ebenen bzw. Softwarebestandteilen neue Funktionalität erbracht. Dafür sind nur die Kosten zu tragen, welche tatsächlich mit der Funktionsweise zu tun haben. Durch die feingranulare Klassenstruktur und der damit verbundenen skalierbaren Konfiguration entstehen jeweils Systeme, die den Anforderungen der Anwendung genügen, aber darüberhinaus keine zusätzlichen Kosten verursachen.

Durch diese Herangehensweise entstehen minimale Systeme, die selbst in dem Bereich der tiefsten eingebetteten Systeme einsetzbar sind. Wenn die oben dargestellten Zahlen einmal einer genaueren Betrachtung unterzogen werden, ist festzustellen, daß es sich jeweils um ein komplettes System mit einem Prozeß handelt. Somit braucht nur noch der Anwendungscode geschrieben und dem System hinzugefügt werden. Wird von einer Ausstattung des Zielsystems von etwa 4kB ROM ausgegangen, verbleibt für die Erbringung der Anwendungsaufgaben jeweils mindestens 3kB. Im unter anderem anvisierten Bereich der ressourcenknappen Systeme ist der zur Verfügung stehende Platz von 3kB ROM durchaus ausreichend für die Entwicklung der Anwendung. Somit ist gezeigt, daß die Softwarefamilie Sad ein breites Spektrum an möglichen Einsatzgebieten abdecken kann.

Weiterhin ist zu untersuchen, wie der Mechanismus der *Template*-Programmierung die Codegröße des Systems beeinflußt, wenn nicht nur ein Prozeßfaden, sondern zahlreiche erzeugt werden.

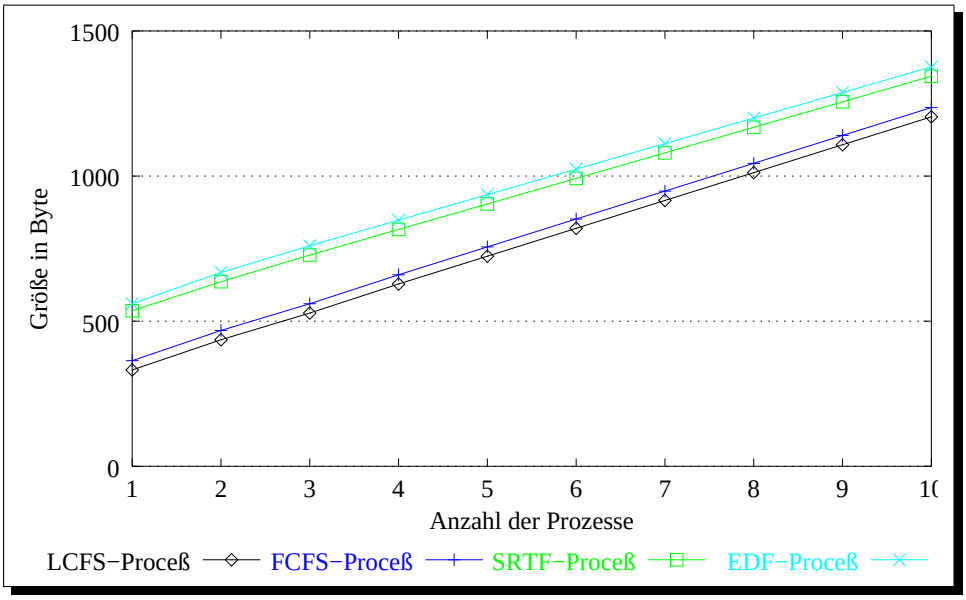


Abbildung 4.1: Auswirkungen der *Template*-Programmierung auf die Codegröße

4.1.3 Auswirkungen der *Template*-Programmierung

Das Programmieren mit Hilfe von *Templates* kann sich auch negativ auf die Systemgrößen auswirken, wenn nicht sorgfältig mit diesem Mechanismus umgegangen wird. Dies liegt in der Tatsache begründet, daß der Übersetzer den Code für die jeweilige *Template*-Variante erzeugt. Somit kann eine Codedopplung entstehen, wenn mehrere verschiedene *Template*-Typen eingesetzt werden. Hier ist nun zu untersuchen, wie sich das System in Bezug auf die Größenzunahme verhält, wenn mehr als ein Prozeß generiert wird.

In dem Diagramm 4.1 ist für einige Beispielkonfigurationen die Codezunahme in Bezug auf die Anzahl der erzeugten Prozesse aufgetragen. Zu sehen ist, daß es sich jeweils um einen linearen Anstieg handelt, der bei jedem Verfahren in etwa 100 Byte beträgt. Das bei allen Verfahren ein ähnlicher Anstieg zu verzeichnen ist, ist einfach zu erklären, da sich die Prozeßabstraktion relativ unabhängig vom Planer darstellt. Nur die durch den Planer benötigten Daten variieren, wobei der sonstige Aufbau der Abstraktion konstant bleibt.

Die Zunahme der Codegröße um jeweils 100 Byte läßt sich weiterhin damit erklären, daß pro zusätzlichem Prozeß eine eigene Klasse geschrieben wurde, die einen eigenen Konstruktor sowie eine leere Methode (für den Anwendungscode des Prozesses) enthält. Ebenfalls wurden die Funktionen des Planers aufgerufen, wodurch infolgedessen die Prozesse in der Warteschlange ausführbereit standen. So muß nur der Code der Anwendungsaufgaben in die entsprechenden Funktionen integriert werden, damit das anwendungsspezifische Verhalten der Prozesse vorzufinden ist. Durch das jeweilige Schreiben neuer Klassen für weitere Prozes-

se wurde gewährleistet, daß der Übersetzer den nicht bereits bestehenden Code wiederverwendet, welches zu verfälschten Ergebnissen geführt hätte. In diesem Fall würden die Werte das Optimierungsverhalten des Compilers wiederspiegeln, was ja nicht Gegenstand der Untersuchungen sein sollte.

Das Diagramm zeigt gleichermaßen den unterschiedlichen Aufwand des Planungsverfahrens, wenn jeweils der Schnittpunkt mit der y-Achse betrachtet wird. Hier ist die Funktionalität des Schedulers bereits enthalten, weshalb bei zusätzlichen Prozeßerzeugungen diese nicht weitere Male hinzukommt. Damit ist gezeigt, daß die Programmierung mit *Templates* nicht gezwungenermaßen zu großen Programmen führt, wenn dieser Mechanismus sinnvoll eingesetzt wird. Eine strategische Rolle spielt dabei die Klasse *teInterface*, welche eine Schnittstelle definiert und gleichzeitig die Inlininggrenze darstellt. So wird sichergestellt, daß der Planercode nicht Bestandteil eines jeden Prozesses wird, sondern nur einmal vorhanden ist und über Funktionsaufrufe zur Verfügung steht.

Noch einmal bezugnehmend auf die Abbildung sei angemerkt, daß nur eine relativ einfache Prozeßabstraktion Anwendung fand. Wenn weitere Schichten hinzukommen, wie beispielsweise die `Sychronisierung` oder die `Blockadensteuerung`, wird der Aufwand bzw. die Codegröße beim Schnittpunkt mit der y-Achse ansteigen, aber danach wird der oben beschriebene konstante Anstieg von etwa 100 Byte pro Prozeß bleiben.

4.2 Zeit- und Taktmessungen

4.2.1 Das Testsystem

Gleichermaßen wie im Abschnitt 4.1.1 müssen eine einheitliche Testumgebung sowie gleiche Konfigurationsbeschreibungen garantiert werden, um die Ergebnisse vergleichbar zu halten. Daher sind die in den weiteren Abschnitten aufgeführten Programme mit dem gleichen Compiler (GNU `g++` Version `2.96`) übersetzt und die bereits bei den Größenmessungen benutzten Beschreibungen wiederverwendet.

Die Zeitmessungen wurden auf einem Intel Pentium® mit 150MHz durchgeführt. Die Grundlage für die Messungen bildete das in [Int97] beschriebene Verfahren. In den nachstehend aufgeführten Tabellen sind die Anzahl der Takte für den Bearbeitungszeitraum der Funktionen angegeben. Wird die Zeit im Nanosekundenbreich benötigt, ist die Taktanzahl mit 6.6 zu multiplizieren.

4.2.2 Taktverbrauch der Verfahren

Um die Zeiten bzw. den Taktverbrauch der einzelnen Funktionen zu messen, mußten verschiedene Testprogramme geschrieben werden. Einerseits war darauf zu achten, ob Kontextwechsel während der Abarbeitung erfolgten und andererseits

sollten Messverfälschungen wie etwa Cacheeffekte ausgeglichen oder vermieden werden.

Um den Cacheeffekt ausschließen zu können, wurde die jeweilige Testsequenz im Programm einige Male hintereinander ausgeführt. In Folge dessen sind die Caches gefüllt und die Messung wird nicht durch die Performance des Hauptspeichers beeinflußt.

Beim zweiten Punkt der Kontextwechselfrage wurde zwischen den nicht präemptiven und den präemptiven Funktionen unterschieden. Beginnend mit der nicht präemptiven Funktion `ready()` war der Taktzähler vor und nach dem Aufruf auszulesen. Die Differenz der beiden erhaltenen Wert bildete das Ergebnis bzw. die Anzahl der verbrauchten Takte.

Fortführend wurde für die Funktionen `shove()`, `pause()`, `delay()` und `smash()` eine andere Herangehensweise nötig, weil diese einen Kontextwechsel durchführen können. Hier durfte nicht einfach vor und nach dem Aufruf der Taktzähler ausgelesen werden, vielmehr war dieser jeweils vor dem Betreten der Funktion sowie am Austrittspunkt nach dem Wechsel abzulesen. Die Differenz der wiederum erhaltenen Werte stellt den Taktverbrauch dar.

Zum Verständnis der Tabelleneinträge sei folgendes vorausgeschickt: In der Tabelle sind konstante Zahlen und auch Formeln für die Anzahl der benötigten Takte zu finden. Wie das Wort konstant bereits verdeutlicht, spiegelt der angegebene Wert das Maximum des Taktverbrauches wieder. Im Bereich der Formeln ist ein Parameter n vorhanden, welcher die Wartelistenlänge zum Zeitpunkt des Funktionsaufrufes bezeichnet. Ist dieser bekannt, kann die maximale Taktnutzung berechnet werden. Oftmals bedarf es jedoch nicht der maximalen Taktanzahl. Wenn beispielsweise der bis dato höchstpriorisierteste Prozeß verdrängt wird, muß dieser am Anfang der Warteliste einsortiert werden. Somit entfällt das Durchsuchen der Liste und die benötigte Zyklusanzahl ist kleiner als das errechnete Maximum.

Weiterhin ist noch anzumerken, daß ein einfacher Kontextwechsel von einem TAL-Prozeß zu einem anderen auf dem oben beschriebenen Testsystem 18 Taktzyklen verbraucht. In der Tabelle 4.3 ist in einigen Formeleinträgen eine abschließende 18 zu finden. Eben diese bezeichnet den Taktverbrauch des Kontextwechsels.

Nachdem einiges zu den Testszenarien und den Messungen gesagt wurde, schließt sich die Untersuchung und Einschätzung der erreichten Ergebnisse nahtlos an. Wie sich schon bei den Größenmessungen herausstellte, skaliert das System hier ebenfalls von den einfachen Verfahren hin zu den komplexeren Varianten. Dies ist aber auch natürlich, wenn der logischen Sicht folgend bedacht wird, daß kompliziertere Planungsstrategien einen höheren Rechenaufwand nach sich ziehen. Die erhöhten Kosten sind aber nur bei den Funktionen anzutreffen, die das Bereitsetzen (`ready()`), das schnellstmögliche Aktivieren (`shove()`) oder das Pausieren (`pause()`) verwirklichen. In all diesen Funktionen werden, wenn Prioritäten vorhanden sind, Sortierungsvorgänge durchgeführt. Dabei muß eine Liste

Strategie	SAD-Funktionen (Messwerte in Taktzyklen)				
	`ready()`	`shove()`	`pause()`	`delay()`	`smash()`
LCFS	4	22	39	36	25
FCFS	8	26	42	37	30
MLFB	$18 + n \cdot 15$	$23 + n \cdot 15 + 18$	$28 + n \cdot 15 + 18$	43	26
EDF	$18 + n \cdot 18$	$30 + n \cdot 18 + 18$	-	36	27
LLF	$18 + n \cdot 18$	$25 + n \cdot 18 + 18$	-	42	32
RM	$18 + n \cdot 15$	$23 + n \cdot 15 + 18$	-	36	26
SPN	$18 + n \cdot 15$	$18 + n \cdot 15$	-	36	26
SRTF	$18 + n \cdot 15$	$23 + n \cdot 15 + 18$	-	36	26

Tabelle 4.3: Taktverbrauch der Schedulerfunktionen der unterschiedlichen Planer

nach der Einfügeposition durchsucht werden, wodurch die Angabe der Laufzeit in Takten nur bei bekannter Listenlänge möglich ist. Hingegen stehen für die beiden Standardverfahren LCFS und FCFS Datenstrukturen zur Speicherung wartender Prozesse bereit, die Einfügevorgänge mit konstanter Taktanzahl realisieren.

Ein zweiter Augenmerk soll auf die Funktion `pause()` gerichtet werden, da in der Tabelle dort nicht für alle Verfahren eine Zahl oder Formel angegeben wurde. Dies begründet sich mit der Tatsache, daß einige Verfahren kein Pausieren von Prozessen unterstützen dürfen. Stünde in den Systemen eine `pause()`-Methode zur Verfügung und würde diese benutzt, hätten die Verfahren ein anderes als ihr vordefiniertes oder mit ihrem Namen assoziiertes Verhalten.

Als drittes soll auf die Funktionen eingegangen werden, die ein konstanter Taktverbrauch kennzeichnet. So haben die Methoden `delay()` und `smash()` in den verschiedenen Konfigurationen zwar unterschiedliche Laufzeiten, jedoch in Bezug auf das jeweilige Verfahren eine konstante Zyklusanzahl. Das ergibt sich aus dem Bereich der Speicherstrategien, die ihrerseits alle auf der gleichen Basis beruhen. Daher ist das Entfernen von Prozeßelementen, welches die Funktionen `delay()` und `smash()` ausführen, bei allen Strategien bis auf FCFS und LLF gleich. Bei FCFS wird eine Warteschlange benutzt, bei der der Endzeiger gegebenenfalls zu aktualisieren ist und bei LLF müssen die Prioritäten beim Entfernen angeglichen werden. Die sonst relativ kleinen Taktunterschiede von Verfahren zu Verfahren rühren aus den noch zusätzlich zu erbringenden Aufgaben.

Durch die gewonnenen Ergebnisse läßt sich zusammenfassend bemerken, daß die maximale Zyklusanzahl entweder konstant ist oder berechnet werden kann. Eine Voraussetzung hierfür ist jedoch die Kenntnis über die Länge der Warteliste der Prozesse. In dem Bereich der tiefsten eingebetteten Systeme und auch bei Echtzeitsystemen kann diese zumeist angegeben werden, weil dort die Anzahl der Prozesse im System eine feste Größe ist. Somit kann diese Größe als Parameter n

System	`pause()`-Test
SAD	61
QNX	407
VxWorks	749

Tabelle 4.4: Zeiten in μ-Sekunden

in die Formel eingesetzt und dadurch der maximale Zykluswert für die einzelnen Funktionen angegeben werden.

Um jedoch die Ergebnisse noch besser einschätzen zu können, muß ein Vergleich mit bereits existierenden Systemen gemacht werden. Als Vergleichssysteme wurden die beiden Echtzeitbetriebssysteme QNX und VxWorks herangezogen. Sie beide verarbeiten präemptive priorisierte Prozesse nach dem FCFS-Prinzip. Folglich mußte das SAD-System ebenfalls so konfiguriert werden.

Das Testszenario bestand in einem System mit zwei gleichprioren Prozessen, welche in einer Endlosschleife ständig die Methode `pause()` benutzten, wodurch eine Prozeßumschaltung initiiert wurde. Die Tabelle 4.4 zeigt die Ergebnisse der Tests, wobei die Werte der beiden kommerziellen System QNX und VxWorks aus [SDO02] entnommen und die dort dargestellten Zeiten in Takte umgerechnet wurden. Es zeigt sich, daß das SAD-System schnell die notwendigen Entscheidungen für Prozeßwechsel trifft und diese durchführt, als es die anderen Systeme bewerkstelligen. Woher diese enormen Unterschiede im Taktverbrauch herrühren kann nicht eindeutig geklärt werden. Unter der Annahme, daß die Systeme sehr ähnliche Prozeßverwaltungen durchführen, wozu das Einsortieren des abgebenden Prozesses und das Entnehmen des zustartenden Prozesses sowie das Initiieren des Prozeßwechsels zählt, sollten eigentlich ungefähr die gleiche Anzahl an Zyklen verbraucht werden. Ein möglicher Grund für die stark abweichenden Ergebnisse könnte der sehr effiziente Umschaltmechanismus der TAL-Bibliothek im Gegensatz zu dem der anderen Systeme sein. Es zeigt sich somit, daß das SAD-System in Verbindung mit der TAL-Bibliothek den Vergleich mit anderen auch kommerziellen Systemen nicht scheuen braucht, im Gegenteil sogar sehr gute vorhersagbare Ergebnisse liefert.

4.3 Die Beispielszenarien

Für die in dem Abschnitt Konfiguration (siehe 3.2) gezeigten Beispielkonfigurationen sollen hier ebenfalls die Größen der erzeugten Systeme angegeben werden.

Beginnend mit dem einfachen LCFS-Planer müssen zu der Testanwendung im System noch einige Anmerkungen getätigt werden. In dem Beispiel arbeitet eine Anwendung mit zwei Prozessen, wobei der erste den anderen Prozeß erzeugt, ihn bereit setzt und sich im nachhinein selber beendet. Der zweite Prozeß betritt

eine Endlosschleife, nachdem er durch den Planer aktiviert wurde. Der Anwendungscode kann beim ersten Prozeß nach dem Erzeugen und vor dem Beenden eingefügt werden und bei dem zweiten Prozeß an Stelle der Endlosschleife. Durch das nicht präemptive Arbeiten der Prozesse, erfolgt kein Kontextwechsel während dem Bereitsetzen des erzeugten Prozesses.

Die Werte in der Tabelle 4.5 sind folgendermaßen zu interpretieren: Die Spalte `Text` beschreibt die Größe des übersetzten Anwendungs- und Systemcodes, wobei in der Beispielanwendung nur ein sehr geringer Anteil an Anwendungscode vorhanden ist. Er beläuft sich nur auf wenige Byte für die Endlosschleife. In der Spalte `Data` sind die Daten für die global angelegten Objekte enthalten, unter denen sich auch ein sehr großzügig ausgelegter Stack mit 1024 Byte Ausdehnung befindet. Es wird nur ein Stack für den zu erzeugenden Prozeß benötigt, da der erste Prozeß den initialen Stack weiterverwendet. Die Gesamtgröße des Systems zeigt die Spalte `Total`.

System	Größen in Byte		
	Text	Data	Total
LCFS	392	1064	1456

Tabelle 4.5: Systemgröße des LCFS-Planers

Die Tabelle zeigt den minimalen erforderlichen Ressourcenverbrauch auf Systemseite mit dem Wert 392 Byte an. Wenn wiederum von einer Ausstattung des Zielsystems von 4kB ROM ausgegangen wird, verbleibt für die Implemtierung der Anwendungsaufgaben mehr als 3kB Speicher, selbst wenn weitere Prozesse angelegt werden sollen. Dem Abschnitt über die Auswirkungen der *Template*-Programmierung (siehe 4.1.3) ist zu entnehmen, daß es pro zusätzlichen Prozeß 100 Byte bedarf.

Nachdem das Beispiel eines einfachen Planers betrachtet wurde, soll mit dem zweiten Konfigurationsbeispiel einem zeitscheiben- und prioritätsbasiertem Scheduler (siehe 3.2.4) fortgesetzt werden. Dabei sind wiederum die ausgemessenen Größen zu untersuchen sowie zu bewerten.

Das Anwendungsszenario stellt sich bei dieser Konfiguration etwas anders dar. Der Unterschied zur vorhergehenden Beschreibung beim einfachen LCFS-Planer ist der Anwendungscode in dem Beispiel. Diesmal besitzen die Prozesse eine Priorität (beide Priorität 10), mit Hilfe derer die Schedulingentscheidung getroffen wird. Da das System gleiche Prioritäten umschaltet, wenn die Zeitscheibe abgelaufen ist, implementieren beide Prozesse eine Endlosschleife. Soll ein spezifischer Anwendungscode laufen, ist er im Austausch für die Endlosschleifen einzusetzen. Ansonsten handelt es sich um ein komplettes Anwendungsszenario.

Die Tabelle 4.6 zeigt zwei verschiedene Größen für das Endsystem auf. Der Grund hierfür liegt darin, daß die Anbindung der Zeitscheibe über die Unterbre-

System	Größen in Byte		
	Text	Data	Total
SAD+Anwendung	484	1064	1542
Unterbrechungsanbindung	1940	392	2338
Gesamt	2424	1456	3880

Tabelle 4.6: Systemgröße des FCFS-Planers

chungsanbindung von PURE erbracht wurde. Es soll so klar zum Ausdruck gebracht werden, welcher Größenanteil von der Anwendung und dem SAD-System im gesamten Endsystem stammt.

Ähnlich dem vorangegangenen Beispiel ist die Größe der Planereinheit im Zusammenhang mit der Anwendung in Bezug auf die erbrachte Funktionalität recht klein. Die Werte zeigen deutlich, daß das System selbst im tiefsten eingebetteten Bereich eingesetzt werden kann, da auch mit der Unterbrechungsanbindung die Grenze von 4kB noch nicht erreicht wird. Sicherlich bestehen noch Optimierungsmöglichkeitem im Bereich der Anbindung von Unterbrechungen, infolgedessen ebenfalls kleinere Systeme entstünden. Die Stacks sind mit 1024 Byte ebenfalls sehr groß dimensioniert.

4.4 Die dinierenden Philosophen

4.4.1 Erklärung

Bei dem Beispiel der dinierenden Philosophen handelt es sich um ein bekanntes Problem aus dem Bereich der Prozeßsynchronisation. Formuliert wurde es erstmal von Edsger W. Dijkstra in [Dij72], um von einem Prozeßsystem zu abstrahieren, in denen die Prozesse mehrere Ressourcen exklusiv benötigen.

Dabei sitzen fünf Philosophen an einem runden gedeckten Tisch. Jeder verbringt sein Leben abwechselnd mit denken und essen. Verspürt ein Philosoph Hunger, versucht er die beiden Gabeln links und rechts neben seinem Teller zu bekommen, um damit zu essen. Nachdem er gesättigt ist, legt er die Gabeln ab und verfällt wieder in tiefgründiges philosophisches Nachdenken.

4.4.2 Pure vs. Sad

Das Problem der dinierenden Philosophen hat an sich nichts mit dem eigentlichen Thema der Arbeit zu tun. Jedoch existiert für die Programmfamilie PURE eine Implementierung (Testprogramm) des beschrieben Sachverhaltes. Aus diesem Grund kann das Beispiel für einen Vergleichstest der beiden Programmsysteme PURE und SAD herangezogen werden.

System	Größe in Byte Text	Data	Total	Differenz Byte	%
PURE	8294	1060	9344	0	100
SAD	7100	628	7728	−1616	82,7

Tabelle 4.7: Systemgrößen des Philosophenbeispiels PURE vs. SAD

Für den Test mußte eine Kopie des Quellprogramms eine Anpassung an das SAD-System erfahren. Dabei wurden die Schedulingspezifika von PURE entfernt und durch SAD-Implemente ersetzt. Der restliche Quelltext blieb unverändert. Somit wirkt sich auf die Größe der beiden übersetzten Programme nur die Planereinheit des jeweiligen Systems aus.

Die Planereinheit sowie die Prozeßabstraktion sind bei SAD ebenfalls wie bei dem PURE-Testprogramm als FCFS-Variante konfiguriert. Infolgedessen können die Ergebnisse verglichen werden, da sichergestellt ist, daß nicht nach unterschiedlichen Verfahren geplant wird.

Zu erwähnen ist noch, daß es sich wiederum um native Compilate (übersetzt mit GNU `g++` Version `2.96`) handelt, welche auf einer x86-Plattform ausführbar sind.

In der Tabelle 4.7 sind die Ergebnisse der beiden Systeme eingetragen, wobei die Werte des PURE-Systems aus [MSGP02] stammen. Es zeigt sich ein erheblicher Größenunterschied zwischen beiden Systemen, der nur auf die verschiedenen Schedulerimplementierungen sowie Prozeßabstraktionen zurückzuführen ist, da sich die Testprogramme bis auf systemspezifische Bestandteile gleichen. In der letzten Spalte wurde die Differenz der Systemgrößen gebildet, wobei einmal der Unterschied in Byte und ein anderes Mal in Prozent angegeben wurde, jeweils in Bezug zum anderen System.

Die Tabelle belegt, daß das SAD-System um 1616 Byte kleiner ist bzw. nur 82,7 % der Größe der PURE-Variante aufweist. Wird der Datenbereich der Systeme untersucht, kann weiterhin festgestellt werden, daß das PURE-System einiges mehr an Daten benötigt. Ein Grund hierfür könnte der Einsatz virtueller Funktionen und deren Kosten innerhalb der Ableitungshierarchie des Prozeß- und Schedulersubsystems. Die Datenbereiche fallen dennoch relativ klein aus, weil die fünf Stacks (fünf Philosophenprozesse) nicht bei den Systemgrößen berücksichtigt wurden, weil in beiden Systemen die gleiche Größe dafür verbraucht wurde. Um aber die Zahlen besser vergleichbar zu gestalten, sind diese Werte abgerechnet.

Als Ergebnis des Vergleichstests kann in Bezug auf die gebrauchten Ressourcen eine Minimierung gezeigt werden, wobei die SAD-Programmfamilie kleinere Systeme erzeugt als es die PURE-Programmfamilie zuläßt. Durch dieses Ergebnis verdeutlicht, lassen sich kleinere Systeme mit gleichem Funktionsumfang generie-

ren, obwohl eine höhere Flexibilität sowie ein größerer Konfigurationsspielraum gegeben ist.

Kapitel 5

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein modulares Softwaresystem entwickelt, welches im Bereich Planungswesen angesiedelt ist. Durch die verschiedensten Einsatzgebiete vorgegeben, sollte das Programmsystem respektive die Programmfamilie eine breite Palette an Anwendungsszenarien unterstützen. Unter anderem war auch der Bereich der tiefsten eingebetteten Systeme abzudecken, in denen meist nur sehr knappe Systemressourcen die Regel sind. Daher darf den Anwendungen nur die Funktionalität bereitgestellt werden, die auch tatsächlich eine Benutzung erfährt. Aus diesem Grund entstand ein Programmsystem mit flexibler, aber dennoch anpaßbarer bzw. maßscheiderbarer Softwarestruktur, welches mit Hilfe von Konfigurationsbeschreibungen individuell auf den Anwendungsfall abgestimmt wird. Sollte geforderte Funktionalität noch nicht Bestandteil der Programmfamilie sein, kann diese durch Erweiterung des Systems unter Nutzung bereits vorhandener Abstraktionen hinzugefügt werden, wodurch infolgedessen ein neues Familienmitglied entsteht.

Die Wiederverwendung von Softwarebausteinen war ebenfalls ein Grundgedanke bei der Entwicklung des Systems, welches aus drei Bestandteilen aufgebaut ist. Zum einen sind die fast untrennbaren Bestandteile Prozeßwesen sowie Prozeßplanung zu nennen, die aufeinander abgestimmt weiterentwickelt und konfiguriert werden müssen, damit die Garantie eines ordnungsgemäßen Verhaltens gegeben ist. Und weiterhin wird der dritte Block durch die Strategien zur Speicherung wartender Prozesse gebildet, der zwar bestimmte Eigenschaften (Verkettungsmöglichkeiten) auf Prozeßseite voraussetzt, aber dennoch relativ unabhängig als Subsystem eine Erweiterung erfahren kann.

Momentan umfaßt das gesamte System bestehend aus den drei Hauptbereichen 54 Klassen plus einigen Hilfklassen[1] mit etwa 2000 Zeilen Quelltext (loc = lines of code). Durch das flexible Design und den konfigurierbaren Ansatz des Softwaresystems war es möglich, mit relativ wenig Programmcode eine Programmfamilie zu schaffen, die sich sehr vielen verschiedenen Situationen anpassen

[1] Bezeichnet unter anderem die Anbindung von bereits vorhandenen Code. Beispielsweise die Unterbrechungs- und Zeitscheibenanbindung sowie den Synchronisationmechanismen

kann. So sind etwa 55296 unterschiedliche Konfigurationsbeschreibungen (Abschnitt 3.2.1) für das SAD-System erstellbar, wobei natürlich auch ein gewisser nicht näher bestimmter Anteil keine sinnvolle Anwendung bietet. Der Großteil der Kombinationen und Konfigurationen ist jedoch nutzbringend einsetzbar, weshalb das System mit einem breiten Einsatzspektrum glänzen kann. Durch den Mechanismus der *Templates* und im Zusammenspiel mit der Objektorientierung wurde auf der Programmebene das flexible Design umgesetzt. Es konnte ebenfalls durch die gemessenen Systemgrößen gezeigt werden, daß eine feingranulare, objektorientierte Softwarestruktur selbst unter Verwendung des *Template*-Mechanismus nicht notwendigerweise zu großen Systemen führt. Die erreichten Größen ermöglichen selbst den Einsatz in restriktiven Bereichen, in denen meist Betriebsmittelknappheit herrscht.

Während der Arbeit stellte sich jedoch heraus, daß noch einige Punkte offen geblieben sind, wobei deren Lösung nicht Gegenstand der Aufgabenstellung war. So können etwa folgende Gebiete in weiteren Forschungsarbeiten oder Projekten verwirklicht werden:

Stapelspeicher: Die immensen Konfigurationsmöglichkeiten des Systems und die im hohen Maße angewandte Technik der Inline-Expansion (aus Effizienz- und Größengründen) und den damit verbundenen Variationen in den Aufruffolgen der Systemfunktionen erschweren die Vorhersage des tatsächlich verwendeten Stapelspeichers. Daher sollte ein Werkzeug das endgültige Programm analysieren und die maximale Tiefe der Funktionsaufrufe ermitteln, um damit den benötigten Stapelspeicher zu berechnen.

Inlining-Optimierung: Durch die Anwendung der Inline-Expansion kann bei unachtsamer Einsatzweise eine Codeaufblähung entstehen, wenn Funktion die als „klein“ angenommen werden an ihren zahlreichen Aufrufstellen textlich ersetzt werden. Ebenso sind „große“ Funktionen die nur eine Aufrufposition im System besitzen textlich zu ersetzen, obwohl sie ansonsten nicht einer Inline-fähigen Methode entsprechen. Ein Werkzeug zur Berechnung der optimalen Ersetzungsstruktur wäre ein großen Hilfsmittel für die Verkleinerung von Systemgrößen.

Konfiguration: Die im Laufe der Arbeit vorgestellten Konfigurationsbeschreibungen sind durch ihre starke Strukturierung im Grunde recht leicht verständlich. Jedoch ist der Aufwand hoch, sich in die Materie der Konfigurierung eines komplexen Systems, wie es das SAD-System darstellt, einzuarbeiten. Daher wäre eine Unterstützung durch ein Konfigurationswerkzeug wünschenswert, das eine Beschreibung automatisch anhand der ausgewählten Eigenschaft aus den Merkmalsdiagrammen erzeugt.

Literaturverzeichnis

[BGP+99] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, St Malo, France, May 1999.

[CE99] Krzysztof Czarnecki and Ulrich Eisenecker. Synthesizing Objects. In *ECOOP'99 Object-Oriented Programming. Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 18–42. Springer-Verlag, Juni 1999.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming-Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

[Cus93] H. Custer. *Inside WINDOWS-NT*. Microsoft Press, 1993.

[Dij72] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, pages 72–93. Academic Press, 1972.

[HFC76] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[HH89] Ralf Guido Herrtwich and Günter Hommel. *Kooperation und Konkurrenz*. Studienreihe Informatik. Springer Verlag, 1989.

[Hüs94] Dipl.-Inform. Thomas Hüsener. *Entwurf komplexer Echtzeitsysteme*. Wissenschaftsverlag Mannheim; Leipzig; Wien; Zürich, 1994. ISBN 3-411-16441-7.

[Hüs95] Dipl.-Inform. Thomas Hüsener. *Objektorientierter Entwurf von nebenläufigen, verteilten und echtzeitfähigen Softwaresystemen*. Spektrum akademischer Verlag GmbH Heidelberg; Berlin; Oxford, 1995. ISBN 3-86025-708-0.

[Int97] Intel. *Using the RDTSC Instruction for Performance Monitoring*. http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM, 1997.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, November 1990.

[May92] Scott Mayers. *Effektiv C++ programmieren*. Addison-Wesley, 1992. erste Auflage.

[MSGP02] Daniel Mahrenholz, Olaf Spinczyk, Andread Gal, and Wolfgang Schröder Preikschat. An aspect-orientied implementation of interrupt synchronization in the pure operating system family. In *the 5th ECOOP Workshop on Object Orientation and Operating Systems*, Juni 11th 2002. ISBN 84-699-8733-X.

[Par75] D. L. Parnas. On the Design and Development of Program Families. Technical Report BS I 75/2, TH Darmstadt, 1975.

[Sch00] Michael Schulze. Präemptives Planungswesen für aktive und passive Objekte. pages 55–56, 2000.

[SDO02] Douglas C. Schmidt, Mayur Deshpande, and Carlos O'Ryan. Operating System Performance in Support of Real-time Middleware. In *the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'02)*. Januar 2002.

[SP94] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[SP99a] Wolfgang Schröder-Preikschat. Betriebssystembaukasten. *LOG IN — Informatische Bildung und Computer in der Schule*, 19(5):67–68, December 1999.

[SP99b] Wolfgang Schröder-Preikschat. UNIX System Programming, 1999. Lecture Notes.

[SP01] Wolfgang Schröder-Preikschat. Betriebssystementwurf, 2001. Lecture Notes.

[SSPSS00] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 270–277, Newport Beach, California, March 15–17, 2000. IEEE Computer Society. ISBN 0-7695-0607-0.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997. third edition.

[Tan85] A. S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 1985. Second Edition.

[Tan88] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1988. Second Edition.

[Urb01] Matthias Urban. PUMA *User's Manual*. Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Verteilte Systeme, http://www-ivs.cs.uni-magdeburg.de/∼puma, 2001.

[Wag94] Klaus W. Wagner. *Einführung in die Theoretische Informatik, Grundlagen und Modelle*. Springer-Verlag Berlin Heidelberg, 1994. ISBN 3540581391.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[ZA95] Dieter Zöbel and Wolfgang Albrecht. *Echtzeitsystem–Grundlagen und Technicken*. Internatinal Thomson Publishing, 1995. ISBN 3-8266-0150-5.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken und Zitate sind als solche kenntlich gemacht. Die Arbeit wurde weder einer anderen Prüfungsbehörde vorgelegt noch veröffentlicht.

Michael Schulze
Magdeburg, den 14. Juli 2002