Diplomarbeit

Scheduling von unbeschränkten Taskfolgen mit relativen Timing Constraints

Wilko Hein
wilko.hein@tu-clausthal.de

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Clausthal, den 8. September 1999, Wilko Hein

Scheduling von unbeschränkten Taskfolgen mit relativen Timing Constraints

Diplomarbeit an der TU Clausthal, Institut für Informatik

Eingereicht von Wilko Hein
Osteröder Straße 6b
D-38678 Clausthal-Zellerfeld

Tel/Fax: +49 5323 3828
Mobil: +49 172 5448 700
mailto: wilko.hein@tu-clausthal.de
http://wilko.hein.net

am 8. September 1999

Betreuung durch Prof. Dr. phil. Klaus Ecker (Gutachter)
TU Clausthal, Institut für Informatik
D-38678 Clausthal-Zellerfeld

Prof. Dr. rer. nat. Ingbert Kupka (Zweitgutachter)
TU Clausthal, Institut für Informatik
D-38678 Clausthal-Zellerfeld

Zusammenfassung

Im Rahmen dieser Diplomarbeit wird das Planen von unendlichen Taskfolgen mit relativen Zeitbeschränkungen auf Single-Prozessor-Systemen untersucht.

Einleitend werden nach einer konkreten Problemformulierung einige bereits veröffentlichte Lösungsstrategien vorgestellt. Diese sind jedoch nur bedingt auf das behandelte Scheduling-Problem anwendbar, da oft nur endliche Schedules oder Probleme ohne Release-Constraints betrachtet werden.

Basierend auf einigen hergeleiteten theoretischen Aussagen werden dann zwei neue Lösungsstrategien formuliert. Die beweisbar korrekte Methode erweitert ein Branch-and-Bound-Vorgehen auf release-beschränkte Probleme; das zweite, heuristische Verfahren dagegen generiert flexible Schedules mittels einer Simulated-Annealing-Strategie. Dabei wird ein neues Repräsentations- und Verarbeitungsmodell für Schedules vorgestellt und verwendet.

In experimentellen Ergebnissen kann schließlich die Qualität der vorgestellten Methoden gezeigt werden. Dazu wird eine eigens implementierte Simulations-Software verwendet. Neben statistischen Auswertungen erfolgt dabei auch die Simulation grundlegender Beispielprobleme.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Problemstellung

Die Planung von Vorgängen und Tätigkeiten ist in vielen Fachgebieten ein weitverbreitetes Problem. Ob in der Unternehmensleitung, dem Operations Research, im Materialfluß-Management oder in Echtzeitsystemen: Es gilt, eine Abfolge von Tätigkeiten zu finden, welche gewisse Bedingungen erfüllt und bezüglich bestimmter Kriterien optimal ist.

Die Bestimmung dieser Tätigkeitsfolge kann entweder bei Bedarf zur Laufzeit, etwa durch zugewiesene Prioritäten, erfolgen oder aber bereits in einer Vorbereitungsphase getroffen werden. Im zweiteren Falle spricht man vom *zeitbezogenen Scheduling*. Bei den dabei verwendeten Methoden kann die Einhaltung der vorgegebenen Beschränkungen sehr viel besser garantiert werden als z.B. bei prioritätsbasierten Methoden. In Rahmen dieser Arbeit[1] wird daher das Augenmerk besonders auf das zeitbezogene Scheduling gelegt.

Die dabei vorzugebenen Beschränkungen werden vorwiegend durch die Anforderungen des Problems auferlegt: Die temporale Vorgängerrelation, der Ausschluß gleichzeitig ablaufender Aktivitäten oder die Einhaltung von gewissen Zeitbeschränkungen für Warte- oder Durchlaufzeiten. In Echtzeitsystemen werden diese *Constraints*[2] durch die zugrundeliegende Aufgabenstellung definiert, indem beispielsweise eine Kontrollaufgabe mindestens einmal pro Minute ausgeführt werden muß. Unterschieden werden muß zwischen absoluten und relativen Constraints. Absolute Constraints geben an, zu welchem Zeitpunkt eine Aufgabe auszuführen ist. Relative Constraints dagegen beziehen sich auf durch andere Jobs festgelegte Ereignisse, vornehmlich die Beendigung einer anderen Aufgabe. Die Aussagen könnten also lauten: “Führe Aufgabe A zwischen 10 und 11 Uhr aus” respektive “Führe Aufgabe A aus, wenn nach Ende von B mindestens 10 Minuten vergangen sind”.

Desweiteren sind Echtzeitsysteme meist auf die zeitlich nicht beschränkte Ausführung der ihnen zugedachten Aufgabe ausgelegt. Für sämtliche Teilaufgaben ist dabei vorgegeben, in welcher Beziehung die einzelne zugehörige Ausführung zu den anderen Tätigkeiten stehen muß. Besonders häufig werden dabei *zyklische* Aufgaben (*Jobs,*

[1] Der vorliegende Text ist nach der seit dem Jahre 1901 gültigen deutschen Rechtschreibung [D94] abgefaßt. In einigen Fällen werden jedoch eingedeutschte Fremdwörter verwendet, um eine Konsistenz mit der englischsprachigen Referenzliteratur zu garantieren. Das Geschlecht der Begriffe wird dabei so festgelegt, wie dies in vielen deutschsprachigen Arbeiten der Fall ist oder wie es aufgrund von synonymer Verwendung sinnvoll ist. Beispielsweise wird der Begriff “Task” maskulin (“der” Task) gebraucht, da *Task*, *Job* und *Auftrag* synonym verwendet werden.

[2] **constraint** *n* Zwang *m*; (*PSYCH*) Befangenheit *f*

Tasks[3]) betrachtet. Dies sind Tätigkeiten, welche mit gewissen zeitlichen Abständen unbeschränkt wiederholt werden müssen. Während der Ausführungsphase ist es notwendig, sämtliche vorgegebenen Beschränkungen einzuhalten, etwa die Verwendung beschränkter Ressourcen.

Häufig werden Echtzeitsystemen nach dem *Single-Processor*-Prinzip designed, d.h. sämtliche Aufträge müssen von einem Prozessor ausgeführt werden – der Systemprozessor ist in diesem Fall die grundlegende, beschränkende Ressource. Auch in dieser Arbeit wird überwiegend nur die Beschränkung *einer* Ressource betrachtet, somit beispielsweise das Scheduling für Einprozessor-Echtzeitsystem. Zu den spezifizierten Jobs ist dabei eine Art und Weise zu finden, in welcher die wiederkehrenden Aufgaben vom System erledigt werden können, ohne dabei die an sie gestellten Bedingungen zu verletzen. Dieser Ausführungsplan wird als *Schedule*[4] bezeichnet und gibt an, zu welchem Zeitpunkt der Prozessor mit welcher Tätigkeit beschäftigt ist.

Beim Scheduling von *unbeschränkten* Taskfolgen ist es besonders wichtig, der steten Wiederholung der Tätigkeiten Rechnung zu tragen. Es gibt dabei mehrere Verfahren, durch die ein Ausführungsplan generiert oder eine Startentscheidung zur Laufzeit vorgenommen werden kann. Hierbei handelt es sich um komplexe Probleme, so daß auf die Struktur der Verfahren besonderes Augenmerk gelegt werden muß. Wichtig sind dabei nicht nur Zeit- und Speicherbedarf, auch Korrektheit und Skalierbarkeit spielen eine große Rolle, da Real-Life-Probleme häufig mit großen Task-Zahlen operieren.

Zusätzlich zum hohen Task-Aufkommen müssen Echtzeitsysteme flexibel auf wechselnde Anforderungen, Unregelmäßigkeiten in der Ausführung oder zusätzliche, einmalig zu erledigende Aufgaben reagieren können. Das Ausführungsmodell darf nicht starr sein, es sollten ohne größeren Aufwand neue Jobs hinzugefügt oder kleinere Abweichungen in der tatsächlichen Ausführungszeit einzelner Jobs toleriert werden können. Gleichzeitig bedarf die Verwendung von relativen Constraints besonderer Betrachtung, da die einzelnen Aufgaben hierdurch nicht nur absolut im Schedule beschränkt, sondern auch relativ zueinander gekoppelt werden. Diese Wechselwirkungen führen, ähnlich wie bei gekoppelten Differentialgleichungen, zu einer sehr starken Steigerung der Komplexität des Problems.

1.2 Motivation

Anhand einiger einfacher Beispiele soll ausgeführt werden, daß dem Scheduling von zyklischen Tasks mit relativen Timing Constraints eine grundlegende Bedeutung zukommt. Zusätzlich wird auf spezielle Probleme der jeweiligen Aufgabenstellung hingewiesen.

- **Meßwerterfassung**
  In einer Meßwarte müssen die sensorischen Ergebnisse mehrerer Sensoren regelmäßig abgefragt und gespeichert werden. Die Sensoren besitzen dabei eine gewisse Trägheit, etwa bedingt durch physikalische Limitationen. Um korrekte Resultate zu erhalten, dürfen die Abfragen daher nicht zu schnell aufeinander erfolgen. Auf der anderen Seite darf mit der Abfrage nicht zu lange gewartet werden, um garantieren zu können, daß die erfaßten Meßergebnisse bei Verarbeitung nicht veraltet sind. Die Abfragezeiten müssen in einem vorgegebenen Rahmen liegen.
  Bei der Abspeicherung der Daten kann dabei durch Fragmentierung des Datenträgers sporadisch eine kurze Verzögerung des Speicherprozesses erfolgen. Dies soll dann jedoch nicht dazu führen, daß der gesamte Meßablauf gestört wird sondern daß im schlimmsten Falle nur lokale Verzögerungen auftreten.

[3] **task** *n* (*Comp*) Prozeß *m*, Auftrag *m*, Aufgabe *f*.
[4] **schedule** *n* (*list*) Liste *f*, Tabelle *f*; (*plan*) Programm *nt*

Desweiteren soll ein Benutzer die Darstellung der Daten am Monitor verändern können, z.B. virtuelle Meßgeräte verschieben oder vergrößern. Dies sind sporadische Aufgaben, welche den Prozessor jedoch zusätzlich belasten und die daher in den Schedule eingeplant werden müssen.
Zur Archivierung der Meßdaten muß zusätzlich jeden Tag zu einer vorher bestimmten Uhrzeit eine Sicherungskopie angelegt werden – in diesem Fall ist das Timing-Constraint also absolut zu verstehen. Die Unterscheidung ist hier relativ klar, da die Messungen untereinander relativ gekoppelt sind und sich stets auf die Beendigung des vorhergehenden Meßvorgangs beziehen.

- **AIMS**
  Schon im Jahre 1994 wurde im Flugzeugbau bei der Boeing 777 das integrierte Airplane Information Management System (AIMS) eingeführt. Die Funktionalität umfaßt dabei sowohl Echtzeit- als auch Nicht-Echtzeit-Aufgaben. Basierend auf ARINC 659, einem Buskonzept, welches auch die Synchronisation mehrerer Prozessoren unterstützt, werden nach [CDHC94] durch Offline-Scheduling von Ausführungs- und Kommunikationsressourcen Ablaufpläne erstellt, die sowohl einen hohen Datendurchsatz besitzen als auch die zeitlichen Anforderungen erfüllen. Dieses System ist sehr zeitkritisch, so daß auf die Korrektheit und Flexibilität des erstellten Schedules hoher Wert gelegt werden muß.

- **Terminplanung im Produktionsprozeß**
  Bei der Herstellung gewisser Waren gilt es ebenso, bestimmte Zeitschranken nicht zu über- oder zu unterschreiten. Bei der Gärung von Wein etwa muß eine gewisse Mindestgärzeit garantiert werden. Während dieser Zeit sind die Fässer nicht für andere Aufgaben verfügbar. Ebenso müssen Transportkapazitäten für Trauben oder Flüssigkeiten eingeplant werden. Zur Überwachung des Gärungsprozesses gilt es ebenso, regelmäßig Proben zu entnehmen und zu untersuchen. Dabei darf zwischen Entnahme und Untersuchung keine zu lange Zeitspanne liegen, um ein korrektes Ergebnis zu garantieren. Außerdem darf die Gesamtgärungszeit nicht zu groß werden, um keinen Essig zu erhalten. Trotzdem sollte der etwaige Ausfall von Arbeitskräften oder Maschinen nicht den Verlust der gesamten Ernte bedeuten, so daß der Terminplan ausreichend flexibel sein muß, um auf Störungen reagieren zu können.
  Weitere Beispiele aus dem Bereich des Operations Research finden sich auch in [DD95].

- **Waschmaschinen-Steuerung**
  Während des Durchlaufs verschiedener Programmstufen müssen gewisse Steuer- und Überwachungstätigkeiten regelmäßig ausgeführt werden, beispielsweise die Kontrolle auf Wasserlecks oder die Auswertung eines programmierten Timers. Je nach Phase des Programmes müssen aber auch zusätzliche (zyklische) Jobs erledigt werden, denkenswert ist der Wechsel von Schleudergeschwindigkeit oder -richtung, die Zugabe von Waschmittel oder eine Temperatursteuerung. Obwohl in diesem Beispiel dieses System nicht zeitkritisch ist, muß sichergestellt werden, daß beim Übergang von einer Phase im Schedule zur nächsten keine Constraintverletzungen auftreten.

All die vorgestellten zeitlichen Beschränkungen und Beziehungen lassen sich als Constraints formulieren. Dabei sind die Bedingungen mehr oder weniger kritisch, im allgemeinen sollen sie jedoch eingehalten werden. Das zugrundeliegende Modell ist stets dasselbe und wird im folgenden für ein beliebiges Echtzeitsystem genauer definiert.

1.3 Bezeichnungen

Zur Problemformulierung soll zuerst die Nomenklatur vorgestellt werden, indem die wichtigsten Schlagworte kurz vorgestellt und ihre Beziehungen untereinander herausgearbeitet werden.

1.3.1 Task

- **Job, Task**
  Unter einem *Job* oder einem *Task* versteht man einen –in diesem Falle regelmäßig auszuführenden– Arbeitsauftrag. Dieser Auftrag wird spezifiziert durch die Dauer der aktiven Tätigkeit und den minimalen und maximalen Abstand bis zur nächsten Ausführung. Dabei werden die Begriffe *Job*, *Task*, *Auftrag* und *Aufgabe*, wie in der Literatur üblich, synonym verwendet.

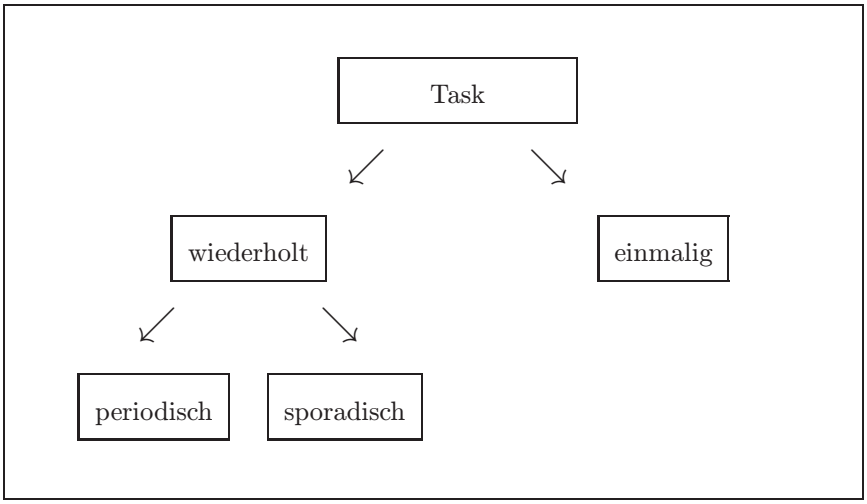

Abbildung 1.1: *Einteilung eines Tasks in verschiedene Ausführungs-Kategorien, etwa nach [CA97]*

  Je nach Aufgabentypus wird der Task nur einmalig oder wiederholt ausgeführt. Für wiederholte Task muß dabei noch zwischen regelmäßig (**periodisch**) und sporadisch ausgeführten Jobs unterschieden werden. Bei **sporadischen** Tasks ist der nächste Ausführungszeitpunkt nicht a-priori gegeben sondern erfolgt in Reaktion auf externe Ereignisse, Benutzerinteraktionen oder ähnliches. Dabei ist oft spezifiziert, daß die erneute Auslösung eines sporadischen Tasks erst nach Ablauf einer Latenzzeit möglich ist, um die Ausführungsfrequenz nach oben zu beschränken.
  Im Rahmen dieser Arbeit wird besonderes Augenmerk auf die periodischen Jobs gelegt.

- **Taskinstanz**
  Die kontinuierliche Ausführung einer vorgegebenen Aufgabe –eines Tasks– erfordert es, wiederholt und regelmäßig *Instanzen*[5] dieses Tasks auszuführen. Jede einzelne Abarbeitung wird dabei als *Taskinstanz* bezeichnet.

[5] **instance** *n* Instanz *f* (konkrete Ausprägung eines Objekts einer Menge von Gegenständen derselben Art)

Die Spezifikation des Arbeitsauftrages erfolgt etwa in Form eines Unterprogrammes, welches in regelmäßigen Zeitintervallen auszuführen ist. Beispielsweise spezifizieren Programmcode und zugehöriges Ausführungsintervall einen Task vollständig. Eine einzelne Taskinstanz entspricht dann der konkreten Ausführung des Codes zu einem vorgegebenen Zeitpunkt. Bei einer zeitlich unbeschränkten Aufgabe gehören zu einem Job somit unendlich viele Taskinstanzen. Diese können, etwa bei periodischer Ausführung, parametrisierte Startzeiten besitzen.

Im Rahmen des Schedulings wird die Definition eines Tasks abstrakt verwendet: Ein Auftrag wird nur durch ein Set an temporalen Parametern definiert (s.u.) und losgelöst von der konkreten zu erledigenden Aufgabe betrachtet. Die Parameter spezifizieren dabei etwa die maximal benötigte Arbeitszeit, die für die einmalige Ausführung einer Taskinstanz benötigt werden kann.

- **Constraints**
  Die zeitliche Abfolge der Taskinstanzen wird mittels sogenannter **Constraints** eingeschränkt. Üblicherweise werden hier sogenannte **Standard Constraints** verwendet: Sie schränken die Zeitspanne zwischen zwei zum Job gehörigen Abarbeitungen auf ein Intervall ein – den minimalen und den maximalen Abstand. Sie gehören zur Gruppe der **Relativen Constraints**, d.h. sie beziehen sich nicht auf absolute Ausführungszeiten sondern auf die zeitliche Distanz zur jeweils letzten Ausführung. Exakt werden hier **relative Releasezeit** und **relative Deadline** angegeben, eine genauere Formalisierung findet sich weiter unten. Im Rahmen dieser Arbeit werden nur Constraints *innerhalb* eines Jobs zugelassen. Beschränkungen, die zwei Jobs synchronisieren, etwa bei Interprozeß-Kommunikation, werden vorerst nicht betrachtet.
  Neben den explizit vorzugebenden Constraints müssen *implizite* Constraints betrachtet werden: Gibt es im System beschränkte Ressourcen, im hier betrachteten Falle etwa der Prozessor, besagt das **Gleichzeitigkeits-Constraint**, daß zu jedem Zeitpunkt nur die zur Verfügung stehenden Ressourcen verwendet werden können. Im Einprozessor-System können somit nie zwei oder mehr Aufgaben gleichzeitig bearbeitet werden.
- Das **Job-Set** bezeichnet eine Menge von Jobs (oder auch Tasks) mit den dazugehörenden definierenden Parametern. Diese Menge umfaßt im allgemeinen sämtliche Aufgaben, die das Echtzeitsystem *regelmäßig* zu erfüllen hat.

1.3.2 Schedule

- Mit **Schedule** wird ein Instanzierungs-Schema bezeichnet, welches sämtliche benötigten Taskinstanzen exakt spezifiziert. Für ein vorgegebenes Zeitintervall wird dadurch ein Abarbeitungsplan dargestellt, der vorgibt, welche Aufgabe zu welchem Zeitpunkt erledigt werden muß. Dies kann etwa dadurch erreicht werden, daß zu jeder Instanz die geplante Startzeit angegeben wird. Dadurch wird (zusammen mit den taskspezifischen Ausführungszeiten) implizit festgelegt, zu welchem Zeitpunkt der Prozessor welchen Job bearbeitet.

  Ein Schedule kann auf vielfältige Art und Weise spezifiziert werden:

  – Die graphische Darstellungen durch **Gantt-Diagramme** (siehe dazu auch [DD95]), wie sie z.B. in Abbildung 1.2 oder im Abschnitt 6.1 verwendet wird, ist besonders intuitiv verständlich. Über der zeitlichen Achse sind durch Balken die einzelnen Taskinstanzen aufgetragen. Pro Zeile wird dabei meist ein Task dargestellt, so daß die Zusammengehörigkeit der Instanzen klar erkenntlich ist.
  – Da die Ausführungszeiten der einzelnen Tasks als konstant (bzw. zumindest nach oben beschränkt) angenommen werden, genügt auch die alleinige

Angabe von **Startzeiten**: Pro Taskinstanz muß dann vorgegeben werden, zu welchem Zeitpunkt mit der Ausführung begonnen werden muß. Dies läßt sich etwa durch Listen realisieren, die zu jedem Job eine Abfolge von Startzeiten angeben. Zu den vorgegebenen Terminen muß dann eine Instanz des korrespondierenden Tasks gestartet werden, beispielsweise also:

$$\begin{aligned} S(J_1) &= (0,\ 13,\ 36, \ldots) \\ S(J_2) &= (9,\ 17,\ 55, \ldots) \\ S(J_3) &= (4,\ 11,\ 21, \ldots) \end{aligned}$$

– Speziell bei Single-Prozessor-Systemen (s.u.) kann auch einfach die **Reihenfolge** der Jobinstanzen angegeben werden, welche dann in unmittelbarer Abfolge ausgeführt werden. Eine Angabe von

$$S = (J_1,\ J_3,\ J_2,\ J_3, \ldots)$$

bedeutet dabei, daß mit einer Instanz des ersten Jobs begonnen werden soll und unmittelbar nach Ende der Ausführung Job 3 gestartet werden muß usw. Zu beachten ist dabei wiederum, daß die Ausführungszeiten oft als konstant angenommen werden.

Im Rahmen dieser Notation kann es notwendig sein, Wartezeiten zu verwenden. Details zu sogenannten *Idle-Tasks* siehe unter 3.2.2.

Die drei genannten Darstellungsmöglichkeiten sind bei Verwendung konstanter Ausführungszeiten auf Single-Prozessor-Systemen äquivalent.

- **Zulässigkeit, Gültigkeit**
  Ein Schedule wird genau dann als *zulässig* oder *gültig* bezeichnet, wenn keine Constraints verletzt werden. Damit sind sowohl die explizit vorgegebenen Timing Constraints als auch Nicht-Gleichzeitigkeits-Bedingungen gemeint.
- **Zyklus**
  Um mittels eines endlichen Schedules eine unendliche Taskfolge spezifizieren zu können, werden *Zyklen* definiert. Der Zyklus besteht dabei aus einem Teil des Schedules, umfaßt jedoch üblicherweise den gesamten Schedule. Zyklenstart und -ende werden zu einem virtuellen "Kreis" verbunden, so daß während der Abarbeitung des Planes bei Erreichen des Zyklenendes ohne Unterbrechung am Zyklenbeginn fortgefahren wird. Diese Verbindungsstelle wird auch **Zyklenschluß** genannt. Zu beachten ist dabei, daß die Timing Constraints auch an der Stelle des Zyklenschlusses nicht verletzt werden dürfen.

1.3.3 Echtzeitumgebung

- **Harte** versus **weiche Echtzeitsysteme**
  Je nach Art des Systems sind Constraint-Verletzungen mehr oder weniger kritisch. In *Hard Realtime Systems* ([FAQRT]) bewirkt eine Deadline-Überschreitung einen völligen Zusammenbruch des Systems oder andere, schwerwiegende Probleme. In instabilen Steuerungssystemen, etwa Raketentriebwerken, muß zum Beispiel eine vorgegebene Steuerrate garantiert aufrechterhalten werden, da ansonsten nicht-korrigierbare Abweichungen von der vorhergesehenen Flugbahn auftreten können.
  Im Gegensatz dazu werden *weiche* Echtzeitsysteme bei Deadline-Überschreitung nur eine Verringerung des Datendurchsatzes o.ä. erfahren. Bei Kommunikationsaufgaben etwa führt ein verspätetes Auslesen eines Datenpuffers dazu, daß das nächste ankommende Datenpaket abgewiesen werden muß, da der Zwischenspeicher noch nicht wieder zur Verfügung steht. In diesem Zusammenhang beeinflußt eine Verspätung zwar die Systemleistung, führt jedoch nicht zu kritischen

Problemen. Allerdings könnte auch dieses System zu einem harten Echtzeitsystem werden, wenn z.B. nicht per Handshaking der Empfang des Datenpaketes quittiert würde: Wäre der Buffer nicht verfügbar, so ginge das neue Paket unwiederbringlich verloren.

- **Offline-** versus **Online-Scheduling**
  Unter Offline-Scheduling wird die Erstellung des Ausführungsplans *vor* der Laufzeit verstanden; während des eigentlichen Ablaufes werden die Instanzen dann nur zu den vorgegebenen Zeiten gestartet. Im Gegensatz dazu werden bei Online-Verfahren *während* der Laufzeit Entscheidungen darüber getroffen, welcher Task als nächster auszuführen ist und wann er gestartet werden muß. Natürlich sind auch hybride Verfahren möglich, die einen Teil der Entscheidung offline vorgeben und erst online die endgültigen Startzeiten berechnen.
- Wie oben bereits erwähnt wurde, werden in dieser Arbeit vorrangig **Einprozessorsysteme** betrachtet, wobei dieser die grundlegende beschränkende Ressource darstellt. Daneben sind auch **Mehrprozessorsysteme** möglich, wobei dann mehrere Tasks parallel ausgeführt werden können. In diesem Falle ergeben sich weitere Beschränkungen etwa durch gemeinsamen Datenzugriff und durch Kommunikationsaufgaben.

1.4 Formale Problembeschreibung

Ein periodischer Jobs J_i wird durch drei temporale Parameter

$$J_i \quad = \quad (e_i,\ r_i,\ \delta_i), \qquad e_i, r_i, \delta_i \in N_0^+$$

spezifiziert (siehe auch Abbildung 1.2). Dabei liegt ein *deterministisches Modell* zugrunde, so daß sämtliche Zeiten als a-priori bekannt angenommen werden können. Die geeignete Festlegung dieser Zeiten, so daß die Problemanforderungen an das *gesamte* System gewährleistet sind, wird etwa in [Sak98] eingehender beschrieben.

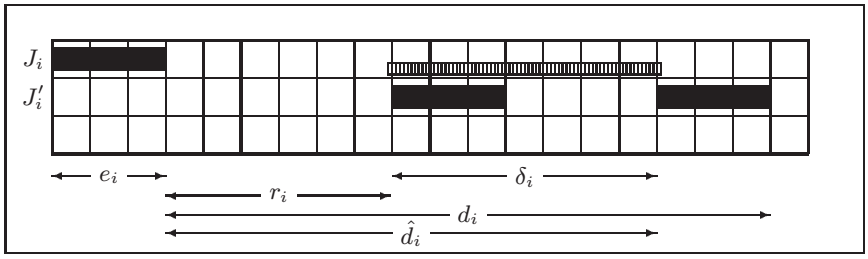

Abbildung 1.2: *Spezifikation des Task J_i durch Tripel $(e_i,\ r_i,\ \delta_i)$. Die ganzen, nichtnegativen Zeitangaben bestimmen die Ausführungszeit (e_i), die Releasezeit (r_i) und die Länge des Start Time Windows (δ_i). Die nächste Instanz des Tasks (J_i', die beiden Extremfälle sind gezeigt) muß innerhalb dieses durch die Parameter vorgegebenen Fensters beginnen. Äquivalent zur Spezifikation über das Slack Time Window ist die Angabe der Deadline $(d_i = r_i + \delta_i + e_i)$ oder der Slack Time $(\hat{d}_i = r_i + \delta_i)$.*

- $\mathbf{e_i} \in N_0^+$

  Obere Schranke für die **Ausführungszeit** (*Execution time*). Die tatsächliche Ausführungszeit weiche dabei in der Realität nur minimal nach unten von dieser Schranke ab, so daß diese Varianzen im Scheduling-Prozeß unberücksichtigt bleiben können. Die Vorgabe konstanter, deterministischer Ausführungszeiten erleichtert den Scheduling-Prozeß stark und ist daher weitverbreitet. Zur Berechnung der maximalen Ausführungszeit für Taskinstanzen siehe [PS91].

  Pathologische Ausführungszeiten $e_i = 0$ sind zwar möglich und sollen nicht generell ausgeschlossen werden, im allgemeinen ist deren Verwendung jedoch nicht sinnvoll, so daß eine Einschränkung auf $e_i \in N^+$ vorgenommen werden kann.

- $\mathbf{r_i} \in N_0^+$

  Relative **Release**[6]**-Zeit** zwischen zwei Taskinstanzen bzgl. des Endes der letzten Task-Instanz. Während dieser Zeitspanne darf keine neue Instanz des betroffenen Jobs ausgeführt werden, der Prozessor steht jedoch für andere Aufgaben zur Verfügung.

- $\delta_\mathbf{i} \in N_0^+$

  Das **Start Time Window** gibt schließlich den verfügbaren Spielraum für den Start der folgenden Taskinstanz an. Diese Darstellung ergibt sich aus den verwendeten Gantt-Diagrammen (Abbildung 1.2) und ist auch für weiter unten angestellte theoretische Betrachtungen gut geeignet.

Häufig wird alternativ zu obigem δ_i die **Slack**[7] **Time** ($\hat{d}_i$) als minimaler zeitlicher Abstand vom Instanzende zum Beginn der nächsten Instanz vorgegeben. Diese Slack Time darf nicht kleiner als die Releasezeit sein ($\hat{d}_i \geq r_i$) und ist bei einigen Anwendungen durch die Problemstellung vorgegeben. In der Literatur wird ebenfalls alternativ die **Relative Deadline** verwendet, $d_i = r_i + \delta_i + e_i$, welche die maximale temporale Distanz vom Ende einer Instanz bis zum Ende der nächsten angibt. Dieser Parameter muß ebenfalls ausreichend dimensioniert sein ($d_i \geq r_i + e_i$).

Je nach Anwendungsfall kann die dritte Komponente im Tripel durch δ_i, $\hat{d}_i$ oder d_i gegeben sein, die Angaben sind jedoch vollständig äquivalent. In dieser Arbeit soll hauptsächlich die Slack-Time-Notation benutzt werden, da sie sich für verwendete theoretische Betrachtungen eignet und einfach validifizierbar sind: Sämtliche nichtnegativen Tripel stellen gültige Jobs dar.

Ohne Beschränkung der Allgemeinheit wird hier angenommen, daß sämtliche Zeitangaben ganzzahlig sind. Daher ist in Abbildung 1.2 zu beachten, daß das Start Time Window genau die ganzzahligen Startzeiten zuläßt, die innerhalb des Fensters liegen. Es ist dabei *nicht* maßgeblich, ob das Intervall $[s'_i,\ s'_i + 1]$ vom Fenster eingeschlossen wird, leicht zu erkennen an den beiden extremalen Instanzen J'_i in der Abbildung.

Zu dem durch

$$\begin{aligned} J &= (J_1,\ J_2, \ldots,\ J_n) \\ J_i &= (e_i,\ r_i,\ \delta_i), \qquad\qquad i = 1, \ldots,\ n \end{aligned}$$

vorgegebenen Job-Set gilt es nun, einen gültigen, unendlichen Schedule zu finden, welcher in Zyklenform dargestellt werden soll. Für jeden Job muß die Anzahl $k_i > 0$ ($i = 1, \ldots, n$) der zu verwendenden Instanzen sowie die jeweilige Startzeit gefunden werden:

$$s_{ij} := \text{const}, \qquad i = 1, \ldots, n, \qquad j = 1, \ldots, k_i \tag{1.1}$$

[6] **release** *n* (*freedom*) Entlassung *f*; (*TECH*) Auslöser *m*; *vt* befreien, entlassen

[7] **slack** *adj* (*loose*) lose, schlaff, locker; (*careless*) nachlässig

spezifiziert die Startzeit von Instanz j zu Job J_i. Der Zyklus beginne zum Zeitpunkt 0 und habe die (zu ermittelnde) Länge t.

An diese Instantiierung werden dabei folgende Anforderungen gestellt, $i = 1, \ldots, n$ und $j = 1, \ldots, k_i$:

$$0 \leq s_{ij} \tag{1.2}$$

$$s_{ij} + e_i \leq t \tag{1.3}$$

$$s_{i0} \in [s_{ik_i} + r_i - t, \ \ldots, \ s_{ik_i} + r_i + \delta_i - t] \tag{1.4}$$

$$s_{ij} \in [s_{i(j-1)} + r_i, \ \ldots, \ s_{i(j-1)} + r_i + \delta_i] \quad \text{für } j > 0 \tag{1.5}$$

$$\emptyset = [s_{ij}, \ \ldots, \ s_{ij} + e_i] \ \cap \ [s_{i'j'}, \ \ldots, \ s_{i'j'} + e_{i'}] \tag{1.6}$$

$$\text{für } i' = 1, \ldots, n, \quad j' = 1, \ldots, k_{i'}, \quad i \neq i'$$

Diese Bedingungen ergeben sich daraus, daß sämtliche Ausführungszeiten innerhalb des Zyklus liegen müssen, (1.2) und (1.3), und daß sowohl an der Stelle des Zyklenschlusses (1.4) als auch innerhalb des Zyklus (1.5) die Timing Constraints eingehalten werden müssen. Weiterhin darf zu jedem Zeitpunkt nur ein Prozeß bearbeitet werden, die Ausführungsintervalle dürfen sich nicht überschneiden (1.6).

Kann ein solcher Schedule, d.h. ein Instantiierungsschema nach (1.1), das den angegebenen Constraints genügt, mit zugehörigen $k_i > 0$ und t gefunden werden, so stellt dieses Schema einen gültigen Schedule dar. Das Finden eines solchen Schedules wird als das **Scheduling-Problem** bezeichnet.

1.5 Struktur des Dokumentes

Im Rahmen dieser Arbeit wird das Scheduling-Problem unter folgenden Gesichtspunkten eingehender betrachtet:

- Verwendbarkeit relativer Timing-Constraints
- Eignung für unendliche Schedules
- Flexibilität des Schedules bezüglich
  - Parameter-Variation zur Laufzeit
  - Hinzufügen dynamischer (aperiodischer) Tasks
  - Unvorhersehbar wechselnder Ausführungsbedingungen
  - Verwendbarkeit bereits existierender gültigen Schedules im inkrementellen Scheduling
- Geringe Komplexität des Planungsprozesses
- Intuitivität des Schedules und des Schedulingprozesses

Im Kapitel 2 werden dazu einige verwandte Lösungsstrategien vorgestellt und kurz dargelegt, inwiefern sie den gestellten Anforderungen genügen. Kapitel 3 liefert einige theoretische Grundlagen für das zyklische Scheduling, insbesondere für das Scheduling von 2-Job-Problemen. Darauf aufbauend widmet sich Kapitel 4 den *vollständigen und korrekten* Scheduling-Verfahren. Es wird dort eine neue Methode zum exakten Scheduling der geforderten Job-Sets vorgestellt. Alternativ dazu kommen im Teil 5 *iterative* (heuristische) Methoden zur Sprache; eine dort spezifizierte innovative Methode erfüllt viele der oben gegebenen Anforderungen. Experimentelle Ergebnisse zu den einzelnen Ansätzen finden sich in Kapitel 6 und werden im zusammenfassenden Teil 7 noch beurteilt. Der Anhang schließlich ist der Dokumentation von zusätzlichen Arbeitsergebnissen vorbehalten.

Kapitel 2

Verwandte Lösungsstrategien

Sämtliche Scheduling-Verfahren für *nichtunterbrechbare Tasks* lassen sich grob in vier Gruppen unterteilen (siehe etwa [Cho97]):

- **Fixed Priority Scheduling**
  Zur Laufzeit wird die Entscheidung, welcher Task als nächstes zur Ausführung kommt, nach vorher (oft vom Entwickler) spezifizierten Prioritäten vorgenommen.

- **Dynamic Priority Scheduling**
  Die Wahl des als nächstes auszuführenden Tasks wird in Abhängigkeit von dynamischen Prioritäten getroffen. Dazu gehört etwa die EDF-Strategie (*Earliest Deadline First*, [Eck92]). Auch durch andere Prioritäts-Funktionen (etwa *Dynamic Priority Ceiling* nach [SBL94] oder *Stack-Based Protocol*) lassen sich auf Einprozessorsystemen gute Resultate erzielen.

- **Static Time-based Scheduling**
  Schon vor Beginn der Ausführung der geplanten Aufgabe werden sämtliche Startzeiten der einzelnen Tasks exakt berechnet. Zur Laufzeit erfolgt dann nur der eigentliche Start der Instanz. Mit solchen Verfahren lassen sich relative Constraints sehr viel einfacher umsetzen als mit prioritätsbasierten Verfahren.

- **Dynamic Time-based Scheduling**
  Werden die offline ermittelten Startzeiten nicht statisch gespeichert sondern dynamisch berechnet oder angepaßt, kann eine sehr viel höhere Flexibilität des Scheduling-Prozesses erreicht werden. Die Komplexität des Problemes jedoch steigt dabei stark an; außerdem muß ein großer Aufwand getrieben werden, um die Fristigkeit aller Tasks auch in der fernen Zukunft zu garantieren.

Die prioritätsbasierten Scheduling-Methoden können dabei relative zeitliche Constraints nur bedingt umsetzen. Aus diesem Grunde werden im folgenden nur Verfahren des *Time-based Scheduling* eingehender betrachtet. Zur einfacheren Klassifikation der vorliegenden Arbeit sollen dazu einige verwandte Lösungsprinzipien vorgestellt werden. Diese werden anhand folgender Kriterien klassifiziert (Abbildung 2.1):

Funktionsprinzip *(Offline / Online)*		
Echtzeitverhalten *(Hard / Soft Realtime System)*		
Ausführungsumgebung *(Single / Multiple Processor)*		
Tasktypen *(Periodisch / Sporadisch bzw. einmalig)*		
Ausführungszeiten *(Exakt / Beschränkt)*		
Release-Zeiten *(Absolut / Relativ)*		
Deadlines *(Absolut / Relativ)*		
Unterstützung Aperiodischer Tasks *(Offline/Online)*		

Abbildung 2.1: *Klassifikation der Scheduling-Algorithmen bezüglich verschiedener Kriterien*

Pro Kategorie sind dabei zwei Merkmalsgruppen vorgesehen. Die genaue Bedeutung ist der Beschriftung zu entnehmen und wird hier kurz zusammengefaßt. Besitzt ein Verfahren das angegebene Merkmal, wird der Eintrag durch "●" markiert; liegt die Eigenschaft nur schwach ausgeprägt oder in Einzelfällen vor, ist ein "○" eingetragen.

- Das **Funktionsprinzip** kann grob in zwei Gruppen eingeteilt werden: Die *Offline*-Verfahren erstellen *vor* der eigentlichen Ausführung des Programms einen Schedule. *Online*-Verfahren entscheiden dagegen dynamisch zur Laufzeit über die Ausführungsreihenfolge oder die exakte Startzeit.
- Unter **Echtzeitverhalten** wird die Systemanforderung bezüglich der Einhaltung der Timing-Constraints verstanden. *Hard Realtime Systems* sind zwingend auf die Gültigkeit der Constraints angewiesen und können ihre Aufgabe bei Verletzung nicht oder nur mit sehr hohen Mehrkosten erfüllen. *Soft Realtime Systems* dagegen erreichen bei Constraintverletzung nur nicht ihre optimale Leistungsfähigkeit, das System an sich kann jedoch weiterarbeiten.
- **Ausführungsumgebungen** können entweder *Single* oder *Multiple Processor Systems* sein. Dabei dominieren bei den angegebenen Methoden die Einprozessorsysteme stark.
- Als **Tasktypen** können etwa *periodische* Tasks auftreten, diese müssen in gewissen zeitlichen Abständen immer wieder ausgeführt werden. Für *sporadische* oder sogar *einmalige* Tasks ist a-priori nicht bekannt, zu welchem Zeitpunkt die Ausführung erfolgen muß. Beim zyklischen Scheduling benötigten diese Tasktypen daher eine Sonderbehandlung.
- Vorgegebene **Ausführungszeiten** können entweder als *exakt* angenommen werden oder variabel mittels unterer und oberer Schranke *beschränkt* werden. Auch bei exakten Vorgaben darf die tatsächliche Ausführungszeit von der geschätzten abweichen – jedoch nur nach unten. Dem Prozeß wird garantiert die benötigte Zeit zugeteilt. Die Intervallvorgabe ist flexibler, aber auch schwerer zu handhaben als eine einzelne obere Schranke.
- Die Angaben für **Release-Zeiten** und **Deadlines** können *absolut* oder *relativ* erfolgen. Sie beziehen sich dann entweder auf die absolute Zeit bei Programmausführung oder auf gewisse Ereignisse, etwa das Ende anderer Taskinstanzen.

- Auch wenn ein Algorithmus auf periodische Tasks spezialisiert ist und hier das Hauptaugenmerk gesetzt wird, sollten **aperiodische Tasks** unterstützt werden. Hierzu zählen sowohl sporadische als auch einmalige Tasks. Die Berücksichtigung kann entweder rein *online* erfolgen, indem etwa Leerkapazitäten ausgenutzt werden, oder schon *offline* erfolgen. Bei der Garantie einer Fristigkeit für sporadische Tasks ist dies etwa notwenig, um eine rechtzeitige Abarbeitung des Tasks garantieren zu können, unabhängig davon, wann der sporadische Task exakt ausgelöst wird.

Eine Zusammenfassung der Klassifikation sämtlicher unten vorgestellter Verfahren findet sich in Anhang B.

2.1 Static Cyclic Scheduling

In [CA94] schedulen Cheng und Agrawala nichtunterbrechbare periodische Task-Sets mit relativen Timing-Constraints. Abweichend von oben definierter Notation werden in der zitierten Arbeit für jeden Task die gewünschte Periode und die maximal mögliche Abweichung nach oben und unten (*High* und *Low Jitter*[1]) von der Optimalperiode gegeben. Diese Definition ist jedoch äquivalent zur Release-Time und Delta-Spezifikation von oben, es gilt:

$$\begin{aligned} r_i &= p_i - \lambda_i \\ \delta_i &= \lambda_i + \eta_i \end{aligned}$$

mit der gewünschten Periode p_i und zugehörigen Jitter-Zeiten λ_i und η_i pro Job. Pro Periode wird dabei *genau eine* Taskinstanz ausgeführt.

Funktionsprinzip *(Offline / Online)*	●	
Echtzeitverhalten *(Hard / Soft Realtime System)*	●	
Ausführungsumgebung *(Single / Multiple Processor)*	●	
Tasktypen *(Periodisch / Sporadisch bzw. einmalig)*	●	
Ausführungszeiten *(Exakt / Beschränkt)*	●	
Release-Zeiten *(Absolut / Relativ)*		●
Deadlines *(Absolut / Relativ)*		●
Unterstützung Aperiodischer Tasks *(Offline/Online)*		

Abbildung 2.2: *Klassifizierender Überblick über die "LCM" Methode aus [CA94]*

Der offline generierte zyklische[2] Schedule besitzt bei diesem Verfahren eine konstante, a-priori berechnete Länge: Basierend auf den verschiedenen Perioden wird das "LCM" (Least Common Multiple, Kleinste gemeinsame Vielfache) berechnet. Dadurch

[1] **jitter** *vi* flattern, zittern
[2] **cycle** ['saikl] *n* Reihe *f*, Zyklus *m*

ist auch die Anzahl n_i der pro Zyklus auszuführenden Jobinstanzen a-priori mitbestimmt:

$$\begin{aligned} LCM &= kgV(p_1, p_2, \ \ldots, \ , p_n) \\ n_i &= \tfrac{LCM}{p_i} \end{aligned}$$

Mit Hilfe dieses Gerüstes werden sämtliche Constraints formuliert: Die Einhaltung der Jitter-Bedingungen, die Nicht-Gleichzeitigkeit von Taskinstanzen und die Zyklus-Bedingung, welche besagt, daß auch an der Verbindungsstelle zwischen Zyklusende und -anfang die Constraints eingehalten werden müssen.

Innerhalb eines Zyklus werden dann sukzessive die Taskinstanzen verteilt. Dazu wird pro Instanz ein Intervall berechnet, innerhalb dessen mit der Ausführung des Tasks begonnen werden muß. Beim Straight-Forward-Ansatz, wobei die Intervalle stets auf der konkreten Startzeit der vorhergehenden Instanz zuzüglich der Jitter-Zeiten

$$s_i^j \in [s_i^{j-1} + p_i - \lambda_i, \ \ldots, \ s_i^{j-1} + p_i + \eta_i]$$

mit den Startzeiten s_i^j für Job i in der j-ten Instanz berechnet werden, ergaben sich dabei Probleme: Wenn etwa die Tasks immer relativ spät innerhalb des vorgesehenen Fensters gestartet werden, können am Ende des Zyklus unter Umständen nicht mehr ausreichend viele Instanzen verplant werden, um die vorgegebene Anzahl n_i zu erreichen.

Um innerhalb des vorgegebenen Rahmens dennoch gültige Schedules erzeugen zu können, werden die Startzeit-Intervalle notfalls verkleinert. Dazu wird bei jeder Taskinstanz gewährleistet, daß noch für alle (an den n_i fehlenden) Ausführungen genügend Zeit bleibt. Die Startintervalle ergeben sich dann zu

$$\begin{aligned} s_i^j \in \big[&\max\{s_i^{j-1} + p_i - \lambda_i, \ s_i^1 + (j-1)p_i - (n_i - j + 1)\eta_i\}, \ \ldots, \\ &\min\{s_i^{j-1} + p_i + \eta_i, \ s_i^1 + (j-1)p_i + (n_i - j + 1)\lambda_i\}\big] \end{aligned}$$

Die Güte der erzeugten Schedules wird mit Hilfe einer Bewertungsfunktion errechnet und basiert auf den Abweichungen von der vorgegebenen Periodizität – der Jitter soll klein gehalten werden. Dazu wird die lineare Funktion

$$\pi = \sum_{i,j} |s_i^j - s_i^{j-1} - p_i|$$

minimiert.

Ausgehend von dieser Problemspezifikation wird ein Algorithmus zur Generierung von Zyklen vorgestellt. Eine Möglichkeit ist es, die Instanzen hintereinander dem Schedule zuzufügen. Die Reihenfolge der Verplanung wird dabei durch die Bewertungsfunktion π vorgegeben: Bevorzugt werden Instanzen eingeplant, die sehr wenig von der geforderten Periodizität abweichen. Falls die bevorzugte Position im Schedule noch verfügbar ist, wird der Task an dieser Stelle verankert. Ansonsten wird durch geeignete Verschiebungen der konkurrierenden Tasks Platz für die neue Instanz geschaffen. Die Verschiebung kann dabei auch rekursiv erfolgen, so daß auch mehrere Tasks gleichzeitig umgeschichtet werden können. Als Resultat wird ein zyklischer Schedule geliefert, in dem für jede Taskinstanz der exakte Startzeitpunkt vermerkt ist.

Zur Festlegung der Reihenfolge, in der die Tasks dem Schedule zugefügt werden, wurden drei Methoden vorgestellt: “Smallest Latest Starttime First”, “Smallest Period First” oder “Smallest Jitter First”. Diese bevorzugen Taskinstanzen,

- deren Startintervall am frühesten endet (Smallest Latest Starttime First, SLsF), der Zyklus wird dadurch von vorne nach hinten aufgebaut,
- die hochperiodisch sind (Smallest Period First), der Zyklus wird somit Jobweise generiert,

- die am unflexibelsten in der Planung sind (Smallest Jitter First).

In der experimentellen Beurteilung arbeiten die Autoren Cheng und Agrawala heraus, daß "SLsF" den anderen Methoden bezüglich des Findens gültiger Schedules überlegen sei.

Bemerkung Die vorgestellte Methode bedient sich dabei nicht des Backtrackings, so daß nicht garantiert werden kann, daß eine gültige Lösung gefunden wird. Die Reihenfolge der Instanzen im Schedule wird nach Verplanung nicht mehr variiert, d.h. bei unglücklicher Wahl der Planungsreihenfolge können sich nicht-lösbare Konflikte ergeben. Desweiteren wird durch die A-Priori-Festlegung der Zyklenlänge LCM und der dabei zu verwendenden Taskzahlen n_i der Planungsraum deutlich eingeschränkt. Dadurch können bei hohen notwendigen Prozessorauslastungen oder bei großen Jitter-Intervallen unter Umständen keine gültigen Schedules generiert werden. Durch das statische Zuweisen der exakten Startzeiten ist der erzeugte Schedule sehr unflexibel: dynamische Tasks können nur bedingt hinzugefügt werden. Auf der anderen Seite ist der Algorithmus bezüglich Zeit- und Speicherkomplexität sehr genügsam.

Ein verallgemeinertes Modell stellen die Autoren in [CA95] vor, bei dem Mehrprozessorsysteme betrachtet werden. Neben den relativen Timing-Constraints zwischen konsekutiven Task-Instanzen können dabei auch Kommunikations- und Latenzzeiten von verschiedenen Jobs verwendet werden. Die Verteilung der Jobs auf die Prozessoren erfolgt iterativ und mit Hilfe des *Simulated Annealing* (siehe Abschnitt 5.2.1).

2.2 "Parametric Dispatching" Methode

Saksena *et al.* verfolgen in [S94], [SGA93] und [GPS95] eine parametrische Strategie. Basierend auf den spezifischen Parametern (etwa: Start- und Ausführungszeiten) werden die Constraints gegeben, siehe etwa Abbildung 2.4. Es gibt dabei keinerlei Beschränkungen auf Standard-Constraints oder sonstige Untergruppen. Durch diese Freiheit können auch parametrisierte und damit relative Constraints einfach formalisiert werden.

Sämtliche so spezifizierte Tasks werden zusammen als *Transaktion* bezeichnet. Dadurch wird auch ausgedrückt, daß es sich nicht um zyklische Probleme handelt sondern um einmalige Abläufe. Jede Transaktion wird auf Kommando ausgelöst und es gilt nun, die exakten Startzeiten der dazugehörenden Tasks festzulegen. Dabei sollen die exakten Ausführungszeiten der einzelnen Aufgaben berücksichtigt werden, die vorher nur in Form von Intervallen abgeschätzt werden konnten: Minimale und maximale Exekutionszeit je Task werden mit $[l_i,\ u_i]$ angegeben.

Neben den oben beschriebenen temporalen Constraints ist auch die Ordnung der Tasks a-priori vorgegeben. Das Problem besteht nur darin, zur Laufzeit den jeweils nächsten Task *so* zu starten, daß auch im weiteren Verlauf alle Beschränkungen eingehalten werden können. Dazu werden die Startzeiten s_i dynamisch auf Grundlage der gemessenen (exakten) Ausführungszeiten e_i zur Laufzeit berechnet.

Ausgehend von dieser Darstellungsweise wird die Schedulability definiert als

$$\exists s_1\ \forall e_1 \in [l_1, u_1]\ \cdots\ \exists s_n\ \forall e_n \in [l_n, u_n]\ C,$$

wobei C für das Constraint Set ähnlich zu Abbildung 2.4 steht.

In dieser Bedingung werden durch die *Fourier-Motzkin*'sche Methode zur Variablenelimination die Existenz- und Allquantoren sukzessive beseitigt. Die dabei entstehenden Skolemisierungs-Funktionen (siehe [RN95]) definieren die Zeitfenster für den Start der Taskinstanzen. Als Parameter werden sämtliche bereits vergangenen Taskinstanzen, d.h. deren Startzeit und die tatsächliche Ausführungszeit, herangezogen (Abbildung 2.5).

Funktionsprinzip *(Offline / Online)*	●	●[a]
Echtzeitverhalten *(Hard / Soft Realtime System)*	●	
Ausführungsumgebung *(Single / Multiple Processor)*	●	
Tasktypen *(Periodisch / Sporadisch bzw. einmalig)*	●	
Ausführungszeiten *(Exakt / Beschränkt)*		●
Release-Zeiten *(Absolut / Relativ)*	●	●
Deadlines *(Absolut / Relativ)*	●	●
Unterstützung Aperiodischer Tasks *(Offline/Online)*		●

[a] Zur Laufzeit sind Variationen der Startzeit innerhalb eines dynamisch berechneten Intervalles möglich

Abbildung 2.3: *Klassifizierender Überblick über "Parametric Dispatching" Methode aus [SGA93]*

$s_4 + e_4$	$\leq$	56	(Task 4 endet spätestens bei $t = 56$)
$s_4 + e_4$	$\leq$	$s_3 + e_3 + 12$	(Task 4 endet spätestens 12 Zeiteinheiten nach Task 3)
$s_2 + e_2 + 18$	$\leq$	s_4	(Task 4 beginnt frühestens 18 Zeiteinheiten nach Ende von Task 2)
	$\ldots$		

Abbildung 2.4: *Allgemeine Constraints*

$$
\begin{array}{rcccl}
F_1^{\min}() & \leq & s_1 & \leq & F_1^{\max}() \\
F_2^{\min}(s_1, e_1) & \leq & s_2 & \leq & F_2^{\max}(s_1, e_1) \\
 & & \vdots & & \vdots \\
F_n^{\min}(s_1, e_1, \ \ldots, \ s_{n-1}, e_{n.-1}) & \leq & s_n & \leq & F_n^{\max}(s_1, e_1, \ \ldots, \ s_{n-1}, e_{n-1})
\end{array}
$$

Abbildung 2.5: *Parametric Calendar*

Angewendet auf beliebige Constraints besitzt dieser Algorithmus eine exponentielle Zeitkomplexität. Werden die Bedingungen dagegen auf sogenannte *Standard Constraints* eingeschränkt, d.h. Constraints ähnlich zu den in dieser Arbeit verwendeten Bedingungen, ergibt sich eine Komplexität von $O(n^3)$ für das Offline-Scheduling und $O(n)$ während des Online Task Dispatchings.

Weiterhin wird in [SGA93] das Schedulen von mehreren Transaktionen kurz betrachtet. Der Einfachheit halber wird dabei von zwei Transaktionen, welche in sich jeweils schedulbar sein müssen, ausgegangen. Die Constraints werden weiter auf *Restricted Standard Constraints* eingeschränkt, Details dazu siehe [SGA93]. Für dermaßen eingeschränkte Problemfelder werden dann weitergehende Schedulability-Bedingungen angegeben. Wegen der geringen Relevanz für das Planen von unendlichen Taskfolgen soll auf diese jedoch nicht näher eingegangen werden.

Bemerkung Das vorgestellte Verfahren ermöglicht einen sehr flexiblen Umgang mit

abschätzbaren Rechenzeiten. Zur Laufzeit kann jederzeit ermittelt werden, wieviel Zeit maximal für einen aperiodischen Task (u.U. etwa einen unterbrechbaren Task) eingeplant werden kann, ohne dabei die Schedulability zu verletzen. Dabei wird nicht nur der jeweils nächste Task in die Berechnung mit einbezogen, sondern implizit der gesamte Schedule betrachtet: Durch die *Fourier-Motzkin*-Elimination ergeben sich die Startfenster genau so, daß auch im schlechtesten Fall mit maximalen Ausführungszeiten die Planbarkeit gewährleistet ist. Als Manko muß dagegen gesehen werden, daß die Reihenfolge der Taskausführungen a-priori festgelegt ist. Dazu muß bereits bei der Formulierung des Problemes entweder eine Vorentscheidung getroffen oder aber auf vom Problem vorgegebene natürliche Ordnungen zurückgegriffen werden. Außerdem läßt sich der Parametric Calendar nur schwer auf das hier betrachtete Problem der unbeschränkten Taskfolgen übertragen, da dann die Zyklenlänge fest vorgegeben und die Verbindungsstelle von Zyklenende und -anfang durch eigene Constraints definiert werden müßte.

In [CA97] und [CA97b] greifen Choi und Agrawala das Parametric Dispatching auf und dynamisieren es. Dadurch können sie auch unendlichen, periodischen Jobs Rechnung tragen. Basierend auf relativen Constraints, etwa relative Releasezeiten und Deadlines, wird wiederum offline eine Intervallermittlung für die Startzeiten durchgeführt. Mit Hilfe dieses *Dynamic Cyclic Dispatching* wird dann online die exakte Startzeit bestimmt. Zur Bestimmung des Parametric Calendars muß die Länge des zyklischen Schedules vorgegeben werden. Timing Constraints können dann, wie gewohnt, für die Jobinstanzen innerhalb eines Zyklus vorgegeben werden. Zusätzlich sind Constraints auch bezüglich zweier aufeinander folgender Zyklendurchläufe möglich. Dadurch muß sichergestellt werden, daß auch an der Verbindungsstelle zwischen zwei konsekutiven Zyklen die vom Problem erforderten Constraints erfüllt sind. Die Dissertation [Cho97] schließlich kombiniert das Dynamic Dispatching mittels eines Parametric Calendars mit dem Scheduling von aperiodischen oder einmaligen Tasks. Diese dynamischen Tasks sind nicht-unterbrechbar und werden mittels einer EDF-Strategie (*Earliest Deadline First*) verplant, sofern der Parametric Calendar dies gestattet.

2.3 "Slack Time Vector" Methode

In [Eck99] wird unter anderem ein Branch-and-Bound-Algorithmus zum Scheduling von Tasks mit relativen Constraints, jedoch ohne Release-Beschränkung, vorgestellt. Daraus ergibt sich auch, daß in diesem Sonderfall $\delta_i = \hat{d}_i$ gelten muß.

Wie beim "Dynamic Dispatching"-Modell, Abschnitt 2.2, werden offline die exakten Startzeiten der Instanzen berechnet. Dabei wird jedoch die Länge des Zyklus nicht a-priori ermittelt und dieser dann mit Instanzen gefüllt, sondern es wird jeder Schedule auf Zyklenschluß geprüft. Dadurch wird sichergestellt, daß für jedes planbare Job-Set auch ein Zyklus gefunden wird und nicht etwa bei ungünstigem LCM ein Set zurückgewiesen wird.

Der Vorteil gegenüber Jobs mit *konstanten* vorgegebenen Perioden ist in einer geringeren Prozessorlast begründet. Obwohl zu jedem relativen Timing-Modell ein ähnliches periodisches Modell aufgestellt werden kann, ist es möglich, daß sich die Prozessorlast schon bei 2-Job-Problemen fast verdoppelt. Auch kann durch die Zuweisung von konstanten Perioden die Schedulability eines Job-Sets verlorengehen oder das spätere Hinzunehmen von aperiodischen Tasks erschwert werden. Umgekehrt jedoch kann zu jedem periodischen Problem trivialerweise ein äquivalentes relatives Problem angegeben werden ($\delta_i = 0$, dann jedoch $r_i \geq 0$), so daß alle Erkenntnisse und Verfahren auch direkt auf periodische Scheduling-Aufgaben übertragbar sind.

In dem verwendeten Modell mit relativen Constraints lassen sich Schedules durch eine *Sequenz* von Jobs spezifizieren:

$$S \quad = \quad (J_1, J_2, J_1, J_3)$$

etwa definiert einen Ablaufplan, bei dem auf eine Instanz von Job 1 eine Job-2-Instanz folgt, abgelöst wieder von einer weiteren 1-er- und 3-er-Instanz. Dabei treten keine unnötigen Verzögerungen auf, der nächste Job des Schedules wird sofort ausgeführt, sowie er freigegeben (released) wird. Bei Scheduling-Problemen ohne Release-Zeiten existiert für jeden gültigen Schedule ein äquivalenter Schedule in dieser Darstellung (siehe unten, Satz 3.2). Die Begriffe *Schedule* und *Sequenz* werden daher im folgenden synonym verwendet.

Zur Darstellung von unendlichen Schedules wird auf sogenannte *partielle Schedules* zurückgegriffen; dies sind Pläne endlicher Länge, welche selbst unendlich oft wiederholt werden.

Sequenz mit Zyklus Jeder gültige Schedule S ist beschreibbar als $S = S_1\overline{S_2}$ mit S_1, S_2 partielle (d.h. endliche), gültige Schedules. Die Konkatenation (Aneinanderhängung) von Sequenzen wird dabei, wie üblich, durch $S_1 S_2$ ausgedrückt, $\overline{S_2}$ steht für die unendliche Wiederholung der Sequenz.

Die Erlangung eines solchen zyklischen Schedules im Falle relativer Deadlines und ohne Release-Zeiten wird als zumindest NP-hart bewiesen. Dazu wird ein einfaches Scheduling-Problem auf das Partitions-Problem zurückgeführt und gezeigt, daß schon in diesem Falle eine Aufteilung aller Jobs auf zwei Mengen notwendig sein kann, um die Gültigkeit eines Schedules zu erlangen.

Funktionsprinzip *(Offline / Online)*	●	
Echtzeitverhalten *(Hard / Soft Realtime System)*	●	
Ausführungsumgebung *(Single / Multiple Processor)*	●	
Tasktypen *(Periodisch / Sporadisch bzw. einmalig)*	●	○
Ausführungszeiten *(Exakt / Beschränkt)*	●	
Release-Zeiten *(Absolut / Relativ)*		
Deadlines *(Absolut / Relativ)*		●
Unterstützung Aperiodischer Tasks *(Offline/Online)*	○	

Abbildung 2.6: *Klassifizierender Überblick über die Branch-and-Bound-Methode mittels "Slack Time Vector" aus [Eck99]*

Zur Spezifikation des Branch-and-Bound-Algorithmus wird der sogenannte **Slack Time Vector** (STV) mit folgender Eigenschaft definiert: Der STV

$$s(t) \quad = \quad \begin{pmatrix} s_1(t) \\ \vdots \\ s_n(t) \end{pmatrix}$$

bedeutet, daß zum Zeitpunkt t im Schedule eine Taskinstanz für Job J_i spätestens nach $s_i(t)$ relativen Zeiteinheiten gestartet werden muß. Initial gilt daher

$$s(0) = \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix},$$

da kein Job seine Deadline überschreiten darf.

Im Laufe des Offline-Schedulings wird der STV dynamisch angepaßt, indem komponentenweise die Ausführungszeit des aktuell hinzugefügten Jobs J_j subtrahiert wird,

$$s(t+e_j) = \begin{pmatrix} s_1(t) - e_j \\ \vdots \\ s_{j-1}(t) - e_j \\ \delta_j \\ s_{j+1}(t) - e_j \\ \vdots \\ s_n(t) - e_j \end{pmatrix},$$

die Ausführungszeit also virtuell "verstreicht". Der neu aufgenommene Job bekommt als neue Slack Time natürlich seine eigene δ-Zeit zugewiesen. Gilt nach dieser Anpassung für mindestens eine Komponente $s_i(t') < 0$, $t' = t + e_j$, so hat dieser Job J_i seine Deadline verletzt und der Schedule ist nicht gültig.

Davon ausgehend kann für eine Sequenz bewiesen werden, daß sie genau dann gültig ist, wenn der STV zu keinem Zeitpunkt in der Sequenz negative Komponenten besitzt. Dabei reicht es aus, nur jene STVs zu überprüfen, die sich nach Ende eines eingeplanten Tasks ergeben.

Diese Aussage kann auch auf Sequenzen mit Zyklus, $S_1\overline{S_2}$, ausgeweitet werden: Wenn zu einem Problem eine gültige unendliche Lösung existiert, so kann auch eine gültige Sequenz in obiger Darstellung gefunden werden. Dazu wird ausgesagt, daß es zu einem Problem nur endlich viele gültige STVs geben kann, da sämtliche Deadlines endlich sind. In einem unendlichen gültigen Schedule muß also mindestens ein STV unendlich oft auftreten. Ebenso muß sich nach jeder Ausführung der Sequenz S_2 in der Zyklennotation, sofern sich jeder Job mindestens einmal in S_2 befindet, derselbe STV ergeben. Da der Übergang zwischen zwei Instanzen von S_2 zulässig ist, sich also kein negativer STV ergibt, kann dieser Teil periodisch wiederholt werden. Andersherum ergibt sich somit aber auch, daß im bekannten unendlichen Schedule der Teil zwischen zwei gleichen STVs, welcher sämtliche Jobs umfaßt, als S_2 bezeichnet werden kann.

Basierend auf dieser Eigenschaft wird ein Branch-and-Bound-Algorithmus formuliert, der einen unendlichen Schedule in Zyklennotation liefert.

2.3.1 Branch-and-Bound-Algorithmen

Branch-and-Bound-Verfahren werden besonders häufig auch im Operations Research ([DD95]) zur Lösung von linearen Optimierungsproblemen unter Nebenbedingungen (z.B. Ganzzahligkeit) eingesetzt. Nach [IBM98] oder [RN95] besteht das grundsätzliche Prinzip der Verfahren darin, einen Suchbaum in einer "günstigen" Reihenfolge zu durchlaufen.

- An einem Entscheidungsknoten (etwa: Einplanen von J_1 oder J_2 an dieser Stelle) führe man Schätzungen für den erwarteten Nutzen unter Nebenbedingungen (etwa: Schedulability) für *jede* Entscheidungsmöglichkeit durch:

  - Nebenbedingungen erfüllt, hoher erwarteter Nutzen: Speichern als bisher aussichtsreichsten Kandidaten, diesen Ast weiterverfolgen
  - Nebenbedingungen verletzt oder der erwartete Nutzen geringer als bei bisher bestem Kandidaten: **Bounding**, Ast nicht weiterverfolgen
  - Ansonsten: Keine Aussage möglich, sämtliche Nachfolger des Entscheidungsknotens müssen im Verlauf betrachtet werden. Es kommt zum **Branching**, d.h. dem Hinzufügen der neuen Kandidaten zu einer Arbeitsliste

- Solange die Arbeitsliste noch nicht leer ist, neuen Kandidaten auswählen und erneut durchführen.

Die nach dieser Struktur konstruierten Verfahren ähneln den sogenannten A^*-Verfahren (siehe [RN95]) und eignen sich besonders gut zum Durchsuchen großer Entscheidungsbäume, wenn eine Schätzung für den erwarteten Nutzen einer Entscheidung möglich ist. In diesem Falle zeichnen sie sich durch ein gutes Laufzeitverhalten, moderaten Speicherbedarf und Korrektheit aus.

2.3.2 Scheduling durch Branch-and-Bound

Angewandt auf das Scheduling-Problem wird der Branch-and-Bound-Algorithmus folgendermaßen formalisiert ([Eck99]):

Jeder Weg zu einem Entscheidungsknoten repräsentiert eine Sequenz; an jedem Knoten wird also die Entscheidung getroffen, welcher Knoten der Sequenz hinzuzufügen ist. Dadurch ergibt sich ein *Branching* der Art und Weise, daß sämtliche Jobinstanzen getestet werden müssen.

Dabei wird jeder Knoten, wie oben beschrieben, mit einem STV bewertet. Enthält dieser eine negative Komponente, so ist der zugehörige Zyklus ungültig und der Knoten muß nicht weiter betrachtet werden (*Lokales Bounding*).

Das *globale Bounding* ergibt sich durch ein Majoranten-Kriterium: Ein STV majorisiert einen anderen, wenn dieser komponentenweise größer oder zumindest gleichgroß ist. Majorisiert ein beliebiger STV s' im Baum, dessen zugehöriger Knoten im Suchbaum bereits bearbeitet wurde und der *nicht* zu einer gültigen Lösung führte, den aktuellen STV s,

$$s = \begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix} \quad \leq \quad \begin{pmatrix} s'_1 \\ \vdots \\ s'_n \end{pmatrix} = s' \tag{2.1}$$

so muß auch dieser Ast nicht weiter betrachtet werden. Dies ist anschaulich klar, da bei einem größeren STV auch ein größerer Planungsfreiraum besteht. Konnte jedoch auch bei dieser größeren Flexibilität keine gültige Sequenz gefunden werden, so kann dies im beschränkteren Falle erst recht nicht geschehen.

Das Verfahren terminiert, wenn alle Knoten bearbeitet sind (Mißerfolg, kein gültiger Schedule zu erzeugen) oder der aktuell betrachtete STV s einen anderen, auf dem Weg zu diesem Knoten entstandenen STV s' dominiert. Aus der Majorisierung folgt, daß an dieser Stelle der Zyklus geschlossen werden kann, indem der Schedule bei Knoten K' fortgesetzt wird. Die Slack Times garantieren die Fristigkeit aller Jobs. Außerdem ist klar, daß auf dem Weg von K' zu K sämtliche Jobs mindestens einmal verplant worden sind, da ansonsten die Komponenten im STV nicht hätten größer sein können – nach obiger dynamischer Anpassung des STVs sind die Komponenten, mit Ausnahme des neu hinzukommenden Jobs, monoton fallend.

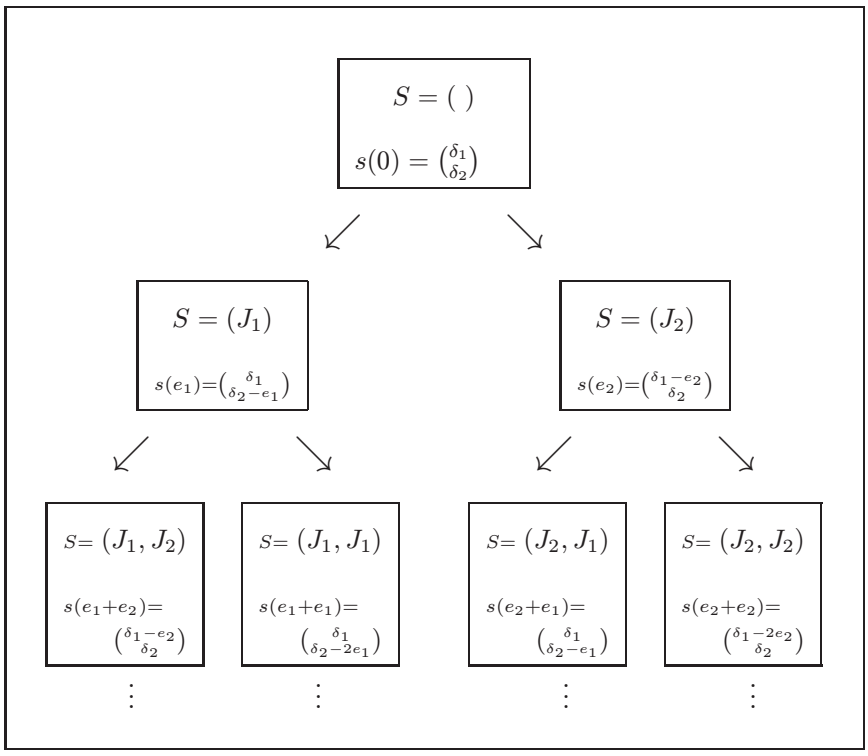

Abbildung 2.7: *Exemplarischer Branch-and-Bound Entscheidungsbaum für ein 2-Job-Problem. Dabei sind nur die ersten beiden Entscheidungsstufen aufgeführt. In den Knoten finden sich die repräsentierten Sequenzen und die zugehörigen Slack Time Vectors.*

2.3.3 Eigenschaften

Für dieses Branch-and-Bound-Verfahren wird in [Eck99] dann die Korrektheit und Vollständigkeit bewiesen: Der Algorithmus terminiert stets mit korrektem Ergebnis – er liefert genau dann eine zulässige Lösung, wenn eine solche existiert, ansonsten wird das Job-Set als nicht-planbar zurückgewiesen. Die Laufzeit wird mindestens mit $O(n^{2^n})$ (mit n als der Anzahl der Jobs) nach oben beschränkt, was aber nur für die Garantie der Termination Relevanz besitzt.

Basierend auf den so gewonnenen Schedules wird dann das inkrementelle Hinzufügen von aperiodischen Tasks behandelt. Für unterbrechbare Tasks müssen dazu im Schedule freie Intervalle identifiziert werden, in denen dann Teile des hinzugekommenen Jobs ausgeführt werden können. Dazu wird der generierte initiale Schedule bezüglich der Prozessorauslastung optimiert, so daß maximal viel Freikapazität für sporadische Tasks entsteht. Für 2-Job-Probleme ohne Releasezeiten werden einige Schedulability-Bedingungen hergeleitet und basierend darauf die für dieses Problem einzig möglichen drei Zyklentypen betrachtet. Diese werden jeweils anschließend bezüglich der Prozessorauslastung optimiert. Der optimale Schedule für das 2-Job-Problem findet sich dann unter diesen drei Optimalzyklen.

Bemerkung Das vorgestellte Verfahren ist für Probleme *ohne* Release-Constraints exakt, d.h. sämtliche schedulbaren Task-Sets werden als solche erkannt und mittels eines gültigen zyklischen Schedules dargestellt. Es muß dabei keine Einschränkung auf

eine vorher mittels LCM berechnete Zyklenlänge erfolgen, da ein Zyklenschluß automatisch und einfach, d.h. ohne großen Rechenaufwand, erkannt wird. Grundsätzliche Probleme eines vollständigen Suchverfahrens wie etwa der hohe Rechenzeitbedarf, die Unflexibilität des erzeugten Schedules oder die Abgeschlossenheit eines jeden Durchlaufes (die Erweiterung eines Schedules um neue Jobs ist nicht oder nur erschwert möglich) bleiben jedoch bestehen.

2.4 IDA^*-Algorithmen

Für das Scheduling von nicht-zyklischen Task-Sets auf Multiprozessor-Systemen wird unter [FK92] ein weiterer Suchalgorithmus vorgestellt. Der dabei verwendete *Iterative Deepening* A^* Algorithmus (siehe [RN95]) gehört zur Gruppe der heuristischen Suchverfahren. Mittels einer Schätzfunktion wird berechnet, wie weit ein ungültiger Schedule von einer gültigen Lösung "entfernt" ist. Diese Funktion dient dann zur Bewertung einzelner Schedules und zur Entscheidung darüber, an welcher Stelle im Suchbaum zuerst weitergearbeitet werden soll.

Da das verwendete Verfahren jedoch speziell auf Multiprozessor-Umgebungen abgestimmt und nicht-zyklisch ist, soll hier nicht weiter auf Einzelheiten eingegangen werden.

Kapitel 3

Theoretische Betrachtungen

Zur Vorbereitung auf die Definition eines iterativen Taskmodelles und zum tiefergehenden Verständnis werden einige theoretische Betrachtungen angestellt. Diese werden dann im nächsten Abschnitt verwendet, um möglichst optimale Modellstrukturen zu finden.

Für Mehrjobprobleme mit n Jobs $J_1, \ldots, J_n$, $J_i = (e_i, r_i, \delta_i)$, gilt es, gültige Schedules (siehe Abschnitt 1.4) zu finden. Wie üblich sind die Aufträge nicht-unterbrechbar, besitzen keine Inter-Job-Constraints und müssen auf einem harten Single-Prozessor-System ausgeführt werden.

3.1 Grundlegende Feststellungen

Trivialerweise ergibt sich für die Parametrisierung mittels der drei Zeitangaben ein sehr einfaches, *notwendiges* Schedulability-Kriterium für Probleme mit mindestens 2 Jobs:

$$\forall\, i \in \{1, \ldots, n\} \qquad \forall\, i' \in \{1, \ldots, n\},\ i' \neq i \qquad e_i \leq r_{i'} + \delta_{i'}\,.$$

Gilt diese Beziehung nicht, so ist ein Task J_i a-priori nicht planbar, da er bei einem anderen Job $J_{i'}$ nicht zwischen zwei Instanzen "paßt". Da jedoch $J_{i'}$ im Zyklus enthalten sein muß und keinen ausreichenden Zwischenraum für J_i-Instanzen bietet, kann das Job-Set nicht planbar sein.

Weiterhin muß bei Verwendung von Deadline-Zeiten klarerweise gelten, daß mit

$$\begin{array}{rrcll} & e_i & \leq & d_i & = r_i + \delta_i + e_i, \\ & r_i & \leq & d_i - e_i & = r_i + \delta_i \\ \Longrightarrow & e_i + r_i & \leq & d_i & \end{array}$$

die Deadline ausreichend groß ist. Entsprechend muß an eine alternativ vorgegebene Slack Time die Bedingung

$$r_i \quad \leq \quad \hat{d}_i \qquad = r_i + \delta_i$$

erfüllt sein, da ansonsten bereits die Definition des einzelnen Jobs inkonsistent wäre. Bei Verwendung der hier favorisierten δ-Notation mit ganzzahligen, nichtnegativen Werten treten solche Inkonsistenz-Problem jedoch gar nicht erst auf.

Die geforderte Einschränkung auf natürliche Zeitangaben läßt sich einfach auf *rationale Zahlen* erweitern und schränkt deshalb die Menge der spezifizierbaren Probleme nur unwesentlich ein. Für die als Quotient gegebenen Zeitangaben läßt sich ein

gemeinsamer Nenner finden (euklidscher Algorithmus, natürliche Zahl) und das gesamte Problem mit diesem Wert skalieren. Dabei muß die **Skalierungs-Invarianz** ausgenutzt werden: Das verwendete Modell ist linear in allen Komponenten und somit invariant bezüglich Skalierung mit positiven ganzzahligen Faktoren. Bei negativen Faktoren führen die Constraint-Ungleichungen zu Problemen, außerdem ergeben negative Ausführungszeiten nur wenig Sinn. Nicht-ganzzahlige Skalierungsfaktoren dagegen beeinflussen die Ganzzahligkeit des Problems: Da viele Aussagen auf der Betrachtung von ganzzahligen Randbedingungen basieren, kann dies zu einem Zusammenfallen von vorher nicht identischen Zeitangaben und damit ungültigen Ergebnissen führen. Daher kann die Skalierbarkeit nur für aus natürlichen Zahlen bestehende Faktoren gewährleistet werden.

3.2 Task-Bezogene Aussagen

3.2.1 Synchronisations-Tasks

Satz 3.1 Für jeden gültigen partiellen Schedule, der um eine weitere Taskinstanz erweitert werden soll, gilt: Mindestens eine der unmittelbar nächsten Instanzen der n Tasks kann zu Beginn des entsprechenden Start Time Windows eingeplant werden.

Beweis Der letzte Task im Schedule ende zum Zeitpunkt t_0. Annahme: Es gibt nur solche Schedules, so daß alle unmittelbar folgenden Taskinstanzen (J_i') der n Tasks mindestens eine Zeiteinheit nach Beginn ihrer Start Time Windows eingeplant werden. Somit können sämtliche Taskinstanzen nach dem Zeitpunkt t_0 um eine Zeiteinheit vorverlegt werden: Bei den unmittelbar folgenden Instanzen J_i' ist dies aufgrund der Annahme des früher beginnenden Start Time Windows möglich. Durch die Verschiebung ändern sich auch sämtliche relativen Constraints zu späteren Instanzen (J_i''), so daß auch diese verschoben werden können. Somit ist die Annahme falsch und mindestens ein Task beginnt zu Anfang des Start Time Windows. ■

Bemerkung Durch die Verplanung einer der unmittelbar folgenden Instanzen zu Beginn des entsprechenden STWs können relative Constraints zu absoluten Constraints werden: Andere Taskinstanzen können an dem festgelegten Task ausgerichtet werden; der an dieser Stelle festgelegte Task wird daher auch als **Synchronisations-Task** bezeichnet.

Bezogen auf 2-Job-Probleme bedeutet dies auch, daß für jedes planbare Taskset ein Schedule definiert werden kann, der die unmittelbare Abfolge J_1, J_2 ohne dazwischenliegende Idlezeit enthält.

3.2.2 Notwendigkeit impliziter Idle-Tasks

Bei Verwendung von Releasezeiten ergeben sich unter Umständen Schedules, welche den Prozessor nicht lückenlos auslasten. Wie in der Technik üblich, wird der Prozessor in diesen Zeiten nicht deaktiviert sondern arbeitet eine Reihe von *NOP*[1]-Instructions ab. Beim Scheduling werden solche Phasen auch als **Idle-Tasks** bezeichnet, dies sind Tasks, in denen der Prozessor mit dem "Nichtstun" beschäftigt ist.

Bei Darstellung der Schedules in Listenform (etwa $(J_1, J_2, J_1, J_3, \ldots)$) ergeben solche Idle-Tasks in gewissen Fällen *implizit*, da die Release-Zeit für den nächsten zu startenden Job noch nicht verstrichen ist. Oft ist es dabei auch nicht möglich, Sche-

[1] NOP: No Operation

dules *ohne* solche Idle-Intervalle zu erzeugen. Beispielsweise führt die Probleminstanz mit zwei Jobs und

$$n = 2; \quad e_1 = 1; \quad e_2 = 3; \quad r_1 = 4; \quad r_2 = 0; \quad \delta_1 = 1; \quad \delta_2 = 4$$

nach Planung von J_1, J_2 zu folgender Konstellation (Abbildung 3.1): Da nur J_2

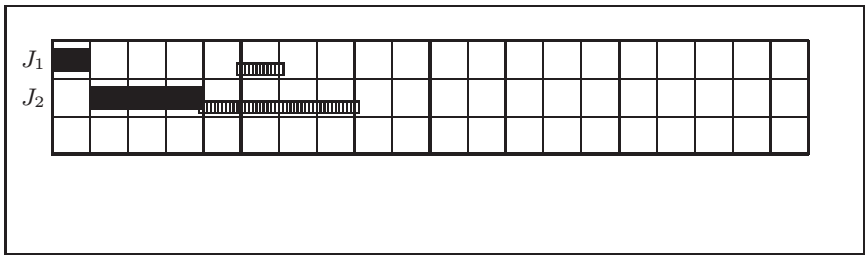

Abbildung 3.1: *Notwendigkeit der Planung eines Idle-Tasks zur Erzeugung eines gültigen Schedules*

anschließend freigegeben ist, müßte, wenn keine Idletasks auftreten dürfen, J_2 ein zweites Mal verplant werden. Dies würde jedoch sofort eine Deadline-Überschreitung bei J_1 bewirken, womit dieser Schedule-Ansatz verworfen würde.

Soll jedoch eine weitere Instanz von J_1 angefügt werden, muß ein Idle-Task der Länge 1 eingeschoben werden. Der mögliche zyklische Schedule ergibt sich dann umgehend in Abbildung 3.2.

Diese Art von Idle-Tasks werden als *Implizite Idle-Tasks* bezeichnet, da sie exakt die Zeit bis zum Beginn des Start Time Windows abdecken. Allein aus der Angabe, welcher Task als nächster zu starten ist, kann implizit erkannt werden, ob vorher eine Idle-Zeit abzuwarten ist. Im Scheduling-Prozeß muß dieser Typus von Idle-Task daher nicht explizit verplant werden.

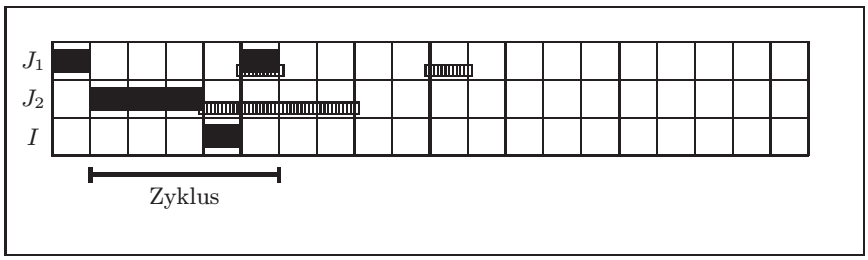

Abbildung 3.2: *Nach Einfügen des Idle-Tasks I ergibt sich ein gültiger zyklischer Schedule*

Satz 3.2 Für Problem-Sets *ohne* Release-Zeiten gilt damit insbesondere, daß zu jedem planbaren Problem ein *dichter Schedule*, d.h. ein Schedule ohne Idle-Zeiten, existiert.

Beweis Nach Satz 3.1 kann stets mindestens eine Instanz zu Beginn des Start Time Windows eingeplant werden. In der Abwesenheit von Release-Zeiten ist aber immer jeder Task sofort verfügbar, so daß jede Instanz sofort beginnen könnte. Gibt es daher einen gültigen Schedule, so kann durch das oben im Beweis aufgezeigte Linksverschieben ein dichter Schedule erzeugt werden. ■

3.2.3 Notwendigkeit expliziter Idle-Tasks

Die alleinige Verwendung impliziter Idle-Tasks zur Überbrückung der Zeiträume bis zu einem Release genügt im allgemeinen nicht, um sämtliche möglichen Schedules zu erzeugen. Für zwei Jobs mit

$$n = 2; \quad e_1 = 1; \quad e_2 = 3; \quad r_1 = 2; \quad r_2 = 5; \quad \delta_1 = 1; \quad \delta_2 = 1$$

werde ohne Beschränkung der Allgemeinheit beginnend mit (J_1, J_2, J_1) geplant (Abbildung 3.3). Der umgehende Start von J_1 nach Verstreichen der Release Time (Abbildung 3.4) führt später zur eigenen Deadline-Verletzung. Eine Planung von J_1 jedoch erst eine Zeiteinheit nach der Release-Zeit (sozusagen das "unnötig" lange Warten) führt schließlich aber doch zu einem gültigen Schedule (Abbildung 3.5).

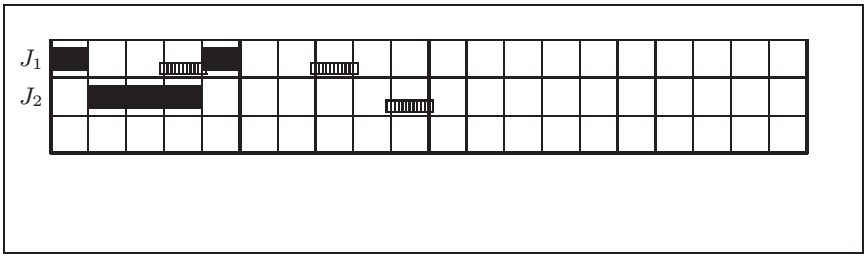

Abbildung 3.3: *Allein das Verplanen von trivialen Idle-Tasks bis zum Release eines gewissen Jobs genügt normalerweise nicht, um sämtliche Job-Sets zu schedulen. Startsituation, mögliche Fortsetzungen in Abbildungen 3.4 und 3.5*

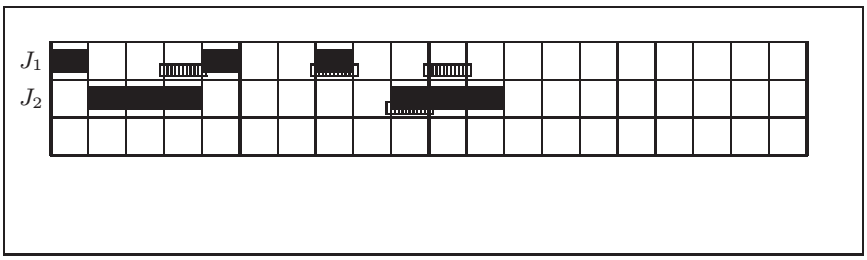

Abbildung 3.4: *Wird J_1 sofort zu Beginn seines Start Time Windows gestartet, so führt das anschließend notwendige Starten von J_2 nach dem Release zu einer Deadline-Überschreitung von J_1.*

Dadurch ergibt sich umgehend

Satz 3.3 Für Problem-Sets mit nichttrivialen Release-Zeiten müssen Idle-Zeiten während des Schedulings explizit beachtet werden, um das Finden eines gültigen Schedules garantieren zu können. ■

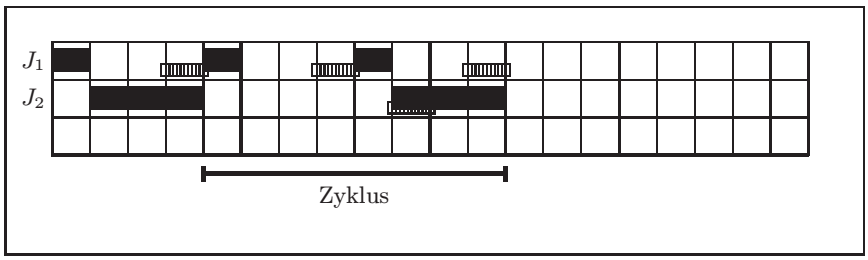

Abbildung 3.5: *Wird jedoch vor dem Start von J_1 ein längerer Idle-Task eingeführt, so daß J_1 erst zum Ende seines Start Time Windows eingeplant wird, ergibt sich ein gültiger, zyklischer Schedule.*

3.3 Zyklus-Bezogene Aussagen

3.3.1 Normierbarkeit

Satz 3.4 Jeder gültige zyklische Schedule kann so repräsentiert werden, daß er direkt (keine vorhergehenden expliziten/impliziten Idle-Zeiten) mit einer Taskinstanz eines beliebigen Jobs beginnt oder endet.

Beweis Aufgrund der Definition eines Zyklus ist es klar, daß dieser rotiert werden kann, d.h.

$$\overline{(J_1 J_2 \ \ldots \ J_n)} \quad = \quad J_1 \, \overline{(J_2 J_3 \ \ldots \ J_n J_1)}.$$

Da für ein Scheduling-Problem der Startvorgang vernachlässigt werden kann und nur die unendliche Wiederholung des zyklischen Teiles relevant ist, kann der vorgezogene J_1 entfallen. Durch Rotation kann somit jede im Zyklus enthaltene Taskinstanz an den Anfang verschoben werden. Da jeder Task mindestens einmal enthalten sein muß, ergibt sich die Aussage des Satzes. Explizite Idle-Zeiten müssen dabei genau wie Jobs rotiert werden, implizite Idles ergeben sich automatisch. Vor einem Task notwendige implizite Idle-Times werden durch einen verspäteten Release des Tasks bedingt und sind somit von der Rotation nicht extra betroffen. ■

Zur *Normierung* kann somit jeder Zyklus so dargestellt werden, daß er etwa direkt mit einer Instanz von J_1 beginnt.

3.3.2 Zyklische Schedules

Satz 3.5 Zu jedem planbaren Job-Set von periodischen Jobs existiert ein unendlicher Schedule in zyklischer Form.

Beweis Ergibt sich für den Fall von Job-Sets ohne Release-Zeiten unmittelbar aus dem in [Eck99] vorgestellten Branch-and-Bound-Algorithmus. Die Anzahl der Slack Time Vectors ist beschränkt und somit wird im Falle der Schedulability früher oder später ein Zyklus gefunden.

Die Spezialisierung des Problemes durch die Vergabe von Release-Zeiten schränkt den Lösungsraum ein. Von den im Branch-and-Bound-Verfahren gefundenen zyklischen Schedules erfüllen nur einige die Release-Constraints. Da es sich um eine Einschränkung der Lösungsmenge handelt, wird ein die Constraints erfüllender Zyklus somit auf jeden Fall gefunden. ■

Bemerkung Die Menge der zyklischen Schedules schöpft damit den unendlichen Lösungsraum des vorgestellten Scheduling-Problems mit relativen Constraints aus.

3.4 2-Job-Probleme

Die Betrachtung von 2-Job-Problemen sieht auf den ersten Blick relativ "trivial" aus. Es zeigt sich jedoch, daß sich auch im Falle $n = 2$ schon interessante Resultate erzielen lassen. Im folgenden werden dazu Kriterien erarbeitet, die die Schedulability von 2-Job-Problemen angeben. Die Erkenntnisse lassen sich dann auch auf Mehrjob-Probleme (Abschnitt 3.5.2) anwenden.

3.4.1 Einpassungs-Kriterium (bei $r_1 = r_2 = 0$)

Satz 3.6 Zwei Jobs $(e_1, 0, \delta_1)$, $(e_2, 0, \delta_2)$ sind genau dann planbar, wenn das *Einpassungs-Kriterium* gilt:

$$\begin{array}{lrcl} & e_1 & \leq & \delta_2 \\ \wedge & e_2 & \leq & \delta_1 \end{array}$$

Beweis Gilt entweder $e_1 > \delta_2$ oder $e_2 > \delta_1$, so überdeckt eine Taskinstanz das gesamte Starttime-Window des anderen Tasks, dieser kann also nicht im Schedule verplant werden.

Andererseits kann für beliebige $e_1, \delta_1, e_2, \delta_2$ ein trivialer, gültiger Schedule angegeben werden: (J_1, J_2) ist ohne Idle-Zeiten gültig. Nach der Ausführung einer Instanz ist das Ende des Starttime-Window des anderen Tasks noch nicht erreicht. ∎

Das Kriterium läßt sich auch in folgender Form darstellen:

$$\begin{array}{lrcll} & e_1 & \leq & \delta_2 = d_2 - e_2 & \\ \wedge & e_2 & \leq & \delta_1 = d_1 - e_1 & \\ \Longleftrightarrow & e_1 + e_2 & \leq & d_1 & (3.1) \\ \wedge & e_1 + e_2 & \leq & d_2 & (3.2) \end{array}$$

3.4.2 Notwendiges Utilisation-Kriterium (bei $r_1 = r_2 = 0$)

Satz 3.7 Das *Utilisation-Kriterium* (etwa nach [Eck99])

$$U = \frac{e_1}{d_1} + \frac{e_2}{d_2} \leq 1 \qquad (3.3)$$

stellt eine notwendige, jedoch nicht hinreichende Bedingung für die Planbarkeit eines 2-Job-Problems dar. Zu beachten ist dabei die Verwendung von Deadlines d_1, d_2 anstelle von Slack-Times (siehe 1.4):

$$\begin{array}{rcl} d_1 & = & \delta_1 + e_1 \\ d_2 & = & \delta_2 + e_2 \end{array}$$

Beweis Die Aussage folgt aus Satz 3.6. Bei einem schedulbaren Taskset sei o.B.d.A. $d_1 \leq d_2$, ansonsten erfolge Tausch der Taskparameter.

$$\begin{aligned} e_1 d_2 + e_2 d_1 &= (e_1 + e_2) d_2 - e_2 d_2 + e_2 d_1 \\ &\overset{(3.1)}{\leq} d_1 d_2 + e_2 \underbrace{(d_1 - d_2)}_{\leq 0} \\ &\leq d_1 d_2 \\ \Longrightarrow \quad \frac{e_1 d_2 + e_2 d_1}{d_1 d_2} = \frac{e_1}{d_1} + \frac{e_2}{d_2} &\leq 1 \quad \Longrightarrow (3.3) \end{aligned}$$

■

Bemerkung Die Bedingung ist nicht hinreichend, da etwa der Fall $(1, 0, 1)$, $(2, 0, 10)$ zu einer gültigen Utilisation-Bedingung ($\frac{1}{2} + \frac{2}{12} \leq 1$) führt. Trivialerweise existiert jedoch kein gültiger Schedule, da zwischen zwei J_1-Instanzen nur maximal 1 Zeiteinheit verfügbar ist, J_2 jedoch 2 Zeiteinheiten für die Ausführung benötigt.

3.4.3 Minimale Utilisation bei $r_1 = r_2 = 0$

Satz 3.8 Nach [Eck99] können zwei Jobs ohne Release-Beschränkungen genau dann mit Perioden $e_1 + \delta_1$ respektive $e_2 + \delta_2$ verplant werden, wenn

$$e = e_1 + e_2 \quad \leq \quad \mathrm{ggT}(\delta_1 + e_1,\ \delta_2 + e_2)$$

gilt. In diesem Falle wird eine minimale Prozessorauslastung erreicht, da sämtliche Tasks erst am Ende ihres Start Time Windows ausgeführt werden.

Beweis Siehe [Eck99]

3.4.4 Kriterium bei $r_1 = 0$

Satz 3.9 Ein 2-er-Taskset mit Parametern $(e_1, 0, \delta_1)$, (e_2, r_2, δ_2) ist genau dann planbar, wenn gilt:

$$\left\lceil \frac{e_2 + r_2}{e_1 + \delta_1} \right\rceil \quad \leq \quad \left\lfloor \frac{r_2 + \delta_2}{e_1} \right\rfloor \tag{3.4}$$

Dabei bezeichnen $\lceil\ \rceil$ und $\lfloor\ \rfloor$ die nächstgrößere respektive nächstkleinere ganze Zahl.

Beweis Trivialerweise müssen die Ausführungszeiten kürzer als die maximalen Deadline des jeweils anderen Jobs sein:

$$e_1 \quad \leq \quad r_2 + \delta_2 \tag{3.5}$$

$$e_2 \quad \leq \quad \delta_1 \tag{3.6}$$

Nach Satz 3.1 und Satz 3.4 ist es stets möglich, einen gültigen Schedule in der Form aus Abbildung 3.6 darzustellen, d. h. daß der Schedule durch die Synchronisations-Taskfolge begonnen wird.

Jeder Zyklus werde somit von der unmittelbaren Abfolge von J_1 und J_2 begonnen. Weiterhin gibt es einen minimalen Zyklus mit genau einer J_2-Instanz, da $r_1 = 0$ gilt und J_1 somit stets eingeplant werden kann. Der Rest des Zyklus muß mit weiteren J_1-Instanzen aufgefüllt werden, so daß nach der letzten Instanz wieder J_2 unmittelbar beginnen kann. Die Anzahl dieser Zwischeninstanzen ist variabel und kann zwischen 1 und $\frac{r_2 + \delta_2}{e_1}$ liegen.

Ist das Taskset planbar, so gibt es ein $n \geq 1$, wobei n die Anzahl der J_1-Zwischeninstanzen bestimmt. Der zyklische Schedule hat dann die Form $J_1, J_2, (J_1)^{n-1}$ oder

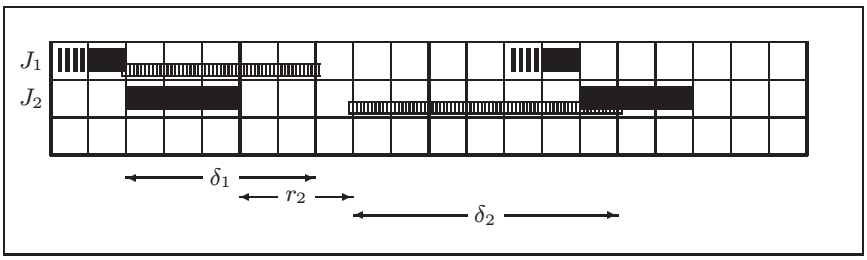

Abbildung 3.6: *Standardsituation des unmittelbaren Starts von J_2 nach Ende einer J_1-Instanz*

auch einfach $(J_1)^n, J_2$. Die Idle-Zeiten zwischen den einzelnen Instanzen sind dabei variabel; zwischen den J_1-Instanzen können jeweils Idlephasen der Länge $0 - \delta_1$ verplant werden, ohne die Constraints zu verletzen.

Für das Kriterium der Schedulability gilt es nun zu verifizieren, ob es ein n gibt, so daß mit Hilfe von n Instanzen von J_1 und entsprechenden Idlezeiten die Wartezeit bis zum Release von J_2 überbrückt werden kann und J_2 trotzdem noch termingerecht innerhalb des Starttime-Windows gestartet werden kann:

Für gegebenes n bezeichne w die einzuplanende Idle-Wartezeit, welche durch n Instanzen von J_1 überbrückt werden kann. Für w gilt:

$$w \quad \in \quad [e_2,\ e_2+1,\ e_2+2,\ \ldots,\ n\,\delta_1] \tag{3.7}$$

Die untere Schranke rührt daher, daß zumindest die Ausführungszeit von J_2 abgewartet werden muß. Mit einem solchen Paar (n, w) kann ein gültiger Schedule dann angegeben werden, wenn durch

$$n\,e_1 + w \quad \in \quad [\,e_2 + r_2,\ \ldots,\ e_2 + r_2 + \delta_2] \tag{3.8}$$

sichergestellt wird, daß nach Ausführung von n Instanzen die letzte Ausführung von J_1 innerhalb des Starttime-Windows von J_2 endet. Von der Wartezeit w sind dabei mindestens e_2 Zeiteinheiten während der Ausführung von J_2 einzuplanen, wegen $w \geq e_2$ ist dies immer möglich. Der Rest der Wartezeit kann wegen $w \leq n \cdot \delta_1$ dann beliebig zwischen die anderen J_1-Instanzen verteilt werden, ohne dabei die Deadline-Kriterien zu verletzen. Der Zyklus ist somit vollständig; der nächste Durchlauf kann mit J_2 begonnen werden, da die Releasezeit verstrichen ist.

Um eine geschlossene Repräsentation des Kriteriums zu ermöglichen, müssen die Existenzquantoren in

$$\exists\ n \geq 1 \quad \exists\ w \in [e_2, \ldots,\ n\,\delta_1] \qquad n\,e_1 + w \in [\,p_2 + r_2, \ldots,\ p_2 + r_2 + \delta_2] \tag{3.9}$$

beseitigt werden. w wird dazu durch folgende Form parametrisiert:

$$\begin{aligned}
& & w &= e_2 + \lambda(n\,\delta_1 - e_2), \qquad \lambda \in [0,1] \\
\Longrightarrow & & n\,e_1 + w &= n\,e_1 + e_2 + \lambda(n\,\delta_1 - e_2) \\
\overset{(3.8)}{\Longrightarrow} & n\,e_1 + e_2 + \lambda(n\,\delta_1 - e_2) & &\in [e_2 + r_2,\ \ldots,\ e_2 + r_2 + \delta_2] \\
\Longrightarrow & n\,e_1 + \lambda(n\,\delta_1 - e_2) & &\in [r_2,\ \ldots,\ r_2 + \delta_2]
\end{aligned}$$

Dadurch ergeben sich untere und obere Schranke für λ, insofern folgende Quotienten definiert sind:

$$\begin{aligned} & n\,e_1 + \lambda(n\,\delta_1 - e_2) \;\geq\; r_2 && (3.10)\\ \Longrightarrow\quad & \lambda \;\geq\; \tfrac{r_2 - n\,e_1}{n\,\delta_1 - e_2} \quad =: \lambda_{\min} \end{aligned}$$

$$\begin{aligned} & n\,e_1 + \lambda(n\,\delta_1 - e_2) \;\leq\; r_2 + \delta_2 && (3.11)\\ \Longrightarrow\quad & \lambda \;\leq\; \tfrac{\delta_2 + r_2 - n\,e_1}{n\,\delta_1 - e_2} \quad =: \lambda_{\max} \end{aligned}$$

Die Wahl eines $0 \leq \lambda \leq 1$ ist also genau dann möglich, wenn sich die Intervalle $[0,\ \ldots,\ 1]$ und $[\lambda_{min},\ \ldots,\ \lambda_{\max}]$ überschneiden, also

$$\lambda_{\min} \leq \lambda_{\max} \quad \wedge \quad \lambda_{\max} \geq 0 \quad \wedge \quad \lambda_{\min} \leq 1 \qquad (3.12)$$

gilt. Zu dem vorgegebenen n kann in diesem Falle das zugehörige w berechnet werden, so daß ein gültiger Zyklus erstellt werden kann. Die Ganzzahligkeit von w ist hier nicht erforderlich, da auch mit theoretisch nicht-ganzzahligen Wartezeiten gerechnet werden kann. Praktisch würde die Instanz mit der nächstkleineren ganzzahligen Wartezeit eingeplant werden und der nichtganzzahlige Rest kumuliert. Bei Erreichen einer Restesumme über 1 würde eine zusätzliche Warteeinheit eingeschoben.

Im Falle von $n\,\delta_1 = e_2$ ist w eindeutig spezifiziert ($w = e_2$) und unabhängig von λ:

$n\,\delta_1 = e_2$:

$$\begin{aligned} & & w \;&=\; e_2 = n\,\delta_1 = \text{const.} \quad \wedge \quad n \geq 1 \\ \overset{(3.6)}{\Longrightarrow}\quad & & n \;&=\; 1 \\ \Longrightarrow\quad & (3.10) \;&\Leftrightarrow\; & e_1 \geq r_2 \\ \wedge\quad & (3.11) \;&\Leftrightarrow\; & e_1 \leq \delta_2 + r_2 \\ \Longleftrightarrow\quad & (3.12) \;&\Leftrightarrow\; & r_2 \leq e_1 \leq \delta_2 + r_2 \qquad (3.13) \end{aligned}$$

Ansonsten gilt aufgrund (3.6), daß $n\,\delta_1 > e_2$:

$n\,\delta_1 > e_2$:

$$\begin{aligned} (3.12) \;&\Leftrightarrow\; & \tfrac{r_2 - n\,e_1}{n\,\delta_1 - e_2} &\leq \tfrac{\delta_2 + r_2 - n\,e_1}{n\,\delta_1 - e_2} \\ & \wedge\; & \tfrac{\delta_2 + r_2 - n\,e_1}{n\,\delta_1 - e_2} &\geq 0 \\ & \wedge\; & \tfrac{r_2 - n\,e_1}{n\,\delta_1 - e_2} &\leq 1 \\ \Longleftrightarrow\quad (3.12) \;&\Leftrightarrow\; & \delta_2 + r_2 \geq n\,e_1 \quad \wedge \quad & r_2 + e_2 \leq n\,(e_1 + \delta_1) \qquad (3.14) \end{aligned}$$

Die Schedulability-Bedingung (3.9) ergibt sich somit zu

$$\begin{aligned} (3.9) \quad \Leftrightarrow \quad & (\ e_2 = \delta_1 \quad \wedge \quad r_2 \leq e_1 \leq r_2 + \delta_2\ ) \quad \vee \qquad (3.15)\\ & (\ e_2 < \delta_1 \quad \wedge \quad \exists\, n \geq 1 \text{ mit } (3.14)\ )\,. \end{aligned}$$

Für das n ergeben sich aus (3.14) folgende Schranken:

$$\begin{aligned} & \tfrac{\delta_2 + r_2}{e_1} \geq n \qquad \wedge \qquad n \geq \tfrac{r_2 + e_2}{e_1 + \delta_1} \\ \Longleftrightarrow\quad & \tfrac{r_2 + e_2}{e_1 + \delta_1} \;\leq\; n \;\leq\; \tfrac{\delta_2 + r_2}{e_1} \end{aligned}$$

Da n ganzzahlig sein muß, kann solch ein Wert genau dann gefunden werden, wenn

$$\left\lceil \frac{r_2+e_2}{e_1+\delta_1} \right\rceil \quad \leq \quad \left\lfloor \frac{\delta_2+r_2}{e_1} \right\rfloor$$

gilt. In dieser Bedingung ist jedoch für den Fall $e_2 = \delta_1$ (und damit auch $n = 1$) Bedingung (3.13) bereits enthalten, da gilt:

$$\begin{aligned}
& \left\lceil \frac{r_2+e_2}{e_1+\delta_1} \right\rceil \leq n = 1 \leq \left\lfloor \frac{\delta_2+r_2}{e_1} \right\rfloor \\
\Longrightarrow \quad & r_2 + e_2 \leq e_1 + \delta_1 \quad \wedge \quad e_1 \leq \delta_2 + r_2 \\
\Longrightarrow \quad & r_2 + e_2 \leq e_1 + e_2 \quad \wedge \quad e_1 \leq \delta_2 + r_2 \\
\Longrightarrow \quad & r_2 \leq e_1 \quad \wedge \quad e_1 \leq \delta_2 + r_2 \quad \Longrightarrow \quad (3.13)
\end{aligned}$$

Die vereinfachte Bedingung ergibt sich somit zu (3.4). ■

3.4.5 Kriterium für $\delta_1 = \delta_2 = 0$

Hinter dem Taskset $(e_1, r_1, 0)$, $(e_2, r_2, 0)$ verbergen sich zwei Tasks mit jeweils konstanter und bekannter Periode. In Anlehnung an [Eck99] kann die Schedulability formuliert werden:

Satz 3.10 Ein 2-er-Job-Set mit $\delta_1 = \delta_2 = 0$ ist genau dann planbar, wenn

$$e = e_1 + e_2 \quad \leq \quad \mathrm{ggT}(e_1 + r_2,\ e_2 + r_2)$$

gilt.

Beweis Setze in Satz 3.8 $\delta_1' := r_1$, $\delta_2' := r_2$ und $r_1' = r_2' := 0$. Die zu beweisende Aussage ergibt sich dann sofort über die Schedulability-Bedingung mit maximaler Distanz. ■

3.4.6 Heuristik bei $\delta_1 = 0$

Für ein 2-er-Taskset mit Parametern $(e_1, r_1, 0)$, (e_2, r_2, δ_2), $\delta_2 \neq 0$, konnte keine geschlossene Schedulability-Bedingung *ohne* Quantoren angegeben werden, es ist jedoch die Formulierung einer notwendigen Bedingung als Heuristik möglich:

Satz 3.11 Das Set ist nur dann planbar, wenn gilt:

$$\begin{aligned}
& r_1 \geq e_2 \\
\wedge \quad & r_2 + \delta_2 \geq e_1 \\
\wedge \quad \Bigg(\quad & \left\lceil \frac{r_1+e1}{\delta_2+e_2+r_2} \right\rceil \leq \min \left\{ \left\lfloor \frac{r_1-e_2}{e_2+r_2} + 1 \right\rfloor,\ \left\lfloor \frac{r_1}{2e_2} + \frac{1}{2} \right\rfloor \right\} \quad \vee \\
& \left\lceil \frac{e_2+r_2}{e_1+r_1} \right\rceil \leq \min \left\{ \left\lfloor \frac{r_1+r_2+\delta_2}{e_1+r_1} \right\rfloor,\ \left\lfloor \frac{2(r_2+\delta_2)+e_2-e_1}{e_1+r_1} \right\rfloor \right\} \quad \Bigg)
\end{aligned} \tag{3.16}$$

Beweis Die ersten beiden Elemente von (3.16) stellen mit

$$\begin{aligned}
r_1 & \geq e_2 \\
r_2 + \delta_2 & \geq e_1
\end{aligned} \tag{3.17}$$

triviale Forderungen an das Job-Set dar, da nur so ausreichend Ausführungszeit für jeden Task zur Verfügung steht. Durch $\delta_1 = 0$ wird festgelegt, daß J_1 mit einer nicht-variablen Periode (hart-periodisch) verplant werden muß. Es sei angenommen, daß auch J_2 mit einer festen, jedoch nicht a-priori bestimmten Frequenz gestartet wird. Der Abstand zweier Instanzen wird mit

$$d \quad = \quad r_2 + \lambda \cdot \delta_2, \quad \lambda \in [0,1]$$

spezifiziert und muß nicht ganzzahlig sein, da nach einer *notwendigen* Bedingung gesucht wird. Das Starttime-Window wird dazu mit dem Parameter λ linear skaliert.

Zur weiteren Betrachtung ist eine Fallunterscheidung notwendig. Entweder ist die Frequenz von J_2 größer oder kleiner als die Frequenz von J_1:

Fall I: J_2 ist höherfrequent als J_1.
Durch J_2 müssen die Abstände zwischen den hartperiodischen Instanzen J_1 ausgefüllt werden. Pro Zwischenraum seien $n \in N$ Instanzen von J_2 einzuplanen; dies ist nur dann möglich, wenn gilt:

$$\begin{array}{lll} \exists\, \lambda \in [0,1] \quad \exists\, n \in N & (n-1)\cdot(e_2 + r_2 + \lambda\delta_2) + e_2 & \leq \quad r_1 \\ \wedge & n \cdot (e_2 + r_2 + \lambda\delta_2) & \geq \quad r_1 + e_1 \end{array}$$

Es ergeben sich damit Schranken für λ zu

$$\begin{array}{lll} & \lambda & \leq \quad \frac{r_1 - e_2 - (n-1)(e_2+r_2)}{(n-1)\delta_2} \quad =: \lambda_1 \\ \wedge & \lambda & \geq \quad \frac{r_1 + e_1 - n(e_2+r_2)}{n\delta_2} \quad =: \lambda_2, \end{array}$$

was aufgrund des Definitionsbereiches von $\lambda \in [0,1]$ zu folgenden notwendigen Bedingungen führt:

$$\lambda_1 \geq 0, \quad \lambda_2 \leq 1, \quad \lambda_1 \geq \lambda_2$$

$$\begin{array}{lrcll} \Longrightarrow & r_1 & \geq & e_2 + (n-1)(e_2+r_2) & [\lambda_1 \geq 0] \qquad (3.18) \\ \wedge & r_1 + e_1 & \leq & n(\delta_2 + e_2 + r_2) & [\lambda_2 \leq 1] \qquad (3.19) \\ \wedge & \frac{r_1 - e_2}{n-1} & \geq & \frac{r_1 + e_2}{n} & [\lambda_1 \geq \lambda_2] \qquad (3.20) \end{array}$$

Daraus ergeben sich notwendige Bedingungen für n,

$$\begin{array}{lll} \overset{(3.18)}{\Longrightarrow} & n & \leq \quad \frac{r_1 - e_2}{e_2 + r_2} + 1 \\ \overset{(3.19)}{\Longrightarrow} & n & \geq \quad \frac{r_1 + e_1}{\delta_2 + e_2 + r_2} \\ \overset{(3.20)}{\Longrightarrow} & n & \leq \quad \frac{r_1}{2e_2} + \frac{1}{2} \end{array}$$

welche sich aufgrund der geforderten Ganzzahligkeit von n kombinieren lassen zur notwendigen Bedingung im *Fall I*:

$$\left\lceil \frac{r_1 + e_1}{\delta_2 + e_2 + r_2} \right\rceil \;\leq\; \min \left\{ \left\lfloor \frac{r_1 - e_2}{e_2 + r_2} + 1 \right\rfloor, \; \left\lfloor \frac{r_1}{2e_2} + \frac{1}{2} \right\rfloor \right\} \qquad (3.21)$$

Fall II: J_1 ist höherfrequent als J_2.
Analog zum Fall I füllen n J_1-Ausführungen die Abstände zwischen J_2-Instanzen. Wiederum wird auch die Frequenz von J_2 als konstant und durch λ skaliert angenommen:

$$\exists\, \lambda \in [0,1] \quad \exists\, n \in N \qquad n(e_1+r_1)-r_1 \;\leq\; r_2+\lambda\delta_2 \tag{3.22}$$

$$\wedge \qquad n(e_1+r_1) \;\geq\; r_2+\lambda\delta_2+e_2 \tag{3.23}$$

$$\wedge \qquad n(e_1+r_1)+e_1 \;\leq\; 2(r_2+\lambda\delta_2)+e_2 \tag{3.24}$$

Zu gegebenem λ wird ein n in der Weise gesucht, daß genau in der Wartezeit zwischen der n-ten und $(n+1)$-ten Instanz von J_1 eine J_2-Instanz ausgeführt werden kann. Dazu müssen die n Ausführungen von J_1 in den Zwischenraum von J_2-Instanzen passen (3.22) und gleichzeitig garantieren, daß der anschließend zu startende J_1 zu keiner Deadline-Überschreitung von J_2 führt (3.23). Zusätzlich muß noch gefordert werden, daß auch die $(n+1)$-te J_1-Instanz beendet werden kann, bevor J_2 das nächste Mal fällig ist (3.24).

Es ergeben sich für λ

$$\overset{(3.22)}{\Longrightarrow} \qquad \lambda \;\geq\; \tfrac{n(e_1+r_1)-r_1-r_2}{\delta_2} \;=:\; \lambda_1 \tag{3.25}$$

$$\overset{(3.23)}{\wedge} \qquad \lambda \;\leq\; \tfrac{n(e_1+r_1)-r_2-e_2}{\delta_2} \;=:\; \lambda_2 \tag{3.26}$$

$$\overset{(3.24)}{\wedge} \qquad \lambda \;\geq\; \tfrac{n(e_1+r_1)+e_1-2r_2-e_2}{2\delta_2} \;=:\; \lambda_3 \tag{3.27}$$

und zusammen mit $\lambda \in [0,1]$ und $\lambda_1, \lambda_3 \leq \lambda \leq \lambda_2$ folgende notwendigen Bedingungen:

$$\lambda_1 \leq 1, \quad \lambda_2 \geq 0, \quad \lambda_3 \leq 1, \quad \lambda_1, \lambda_3 \leq \lambda_2$$

$$\overset{(3.25)}{\Longrightarrow} \qquad n(e_1+r_1) \;\leq\; r_1+r_2+\delta_2 \qquad [\lambda_1 \leq 1] \tag{3.28}$$

$$\overset{(3.26)}{\wedge} \qquad n(e_1+r_1) \;\geq\; r_2+e_2 \qquad [\lambda_2 \geq 0] \tag{3.29}$$

$$\wedge \qquad r_1 \;\geq\; e_2 \qquad [\lambda_1 \leq \lambda_2] \tag{3.30}$$

$$\overset{(3.27)}{\wedge} \qquad 2\delta_2+2r_2+e_2 \;\geq\; n(e_1+r_1)+e_1 \qquad [\lambda_3 \leq 1] \tag{3.31}$$

$$\wedge \qquad e_1-e_2 \;\leq\; n(e_1+r_1) \qquad [\lambda_3 \leq \lambda_2] \tag{3.32}$$

Dabei sind (3.30) (wegen (3.17)) und (3.32) implizit erfüllt:

$$e_1-e_2 \;\leq\; e_1 \;\leq\; e_1+r_1$$

$$\Longrightarrow \qquad e_1-e_2 \;\leq\; n(e_1+r_1)$$

Dadurch ergeben sich Constraints an n.

$$\overset{(3.28)}{\Longrightarrow} \qquad n \;\leq\; \tfrac{r_1+r_2+\delta_2}{e_1+r_1}$$

$$\overset{(3.29)}{\Longrightarrow} \qquad n \;\geq\; \tfrac{e_2+r_2}{e_1+r_1}$$

$$\overset{(3.31)}{\Longrightarrow} \qquad n \;\leq\; \tfrac{e_2+2r_2+2\delta_2-e_1}{e_1+r_1}$$

Zusammen mit der Ganzzahligkeit von n ist die notwendige Bedingung im *Fall II* also:

$$\left\lceil \tfrac{e_2+r_2}{e_1+r_1} \right\rceil \;\leq\; \min\left\{ \left\lfloor \tfrac{r_1+r_2+\delta_2}{e_1+r_1} \right\rfloor, \; \left\lfloor \tfrac{2(r_2+\delta_2)+e_2-e_1}{e_1+r_1} \right\rfloor \right\} \tag{3.33}$$

Unter den gegebenen Voraussetzungen ist es für die Existenz eines gültigen Schedules notwendig, daß entweder (3.21) oder (3.33) gilt; dies führt umgehend zur angegebenen Heuristik. ■

3.4.7 Kombinierbarkeit

Sämtliche oben vorgestellten Kriterien und Heuristiken, die jeweils nur einen Teil des verfügbaren Parameterbereiches abdecken, lassen sich zu einer einzigen 2-Job-Heuristik kombinieren. Mittels Fallunterscheidungen muß (in angegebener Reihenfolge) verifiziert werden, welches Kriterium bzw. welche Heuristik anwendbar ist:

$r_1 = 0 \ \vee \ r_2 = 0$	$\Longrightarrow$	Kriterium 3.9
$\delta_1 = \delta_2 = 0$	$\Longrightarrow$	Kriterium 3.10
$\delta_1 = 0 \ \vee \ \delta_2 = 0$	$\Longrightarrow$	Heuristik 3.11
Sonst	$\Longrightarrow$	Keine Aussage möglich

Zu experimentellen Resultaten der einzelnen Kriterium und der dargestellten Kombination siehe auch Abschnitt 6.3.

3.5 Mehrjob-Probleme

Bei der Verwendung von mehr als zwei Jobs nimmt die Komplexität des Scheduling-Problems überproportional zu, auch die Formulierung geschlossener Schedulability-Kriterien und -Heuristiken wird stark verkompliziert. Normalerweise muß zur exakten Ermittlung der Planbarkeit auf Suchalgorithmen, wie etwa die vorgestellten Branch-and-Bound-Algorithmen, zurückgegriffen werden. Im Rahmen dieser Arbeit soll nur ein kleiner Ausblick auf *theoretische* Betrachtungen zu Mehrjob-Problemen gegeben werden.

Die im Abschnitt 3.1 vorgestellten grundlegenden Kriterien wurden bereits für Mehrjob-Probleme vorgestellt und stellen wiederum eine minimale Basis für weitere Betrachtungen dar.

3.5.1 Notwendiges Utilisation-Kriterium

Damit ist das in Satz 3.7 dargestellte Utilisation-Kriterium auf Mehrjobprobleme mit nicht-trivialen Release-Zeiten anwendbar. Für einen einzelnen Job

$$J_i \quad = \quad (e_1, r_1, \delta_1)$$

ist die tatsächliche relative Prozessorbelastung P_i pro Job J_i nach unten beschränkt durch

$$\frac{e_i}{e_i + r_i + \delta_i} \quad \leq \quad P_i,$$

womit sich in diesem Falle ein notwendiges Kriterium ergibt:

Satz 3.12 Das *Utilisation-Kriterium*

$$\begin{aligned} & \sum_i \frac{e_i}{e_i + r_i + \delta_i} \ \leq \ P \ = \ \sum_i P_i \ \leq \ 1 \\ \Longrightarrow \qquad & \sum_i \frac{e_i}{e_i + r_i + \delta_i} \ \leq \ 1 \end{aligned}$$

stellt eine notwendige, jedoch nicht hinreichende Bedingung für die Planbarkeit eines Mehrjobproblems dar.

3.5.2 Anwendbarkeit von 2-Job-Kriterien

Oben gewonnene Erkenntnisse für 2-Job-Probleme lassen sich auch beim inkrementellen Scheduling von Mehrjob-Problemen ohne Release-Constraints verwenden. Muß ein vorhandener zyklischer Schedule um *einen* zusätzlichen Job erweitert werden, kann dessen Schedulability vorab heuristisch abgeschätzt werden:

Wurde ein Zyklus gefunden, bei dem sich das Zyklenende in einem gewissen Bereich variieren läßt (siehe etwa Beispiel in Abbildung 6.19 auf Seite 76), kann dieser Schedule als einzelner Job aufgefaßt werden. Bezeichnet S die zyklische Sequenz kleinster Länge t und ist

$$\hat{\delta} \;:=\; |s(t)|_{\min} \quad = \quad \left| \begin{pmatrix} s_1(t) \\ \vdots \\ s_n(t) \end{pmatrix} \right|_{\min} \quad = \; \min(s_1(t),\; \ldots,\; s_n(t)) \tag{3.34}$$

der am Zyklenende verfügbare zeitliche Spielraum (verwendet wird die Notation aus Abschnitt 2.3, sie ist jedoch anpaßbar an die erweiterte Methode aus 4.2), so sind auch

$$\overline{S},\;\; \overline{(S,\; I_1)},\;\; \overline{(S,\; I_2)},\;\; \ldots,\;\; \overline{(S,\; I_{\hat{\delta}})}$$

gültige zyklische Schedules. I_j bezeichnet dabei eine explizite Idle-Zeit der Länge j. Diese Darstellung ist dann aggregierbar zu einem einzelnen Task mit

$$J' \quad := \quad (t, 0, \hat{\delta}),$$

wobei keine Release-Zeiten betrachtet werden. Soll nun der vorgegebene zyklische Schedule um einen weiteren Job J'_{neu} erweitert werden, geben die im Abschnitt 3.4 vorgestellten Kriterien für das neu definierte 2-Job-Problem

$$\begin{aligned} J' \quad &= \quad (t, 0, \hat{\delta}) \\ J'_{\text{neu}} \quad &= \quad (e, r, \delta) \end{aligned}$$

einen Hinweis auf die inkrementelle Schedulability.

In Abschnitt 4.2 wird später eine Anpassung des Zyklenmodelles auf Probleme *mit* Release-Constraints vorgestellt. Nach einer Modifikation der $\hat{\delta}$-Berechnung (3.34) sind die Ergebnisse auch auf Mehrjob-Probleme *mit* Release-Zeiten übertragbar. Die Zyklenlänge kann dann nur in Bereichen

$$\overline{(S,\; I_{\hat{r}})},\;\; \overline{(S,\; I_{\hat{r}+1})},\;\; \ldots,\;\; \overline{(S,\; I_{\hat{\delta}})}$$

variiert werden und führt damit zur Aggregation

$$J' \quad = \quad (t, \hat{r}, \hat{\delta}),$$

welche wiederum mittels der vorgestellten 2-Job-Kriterien als Hinweis auf die inkrementelle Schedulability eines Jobs J'_{neu} dienen kann.

Kapitel 4

Vollständige Verfahren

Das Scheduling-Problem ist NP-Vollständig, siehe auch [S94]. Prüfung auf Schedulability bzw. Erstellung eines gültigen Schedules ist somit nicht immer mit polynomiellem Zeitaufwand möglich. Dennoch können vollständige Verfahren –zumindest bei relativ kleinen Job-Sets– eingesetzt werden, um gültige Schedules zu generieren. Mit **Vollständigen Verfahren** werden dabei nach [RN95] Suchstrategien bezeichnet, welche *garantiert* eine Lösung finden, sofern eine solche existiert. Zusätzlich ist es sinnvoll, ein garantiertes **Terminieren** der Algorithmen zu fordern.

Besonders weitverbreitete Methoden im Scheduling sind dabei Branch-and-Bound-Algorithmen. Zwei Vertreter dieser Gattung werden im folgenden eingehender betrachtet.

4.1 Branch and Bound – "Slack Time Vector"

In der Vorstellung verwandter Lösungsmethoden wurde in Kapitel 2.3 die Slack-Time-Vector-Methode aus [Eck99] behandelt.

Das dort angegebene Verfahren dient zur Bestimmung von Schedules für Job-Sets *ohne* Release-Constraints. Innerhalb des Branch-and-Bound Entscheidungsbaumes werden dabei sukzessive Jobinstanzen verplant und die entstandenen partiellen Sequenzen auf Zyklenschluß geprüft. Dazu werden die vorgestellten *Slack Time Vectors* verwendet. In dem Releasezeit-freien Fall konnte die Korrektheit des Algorithmus gezeigt werden. Einzelheiten bezüglich des Verfahrens siehe im Kapitel 2.3 oder direkt bei [Eck99].

4.2 Branch and Bound – "Relative Starttime Vector"

Der Nachteil oben beschriebener Slack Time Vector Methode besteht darin, daß nur Tasks mit trivialen Release-Zeiten von konstant 0 verwendet werden können. Wenn auch nur ein Task ein Release-Constraint besitzt, versagt das Majoranten-Kriterium (siehe Abschnitt 2.3.2) als Garant für die Existenz eines zyklischen Schedules.

Das Verfahren soll daher auf beliebige Job-Sets (e_i, r_i, δ_i) erweitert werden. Der Slack Time Vector wird dabei durch den weiter unten vorgestellten *Relative Starttime Vector* ersetzt. Dieser dient in Verbindung mit einem *Constraint Vector* zur Überprüfung der Timing Constraints und zum Erkennen von zyklischen Sequenzen. Insgesamt läßt sich das neue Verfahren nach Abbildung 4.1 klassifizieren:

Funktionsprinzip *(Offline / Online)*	●	
Echtzeitverhalten *(Hard / Soft Realtime System)*	●	
Ausführungsumgebung *(Single / Multiple Processor)*	●	
Tasktypen *(Periodisch / Sporadisch bzw. einmalig)*	●	
Ausführungszeiten *(Exakt / Beschränkt)*	●	
Release-Zeiten *(Absolut / Relativ)*	○[a]	●
Deadlines *(Absolut / Relativ)*	○[b]	●
Unterstützung Aperiodischer Tasks *(Offline/Online)*	○	

[a] Für jeweils erste Taskinstanzen
[b] Für jeweils erste Taskinstanzen

Abbildung 4.1: *Klassifizierender Überblick über die Methode der "Relative Starttime Vectors"*

Ein Schedule, d.h. eine Sequenz

$$S = (S_1, S_2, \ldots, S_n) \cong (J_{S_1}, J_{S_2}, \ldots, J_{S_n})$$

der Länge $|S| = n$, bezeichne die Reihenfolge der zu dem gegebenen Task-Set gehörenden Instanzen. Der nachfolgende Task wird umgehend gestartet, sowie der ursprüngliche beendet wird. Leerlaufzeiten werden durch implizite oder explizite *Idle-Tasks* (siehe Abschnitt 3.2.3) erfaßt.

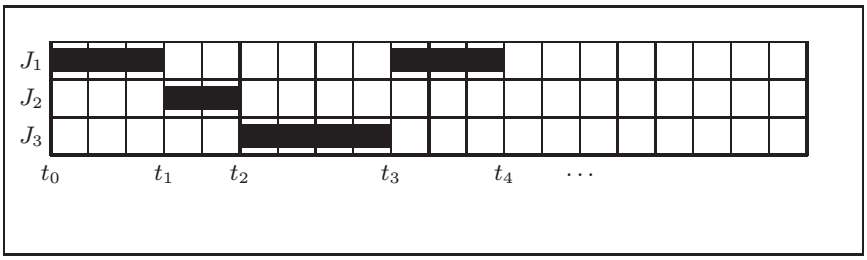

Abbildung 4.2: *Beispiel eines partiellen Schedules für drei Tasks mit korrespondierenden Offline-Scheduling Zeitmarken*

Ein initial leerer Schedule $S = ()$ wird am Ende sukzessive um eine Taskinstanz erweitert. Bei der Erweiterung dürfen die Timing Constraints nicht verletzt werden und es muß geprüft werden, ob durch das Anfügen der Instanz ein Zyklus entstanden ist. Dies geschieht mit Hilfe der *Relative Starttime Vectors* und der *Relative Constraint Vectors*. Zuerst soll dazu der Constraint Vector vorgestellt und eine Aussage bezüglich der Gültigkeit eines partiellen Schedules getroffen werden. Im Abschnitt 4.2.4 wird dann auch der Starttime Vector definiert.

4.2.1 Relative Constraint Vector (RCV)

Dem aktuell betrachteten Sequenz-Kandidaten, welcher einen Schedule bis zum Zeitpunkt t_i repräsentiert, wird im Offline-Scheduling ein **Relative Constraint Vector**

(RCV) zugeordnet. Dieser enthält für jeden Job die noch zur Verfügung stehende Deadline, also den zeitlichen Abstand bis zum Ende des Start Time Windows.

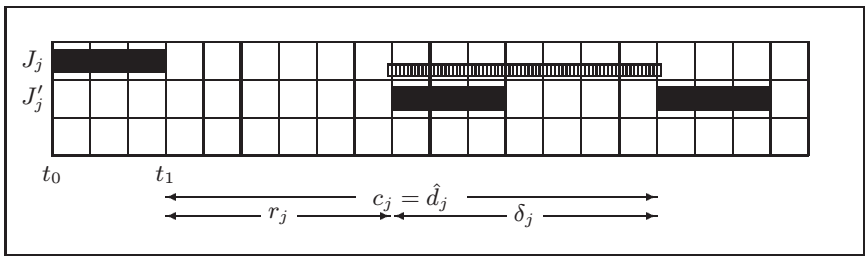

Abbildung 4.3: *Relative Constraints an Task. Für Task J_j wird durch Slack Time $\hat{d}_j$ (implizit gegeben durch Deadline) und Release Time r_j das Start Time Window (Schmaler Balken, δ_j) definiert.*

Der RCV besitzt die Form

$$C_{t_i} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}, \quad c_i \in R_0^+$$

und wird initial, d.h. mit leerem Schedule, auf

$$C_{t_0} := \begin{pmatrix} \hat{d}_1 \\ \vdots \\ \hat{d}_n \end{pmatrix} = \begin{pmatrix} r_1 + \delta_1 \\ \vdots \\ r_n + \delta_n \end{pmatrix}$$

festgesetzt. An dieser Stelle lassen sich für die jeweils erste Taskinstanz im Schedule aber auch beliebige andere Werte als absolute Constraints vorgeben.

In [Eck99] enthielt der Slack Time Vector stets den *spätesten* Beginn der folgenden Instanz, hier wird ein Zeitintervall codiert: Aus dem Abstand zum *Ende* des Fensters und der Fensterlänge selbst ergibt sich, daß bei

$$c_j > \delta_j$$

der korrespondierende Task noch nicht freigegeben (nicht released) ist und eine verbleibende Release-Time von

$$r' = c_j - \delta_j$$

besteht. Andererseits gilt bei

$$c_j \leq \delta_j,$$

daß der entsprechende Task bereits freigegeben wurde und jederzeit gescheduled werden darf. Der aktuelle Planungszeitpunkt liegt *innerhalb* des Start Time Windows.

Zusammenfassend kann das *relative* Start Time Window I_j, innerhalb dessen die nächste Instanz des Jobs J_j gestartet werden muß, folgendermaßen aus dem RCV ermittelt werden:

$$\begin{aligned} c_j \leq \delta_j &\implies I_j = [0, c_j] \\ c_j > \delta_j &\implies I_j = [c_j - \delta_j, c_j]. \end{aligned}$$

4.2.2 Update des RCV

Wie oben erwähnt, muß nur für das Ende der aktuellen Sequenz ein RCV vorhanden sein, der gesamte Branch-and-Bound-Algorithmus kommt daher strenggenommen mit *einem* RCV aus. Bei Hinzufügung einer neuen Jobinstanz J_j an die Sequenz, ausgehend von Abbildung 4.3 siehe Abbildungen 4.4 und 4.5 mit verschiedenen Tasklängen, müssen die Elemente des Vektors (analog zum Slack Time Vector) an den neuen Planungszeitpunkt angepaßt werden. Die Elemente des neuen RCV $\bar{C}$ ergeben sich aus C durch:

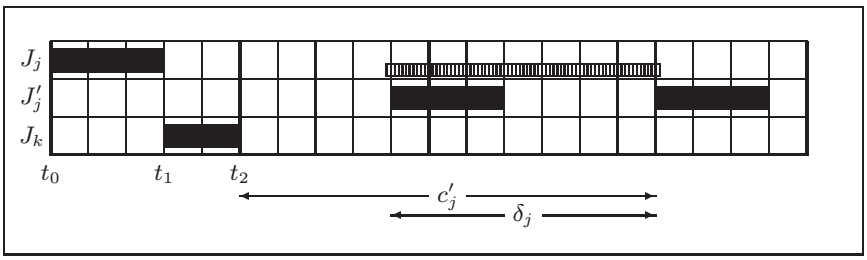

Abbildung 4.4: *Veränderung des RCVs beim Scheduling eines kurzen Tasks, so daß $c'_j > \delta_j$*

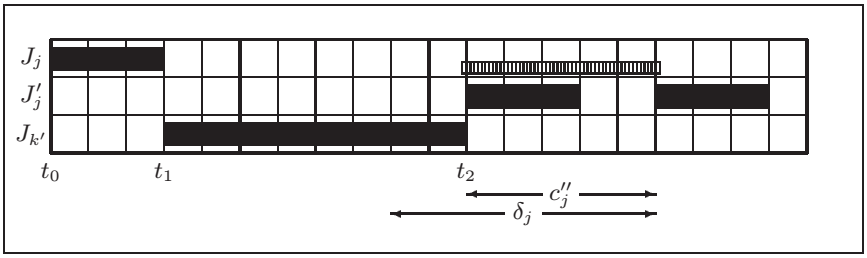

Abbildung 4.5: *Modifikation des RCVs bei Planung eines langen Tasks (respektive mehrerer kurzer Tasks), so daß $c''_j \leq \delta_j$ und Task J_j sofort ausführbar wird. Das relative Start Time Window verkleinert sich somit.*

$$\bar{c}_i \quad = \quad \begin{cases} r_i + \delta_i & : \quad i = j \\ c_i - e_j & : \quad i \neq j \end{cases}$$

Nach dem Update müssen folgende Constraints auf Verletzungen geprüft werden.

4.2.3 Gültigkeit des RCV

Der Relative Contraint Vector muß einige Bedingungen erfüllen, wenn der weitere Scheduling-Prozeß im Branch-and-Bound-Verfahren zu einem gültigen Schedule führen soll (Γ bezeichnet dabei die Menge $\Gamma = \{1, \ldots, n\}$):

- $c_j \geq 0 \quad \forall\, j \in \Gamma$

  Besitzt der RCV negative Komponenten, so ist das relative Start Time Window für den korrespondierenden Task bereits verstrichen und es ist somit zu einer Deadline-Überschreitung gekommen.

- $\exists\, j \in J \quad e_j \;\leq\; \min\limits_{i\ \in \Gamma \setminus \{j\}} c_i$

  Mindestens ein Task muß hinzugefügt werden können, ohne eine Deadline-Überschreitung bei den anderen Jobs zu bedingen. Werden zwei Jobs gleichzeitig "fällig", kommt es mindestens bei einem zu einer Deadline-Überschreitung.

- $\left| \left\{ \;\; i \;\; \mid \;\; i \neq j \;\; \wedge \;\; e_j \;=\; c_i \;\; \right\} \right| \;\leq 1 \quad \forall\, j \in \Gamma$

  Nur höchstens ein Job darf durch die Einplanung einer anderen Instanz an die Grenze des eigenen Start Time Windows stoßen.

- $\sum\limits_{j \in J} c_j \;\geq\; \sum\limits_{j \in J} e_j$

  In der gesamten, noch zur Verfügung stehenden Zeit bis zum Ende aller Start Time Windows müßte jede Instanz mindestens einmal ausgeführt werden können.

Die beschriebenen Anforderungen an den RCV können innerhalb eines Branch-and-Bound-Algorithmus für ein lokales Bounding verwendet werden. Die Effizienz des Algorithmus wird dadurch gesteigert, ohne jedoch die Komplexität verringern zu können. Es soll daher hier nicht weiter nach stärkeren Bounding-Kriterien gesucht werden.

4.2.4 Relative Starttime Vector (RSV)

Analog wird für *jede* Taskanfangs-Zeit t_k innerhalb einer Sequenz der **Relative Starttime Vector** definiert. In einer Sequenz beginnen die einzelnen Instanzen zu den Zeitpunkten $t_1,\ t_2,\ t_3, \ldots,\ t_n$ und für *jede* dieser Zeiten wird der korrespondierende RSV benötigt. Eine Sonderstellung nimmt dabei eine fixe Zeit t_0, etwa der Start der Sequenz, ein:

$$\Delta T(t_k) = \begin{pmatrix} \Delta t_1(t_k) \\ \vdots \\ \Delta t_n(t_k) \end{pmatrix} =: \Delta T_{t_k}, \qquad \Delta t_i(t) \in N_0^+ \cup \{\text{undef}\}$$

Die Semantik der Elemente ist dabei folgendermaßen gegeben: Zum Zeitpunkt t_k liegt der Start von Taskinstanz J_j noch $\Delta t_j(t_k)$ Zeiteinheiten entfernt, absolut wird die nächste Instanz also zum Zeitpunkt $t_k + \Delta t_j(t_k)$ gestartet. Ist $\Delta t_j(t_k) = 0$, so wird J_j gerade zu diesem Zeitpunkt gestartet; ist $\Delta t_j(t_k)$ undefiniert, so wurde noch keine weitere Instanz in den Schedule eingetragen. Zur Illustration dieses Sachverhaltes siehe auch das Beispiel im folgenden Abschnitt.

4.2.5 Update des RSV

Beginnend mit dem leeren Schedule $S = ()$ sind noch keinerlei Taskinstanzen verwendet, es gilt also

$$\Delta T_0 = \begin{pmatrix} \text{undef} \\ \vdots \\ \text{undef} \end{pmatrix}.$$

Wird nun im Verlaufe des Scheduling-Prozesses Task J_j an den Schedule angefügt, so müssen sämtliche im aktuellen Ast des Suchbaumes vorangehenden RSVs bis zum letzten Vorkommen von J_j angepaßt werden. Dies entspricht bei der verwendeten

Spezifikation des Branch-and-Bound-Algorithmus genau den RSVs, die zu den bereits im Schedule befindlichen Taskinstanzen gehören. In der j-ten Komponente wird dazu die Differenz der aktuellen und der vom RSV repräsentierten Zeit gespeichert:

$$() \cong \underbrace{\begin{pmatrix} \text{undef} \\ \text{undef} \\ \text{undef} \end{pmatrix}}_{\Delta T_{t_0}}$$

$$(J_2) \cong \underbrace{\begin{pmatrix} \text{undef} \\ t_1 - t_0 \\ \text{undef} \end{pmatrix}}_{\Delta T_{t_0}} \underbrace{\begin{pmatrix} \text{undef} \\ 0 \\ \text{undef} \end{pmatrix}}_{\Delta T_{t_1}}$$

$$(J_2\ J_3) \cong \underbrace{\begin{pmatrix} \text{undef} \\ t_1 - t_0 \\ t_2 - t_0 \end{pmatrix}}_{\Delta T_{t_0}} \underbrace{\begin{pmatrix} \text{undef} \\ 0 \\ t_2 - t_1 \end{pmatrix}}_{\Delta T_{t_1}} \underbrace{\begin{pmatrix} \text{undef} \\ \text{undef} \\ 0 \end{pmatrix}}_{\Delta T_{t_2}}$$

$$(J_2\ J_3\ J_1) \cong \underbrace{\begin{pmatrix} t_3 - t_0 \\ t_1 - t_0 \\ t_2 - t_0 \end{pmatrix}}_{\Delta T_{t_0}} \underbrace{\begin{pmatrix} t_3 - t_1 \\ 0 \\ t_2 - t_1 \end{pmatrix}}_{\Delta T_{t_1}} \underbrace{\begin{pmatrix} t_3 - t_2 \\ \text{undef} \\ 0 \end{pmatrix}}_{\Delta T_{t_2}} \underbrace{\begin{pmatrix} 0 \\ \text{undef} \\ \text{undef} \end{pmatrix}}_{\Delta T_{t_3}}$$

Es sind oben jeweils sämtliche zu den einzelnen Planungszeitpunkten gehörende RSVs angegeben. Links ist vermerkt, welche partielle Sequenz durch die Liste der RSVs repräsentiert wird.

Da die Update-Methode pro angefügter Instanz die Änderung relativ vieler RSVs betreffen kann, da Taskinstanzen im Schedule weit voneinander entfernt auftreten können, hat dieses Verfahren nur theoretischen Wert. Es wird weiter unten eine Methode zur RSV-Anpassung mit *linearem* Zeitaufwand vorgestellt (Kapitel 4.2.8).

4.2.6 Cycle Checking

Das eigentliche Ziel der Verwendung von RSV und RCV ist neben dem Erstellen *gültiger* Schedules S das Erkennen von gültigen *Zyklen* nach der Definition in 2.3. Für solche Zyklen in der partiellen Sequenz

$$S = (S_1, \ldots, S_{k-1}, \overset{\Delta T}{\underset{\uparrow}{S_k}}, \ldots, \overset{C}{\underset{\uparrow}{S_n}})$$

muß gelten:

- $S_{k-1} = S_n$ (Ein Task schließt den Zyklus $\overline{(S_k, \ldots, S_n)}$)

- RCV C nach S_n muß von RSV ΔT_{t_k} zu S_k "erfüllt" werden, d.h. mit

$$C = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}, \quad \Delta T = \begin{pmatrix} \Delta t_1 \\ \vdots \\ \Delta t_n \end{pmatrix}$$

  muß $\forall\ 1 \leq i \leq n$ gelten:

  - Δt_i definiert
    (Es muß mindestens eine Taskinstanz vom Job J_i im Zyklus geben)
  - $\Delta t_i \leq c_i$
    (Wiederholungstask verletzt nicht die Deadline)
  - $c_i > \delta_i \implies c_i - w_i \leq \Delta t_i$
    (Falls noch Rest-Releasezeit für Task besteht, darf der Wiederholungstask nicht vor Ablauf dieser Zeit gestartet werden)

Ist eine solche Konstellation gefunden, so wird ein unendlicher gültiger Schedule durch Wiederholung des zyklischen Teils definiert.

$$\overline{S} \quad = \quad (S_1,\ \ldots,\ S_{k-1})\ \overline{(S_k,\ \ldots,\ S_n)}$$

Um bei der Prüfung auf Zyklen den Aufwand zu minimieren, können aus obigen Bedingungen abgeleitete Einschränkungen verwendet werden. Wurde als letzter Job J_i verplant, gelten folgende Beschränkungen beim Prüfen auf einen Zyklus:

- Prüfung muß nur erfolgen, wenn sämtliche Jobs bereits mindestens einmal verplant wurden.
- Zyklus kann nur unmittelbar nach den anderen bereits geplanten Instanzen von J_i beginnen. Nach Beispiel 6.1 (Seite 65) muß dies nicht unbedingt das letzte Vorkommen von J_i sein.
- Der zum potentiellen Beginn des Zyklus gehörenden RSV muß vollständig definiert sein, d.h. nach der Prüfstelle muß im Zyklus jeder Task mindestens mit einer Instanz vertreten sein.

Trivialerweise wird auch dann ein gültiger Zyklus gefunden, wenn in einer gültigen Sequenz an zwei verschiedenen Stellen *derselbe RSV* auftritt. Die Erkennung dieser Konstellation kann jedoch erst mit Verspätung erfolgen, da nach Zyklenende noch jede Taskinstanz mindestens ein weiteres Mal eingeplant werden muß, um den RSV vollständig zu definieren.

4.2.7 Idle-Tasks

Nach den Aussagen unter 3.2.2 und 3.2.3 (Seite 24) ist es notwendig, Idle-Zeiten sowohl implizit als auch explizit zu verwenden. In diesem Branch-and-Bound-Modell werden Idle-Tasks daher formal als spezielle Job-Instanzen spezifiziert. Die Sequenz $S = (S_1, S_2, \ldots)$ besteht dann aus Tasks J_j und Idle-Zeiten I_k, wobei k die Länge der Leerlaufzeit angibt:

$$S_i \quad = \quad \begin{cases} J_j & : \text{ Task-Instanz von } J_j \\ I_k & : \text{ Idle-Zeit der Länge } k \end{cases}$$

Alternativ dazu kann zur einfacheren Speicherung eine Unterscheidung nach Vorzeichen der Job-ID vorgenommen werden. Diese Darstellung ist jedoch äquivalent zur obigen Definition und dient nur der einfacheren Formalisierung:

$$S_i = J_j \quad = \quad \begin{cases} j \geq 0 & : \text{ Task-Instanz von } J_j \\ j < 0 & : \text{ Idle-Zeit der Länge } -j \end{cases}$$

An jedem Branch-Knoten im Entscheidungsbaum sind somit nicht nur sämtliche Job-Instanzen zu prüfen (wodurch sich unter Umständen implizite Idle-Zeiten einstellen), auch explizite Idle-Zeiten müssen verplant werden.

Die Länge der Idle-Zeiten i ist dabei beschränkt durch

$$i \quad \in \quad [0, \ \ldots, \ \min_{j \in J} c_j],$$

d.h. es müssen auch so große Idle-Intervalle möglich sein, daß der Start eines Tasks bis an das Ende seines Start Time Windows verzögert wird. Durch diese Tatsache wird der Branching Factor, d.h. die Anzahl der maximal zu betrachtenden Nachfolgeknoten im Baum, stark vergrößert.

Basierend auf Satz 3.1 ist es jedoch möglich, immer mindestens einen der folgenden Task-Instanzen zu Beginn des korrespondierenden Start Time Windows zu schedulen, ohne die weitere Planbarkeit des Problems einzuschränken. Es ist daher denkbar, den Schedule nicht streng gerichtet von vorne nach hinten aufzubauen, sondern Rücksprünge zuzulassen: Dazu müssen dann keine expliziten Idle-Zeiten verplant werden; es ist jedoch dann notwenig, durch implizite Idle-Zeiten entstehende Lücken im Schedule rückwärts und rekursiv durch andere Job-Instanzen auszfüllen. Dadurch würde der Branching Factor wieder auf das gewohnte Maß reduziert werden und auch die Tiefe des Suchbaumes nicht wachsen. Allein der Verwaltungsaufwand für die Festlegung der Reihenfolge der zu verplanenden Instanzen würde leicht ($O(1)$) ansteigen. Aus Zeitgründen konnte diese Optimierung im Algorithmus jedoch nicht implementiert werden.

4.2.8 Optimiertes RSV-Update

Der Updating-Prozeß der Relative Starttime Vectors kann durch ein algorithmisches Detail stark verbessert werden: Es müssen bei Verplanung einer neuen Taskinstanz nicht mehr alle zurückliegenden RSVs bis zu einer gewissen Grenze durchlaufen und modifiziert werden, sondern es genügt eine einfache Änderung.

Anstelle der relativen Startzeiten oder der `undef`-Markierungen werden Zeiger auf entsprechende Listeneinträge gespeichert. In dieser Liste werden die Startzeiten sämtlicher Jobinstanzen vermerkt. Initial sind alle Startzeiten `undef`, bei der Verplanung jedoch werden sie entsprechend gesetzt. Aus dem dort in der Liste gespeicherten absoluten Starttermin und der vom RSV repräsentierten absoluten Zeit kann dann durch Differenzbildung der relative Startzeitpunkt wie oben berechnet werden.

Durch diese Modifikation bleibt die Zeitkomplexität, von eventuell höheren Branching Factors bei der Einplanung expliziter Idle-Tasks abgesehen, von gleicher Ordnung gegenüber dem in [Eck99] angegebenen Branch-and-Bound-Algorithmus *ohne* Berücksichtigung von Release-Zeiten. Bei Verwendung der oben angedeuteten nichtgerichteten Generierung des Schedules zur Vermeidung von expliziten Idle-Zeiten sollte der Algorithmus auch insgesamt dieselbe Komplexität behalten.

4.2.9 Branching and Bounding

Ähnlich zu dem in Abschnitt 2.3 vorgestellten Branch-and-Bound-Algorithmus wird auch in diesem Falle der Suchbaum durchlaufen. Jeder Entscheidungsknoten entspricht

dem Hinzufügen einer Jobinstanz an das Ende der Sequenz, der Pfad zu einem Entscheidungsknoten entspricht dem dabei entstandenen partiellen Schedule.

Alternativ kann in jedem Entscheidungsknoten eine beliebige Taskinstanz oder eine Idlezeit der Länge i, $i \leq \min c_j$, an die Sequenz angehängt werden. Dabei muß wie beschrieben ein neuer RSV generiert und die Liste der zurückliegenden RSVs angepaßt werden. Genauso muß der RCV modifiziert und auf Constrainteinhaltung geprüft werden.

Ergibt sich beim Cycle-Checking, daß ein gültiger Zyklus in der aktuellen Sequenz enthalten ist, so kann mit positivem Ergebnis abgebrochen werden. Andererseits kann *lokales Bounding* erfolgen, wenn Deadlines überschritten wurden oder die in 4.2.3 beschriebenen Constraints nicht erfüllt sind. Ein *globales Bounding* kann erfolgen, wenn dieselbe lokale Konstellation bereits betrachtet und zurückgewiesen wurde. Wurde ein Teilast im Baum durch Bounding vollständig abgegrenzt, so ist erwiesen, daß sämtliche partiellen Schedules auf diesem Ast nicht zu einer gültigen Lösung führen können. Die innerhalb des Teilastes zugewiesenen Relative Starttime Vectors repräsentieren diese Situationen. Konnte daher zu einem vollständig definierten RSV im weiteren Planungsprozeß *kein* gültiger Schedule gefunden werden, so können sämtliche Kandidaten, welche denselben RSV enthalten, bis zu dieser Stelle ebenso zurückgewiesen werden. Praktisch muß nach einem Branching nur geprüft werden, ob einer der beim Update vervollständigten RSVs bereits anderweitig zurückgewiesen wurde. Ein Majorantenkriterium (siehe [Eck99]) scheidet an dieser Stelle aus, da bei diesem Kriterium keine Release-Zeiten berücksichtigt werden können. Es genügt auch nicht zu garantieren, daß sich zwei Start Time Windows überschneiden, da keine konkrete Aussage über den tatsächlichen Start der Instanz gemacht werden kann. Nur wenn die beiden Fenster einander entsprechen ist sichergestellt, daß der tatsächliche Startzeitpunkt beiden Constraints genügt.

4.2.10 Korrektheit und Vollständigkeit

Satz 4.1 Für ein beliebiges Job-Set mit $J_i = (e_i, r_i, \delta_i), \quad e_i, r_i, \delta_i \in N$, ist der vorgestellte Branch-and-Bound-Algorithmus korrekt und vollständig. Er terminiert stets und liefert genau dann einen gültigen zyklischen Schedule, wenn das Job-Set schedulbar ist.

Beweis Für den RSV gilt

$$\begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \leq s = \begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix} \leq \begin{pmatrix} r_1 + \delta_1 \\ \vdots \\ r_n + \delta_n \end{pmatrix}$$

$$\Longrightarrow \qquad \Big| \{s\} \Big| = \prod_{i=1}^{n} (r_i + \delta_i + 1) \ll \infty,$$

da ansonsten eine relative Deadline überschritten würde. Die Anzahl der möglichen RSVs ist demnach beschränkt. Da jeder Job früher oder später zu jedem partiellen Schedule hinzugefügt werden muß, damit es nicht zu einer Deadline-Überschreitung kommt, treten in jedem Ast vollständig definierte RSVs auf. In der Cycle-Check-Bedingung, s.o., wird besagt, daß ein Zweig bereits dann einen Zyklus enthält –und der Algorithmus somit terminiert–, wenn ein RSV auf demselben Ast mehrfach auftritt. Allein diese Bedingung beschränkt die Tiefe des Suchbaumes und garantiert damit die Terminierung. Durch die Bounding-Bedingungen ist es jedoch möglich, die Äste schon sehr viel früher zu kappen, da es zu Deadline-Überschreitungen oder äquivalenten Situationen im Baum kommt.

- "$\Longrightarrow$": *Gefundene Schedules sind gültig*
  Aufgrund der Konstruktion des Algorithmus sind sämtliche partiellen Schedules, die im Planungsprozeß entstehen, gültig. Sowohl explizite Constraints als auch die Nicht-Gleichzeitigkeit werden eingehalten. Durch die Prüfung des RSV mittels des RCV wird auch an der Stelle des Zyklenschlusses die Einhaltung aller Bedingungen gewährleistet. Somit ist auch ein eventuell erreichter zyklischer Schedule gültig.
- "$\Longleftarrow$": *Existierender gültiger Schedule wird gefunden*
  Für jedes planbare ganzzahlige Job-Set muß ein unendlicher Schedule angegeben werden können. Nach den Sätzen 3.3, 3.4 und 3.5 ergibt sich, daß dieser Schedule mittels einer zyklischen Sequenz

  $$S \quad = \quad (S_1,\ S_2,\ \ldots,\ S_n), \qquad S_i = \begin{cases} J_j \\ I_i \end{cases}$$

  so dargestellt werden kann, daß sich der Schedule zu $(S', \overline{S})$ ergibt. J_j bezeichnet dabei eine Instanz von J_j, Idlezeiten der Länge i werden durch I_i dargestellt.

  Da in jedem Knoten des Suchbaumes ein S_i an die Sequenz angehängt wird, muß auch die Sequenz (S', S) früher oder später überprüft werden, sofern vorher noch kein anderer gültiger Zyklus gefunden oder der Ast vorzeitig abgebrochen wurde.

  Der zu angegebener Sequenz führende Ast im Baum kann dabei *nicht* durch ein *globales* Bounding vorzeitig beendet werden, da keiner auf dem Zweig repräsentierten partiellen Schedules zu einer Nicht-Schedulability geführt werden kann – zumindest $(S', \overline{S})$ ist ja stets eine gültige Lösung. Auch kann es, da in der gültigen Sequenz alle Constraints erfüllt sein müssen, zu keinem *lokalen* Bounding kommen, da sämtliche auftretenden RSVs keine negativen Komponenten besitzen.

  Es wird daher der Ast, der durch (S', S) repräsentiert ist, erreicht und der letzte Knoten von S überprüft. Da S als gültige Sequenz angenommen wurde, müssen an der Stelle des Zyklenschlusses alle Constraints erfüllt sein. Genau dies ist durch das unter 4.2.6 angegebene Kriterium erfaßt, der Zyklus wird also erkannt.

Insgesamt terminiert der Algorithmus also stets und liefert einen gültigen Schedule, sofern ein solcher Schedule existiert; jeder generierte Schedule erfüllt die gestellten Bedingungen, so daß die Methode insgesamt korrekt ist. ■

Kapitel 5

Iterative Verfahren

Neben den *vollständigen* Verfahren kommen häufig *iterative* Verfahren zum Einsatz, wenn es um die Lösung oder Optimierung von NP-vollständigen Problemen geht. Das Finden einer Lösungen kann dabei nicht garantiert werden, genausowenig wie das Erreichen eines lokalen oder globalen Optimums bezüglich einer Bewertungsfunktion. Dafür *kann* jedoch bei einer geschickten Wahl von Modellen und Heuristiken auch ein NP-vollständiges Problem schnell, einfach und oft auch mit einer den Anforderungen genügenden Optimalität bearbeitet werden.

5.1 Iterative Nachverbesserung

Basierend auf einer gültigen Ausgangslösung, die etwa mittels eines vollständigen Verfahrens gewonnen werden kann, läßt sich ein Schedule iterativ nachverbessern. Das Ziel ist hier weniger die Erlangung eines gültigen Schedules oder die Entscheidung, ob ein Problem-Set planbar ist, sondern vielmehr die Anpassung eines Schedules an weitergehende Problemerfordernisse.
Exemplarisch sollen hier einige mögliche iterative Algorithmen genannt werden:

- *Dehnung oder Stauchung des Schedules*
  Je nach Algorithmus neigen initiale Schedules dazu, Taskinstanzen möglichst zu Beginn oder zu Ende des Start Time Windows auszuführen. Dies resultiert unter anderem aus den theoretischen Erkenntnissen bezüglich des Schedulings. Strategien wie "Earliest Deadline First" o.ä. führen zu solch einseitigen Schedules. Iterative Nachverbesserung kann jetzt etwa die Prozessorauslastung minimeren, indem möglichst lange Idle-Zeiten in den Schedule eingebaut werden. Ebenso kann versucht werden, überflüssige Idle-Intervalle durch Verschiebung von Instanzen zu beseitigen.

- *Einfügen von sporadischen oder einmaligen Tasks*
  Es kann iterativ versucht werden, den Schedule so anzupassen, daß ein vorgegebener aperiodischer Task eingebaut werden kann. Unter Umständen müssen dazu auch Jobinstanzen verlegt werden.

- *Änderung von Job-Parametern*
  Aufbauend auf gültigen Schedules ist es oft einfach möglich, für ähnliche Probleme ebenso gültige Schedules zu erzeugen. So macht es etwa die Verkürzung von Execution Times oft nicht notwendig, den Schedule von Grund auf neu aufzubauen. Iterative Verfahren können Schedules bezüglich der Job-Parameter modifizieren.

5.2 Directed Simulated Annealing

Neben oben genannten Verfahren sind noch viele weitere iterative Nachverbesserungs-Strategien denkbar. An dieser Stelle soll jedoch nicht weiter ins Detail gegangen werden sondern eine umfassendere iterative Methode vorgestellt werden, die sowohl zur Erzeugung von gültigen Schedules als auch zur späteren Optimierung bezüglich vorgegebener Ziele dienen kann.

Besonderes Augenmerk wird dabei auf

- Stabilität,
- Flexibilität,
- Geschwindigkeit,
- Intuitivität und
- Visualisierbarkeit

des zu definierenden Modelles gelegt. Basieren soll die Methode auf dem *Simulated Annealing*, das hier zuerst kurz vorgestellt wird:

5.2.1 Einführung

Verfahren des **Simulated Annealings** (siehe [FAQNLP]) dienen dazu, durch Parametervariation globale Minima einer vorgegebenen Zielfunktion zu finden. Bevor auf die Methode des Directed Simulated Annealing eingegangen wird, soll der prinzipielle Ablauf des Simulated Annealing vorgestellt werden:

Simulated Annealing:

Zielfunktion $Z = f(s), \quad s = (s_1, s_2, \ldots)$

Initialisiere $T = T_{\max}$

while nicht ausreichend optimal

- Wähle $s' = s + \Delta s$ mit zufälligem Δs
- **if** $f(s') < f(s)$ **then** Akzeptiere $s := s'$
- **if** $f(s') \geq f(s)$ **then**
  - Mit Wahrscheinlichkeit $p = e^{-\frac{f(s')-f(s)}{T}}$
    - Akzeptiere $s := s'$
- $T := T \cdot c, \quad 0 < c < 1$

Abbildung 5.1: *Prinzipielles Simulated Annealing*

Bei dieser Methodik handelt es sich um ein *globales Verfahren*, d.h. es wird stets der gesamte Lösungskandidat betrachtet und modifiziert. Der Name des Simulated Annealing geht aus den Naturwissenschaften hervor: Mit sinkender Temperatur und damit sinkender Brownscher Molekularbewegung wird die Energie, die zum Trennen bereits gebundener Moleküle in einer Schmelze zur Verfügung steht, geringer. Ausgehend von einer initial hohen Temperatur –in diesem Zustand sind die Moleküle nur lose arrangiert und können schnell und einfach ihre Position wechseln– lagern sich

die Elemente bei Abkühlung stärker aneinander an. Die intra-molekularen Bindungskräfte sind stets durch die Moleküle und deren relative Anordnung vorgegeben und der Brownschen Molekularbewegung entgegengesetzt. Die Wahrscheinlichkeit für eine Auflösung einer einmal vorgenommenen Anlagerung und damit für eine Erhöhung der gespeicherten Energie nimmt mit sinkender Temperatur ab, da zur Trennung weniger Energie zur Verfügung steht. Bei einer völligen Erstarrung der Schmelze haben sämtliche Moleküle sich angelagert, es hat sich energetisch ein (zumindest lokales) Minimum eingestellt. Dies kann zum Beispiel bei den regelmäßigen Strukturen in Metallstücken oder Eiskristallen beobachtet werden. Initial ist die Temperatur in der Schmelze hoch, bei Absenkung bis zum Gefrierpunkt jedoch ergeben sich die bekannten regelmäßigen energiearmen Strukturen

Funktionsprinzip *(Offline / Online)*	●	○[a]
Echtzeitverhalten *(Hard / Soft Realtime System)*	●	
Ausführungsumgebung *(Single / Multiple Processor)*	●	○[b]
Tasktypen *(Periodisch / Sporadisch bzw. einmalig)*	●	●
Ausführungszeiten *(Exakt / Beschränkt)*	●	
Release-Zeiten *(Absolut / Relativ)*	○[c]	●
Deadlines *(Absolut / Relativ)*	○[d]	●
Unterstützung Aperiodischer Tasks *(Offline/Online)*	○[e]	●

[a] Inkrementelles Scheduling von aperiodischen Tasks, Parameteränderung etc.
[b] Durch Verbreiterung des Trichters Anpassung an Mehrprozessorsysteme möglich, Kommunikationsaufgaben und Synchronisation bleiben dabei unberücksichtigt
[c] Für aperiodische Tasks
[d] Für aperiodische Tasks
[e] Maximerung der Idle-Zeiten

Abbildung 5.2: *Klassifizierender Überblick über das Hypersledge-Spring-Modell*

Übertragen auf die theoretische Ebene werden pro Iterationsschritt die Parameter willkürlich verändert. Führt dies zu einer Verbesserung des Zielfunktionswertes –die Gesamtenergie sinkt–, so wird der Schritt als eindeutig zielgerichtet akzeptiert und von dieser neuen Parameterkonstellation aus weitergesucht. Steigt die Energie im System durch die gewählte Variation jedoch an, ist der versuchte Schritt nicht direkt zielgerichtet. Abhängig von der aktuellen Temperatur und dem Grade der Verschlechterung wird der Schritt nur mit einer gewissen Wahrscheinlichkeit ausgeführt. Ohne diese Möglichkeit zur kurzzeitigen Verschlechterung würde die Methode nur durch Gradientenabstieg das nächstgelegene Minimum suchen – unabhängig davon, ob es ein globales oder nur lokales Minimum ist.

Experimentelle Ergebnisse auf anderen Gebieten zeigen, daß durch Simulated Annealing schnell gute Lösungen für ein Optimierungsproblem gefunden werden können, siehe dazu auch [RN95].

Angewandt auf das Scheduling von zyklischen Taskfolgen ergeben sich drei Probleme des Ansatzes über Simulated Annealing (SA):

- **Komplexe Modellierung**
  Die Darstellung des Scheduling-Problemes für die Anwendung eines SA-Algo-

rithmus ist nicht intuitiv. Zur Frage stehen etwa die Modellierung von Constraints, von zeitlichem, gegenseitigen Ausschluß und von Optimierungskriterien.

- **Ungültige Lösungen**
  Schedules können nicht nur nach ihrer "Qualität" beurteilt werden, vielmehr muß zwischen gültigen und ungültigen Schedules (Deadline-Überschreitungen, konkurrierende Taskinstanzen) unterschieden werden. Konventionelle SA-Algorithmen dienen nur zur Optimierung, nicht jedoch zur Erzeugung von zulässigen Ausgangslösungen.
- **Starke Interdependenzen**
  Ein Schedule stellt ein System mit sehr starken Interdependenzen dar. Änderungen an einer Stelle können Auswirkungen auf den gesamten Rest des Schedules haben. Bei einem partiell gültigen Schedule ist es daher nicht notwendigerweise wahrscheinlich, daß durch Änderungen im ungültigen Teil ein gültiger Gesamtschedule erstellt werden kann. Beispielsweise können dort vorgenommene Taskverschiebungen im gesamten Rest zu Deadlineüberschreitungen führen.

Zur Lösung dieser Probleme wird hier ein Modell mit Anleihen aus der Physik entwickelt: Sämtliche Taskinstanzen werden durch **Schlitten**[1] modelliert, welche untereinander durch **Federn**[2] gekoppelt sind. Die Wechselwirkungen durch die Federkopplung und durch Berührungen untereinander können über Gleitkräfte beschrieben werden, welche zu einem Rearrangement der Tasks auf der Zeitschiene führen.

Desweiteren wird, um das Problem der starken Interdependenzen zu lösen, der Abkühlungsprozeß im SA nicht gleichförmig sondern *gerichtet* durchgeführt. Bei dem –in der Literatur bisher nicht formulierten– Algorithmus des **Directed Simulated Annealing** (DSA) wird die Temperatur von einem "Ende" der Probe aus abgesenkt. Es wird somit eine Priorität für die Zielerreichung definiert. Zuerst werden nur einige Parameter der Lösung optimiert und dann nach und nach weitere Variablen in den Optimierungsprozeß mit einbezogen. Die Temperatur im Rest der Schmelze ist also während der Abkühlung noch so hoch, daß sämtliche Variationen der Zielfunktion zugelassen werden. Im Verlaufe des DSA schreitet jedoch die Abkühlungsfront weiter voran, bis schließlich die gesamte Probe erstarrt ist.

5.2.2 Hypersledge-Spring-Model

Zur Modellierung der Abhängigkeiten der Taskinstanzen untereinander wird ein der Physik angelehntes Modell verwendet.

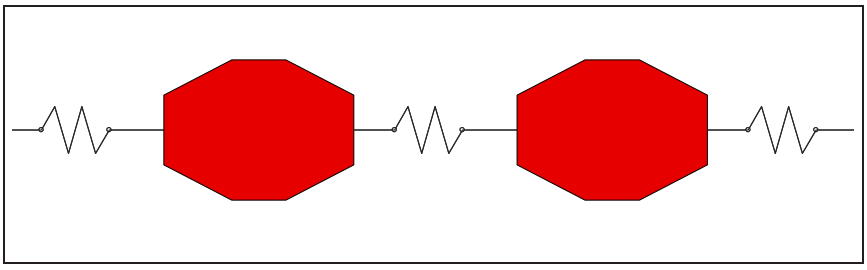

Abbildung 5.3: *Modellierung eines Jobs durch Schlitten für Taskinstanzen und Kopplungsfedern*

Es handelt sich hierbei strenggenommen um ein eindimensionales Modell: Horizontal wird die Zeit im Schedule festgehalten. Die Breite der die Taskinstanzen repräsen-

[1] **sledge** ['sledz] *n* Schlitten *m*
[2] **spring** ['sprin] *n (metal)* Feder *f.* **springy** *adj* federnd, elastisch

tierenden Schlitten ergibt sich aus der Ausführungszeit des Jobs. Analog werden die Federn zwischen benachbarten Instanzschlitten spezifiziert: Die relativen Constraints geben eine minimale (Releasezeit) und maximale (Releasezeit zuzüglich Länge des Start Time Windows) Ausdehnung vor. In der Visualisierung (Abbildung 5.3) ist die Releasezeit in zwei starre Elemente der Feder übertragen, welche sich jeweils rechts und links vom flexiblen Teil befinden. Im flexiblen Bereich sind Ausdehnungen bis zur Länge des Start Time Windows möglich.

Je nach Ausdehnungszustand der Federn wird horizontal eine Kraft auf beide gekoppelten Schlitten ausgeübt; Einzelheiten zur Definition der Kräfte im Teil 5.2.5. Je nach Zielkriterium liegt der neutrale Bereich der Feder bei minimaler, maximaler oder durchschnittlicher Dehnung.

Zur Modellierung von mehreren Jobs wird folgendermaßen vorgegangen: Jeder Job wird durch eine Reihe von Instanzschlitten dargestellt, jeweils mit oben angegebener Federkopplung. All diese Reihen werden zeitlich überlagert, was schließlich Beeinflußungen der Schlitten untereinander bedingt. Das Ziel für einen gültigen Schedule ist die Ausführbarkeit des Schedules auf einem einzelnen Prozessor, wobei sämtliche Tasks als nicht-unterbrechbar betrachtet werden. Pro Zeitabschnitt darf daher nur ein Taskschlitten vorhanden sein (Abbildung 5.4).

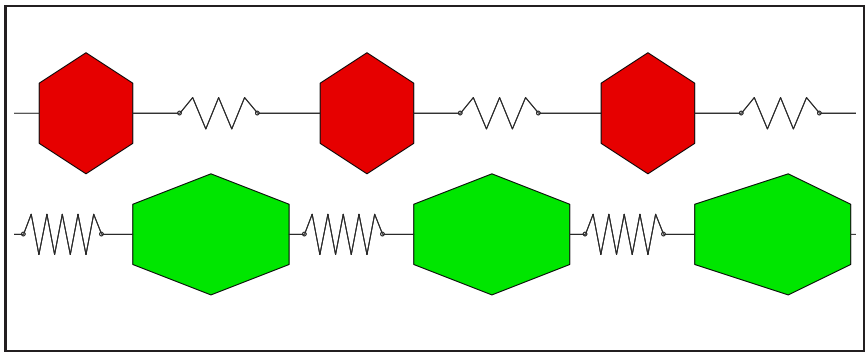

Abbildung 5.4: *Gültiger Schedule mit zwei Jobs und 3 respektive 2 Taskinstanzen. Die Schlittenform ist den angreifenden Federkräften angepaßt.*

Ein Schedule ist in diesem Modell genau dann gültig, wenn sämtliche Federn innerhalb ihres spezifizierten Bereiches gedehnt sind und die Schlitten sich zeitlich nicht überlappen. Um aus einem initial ungültigen Schedule einen erlaubten Plan zu generieren, werden bei der Schrittauswahl im SA-Algorithmus Heuristiken angewendet: Zwei sich nur leicht überlappende Schlitten erzeugen eine korrigierende Kraft (Abbildung 5.5), welche sich aus der Form der Schlitten ergibt.

Die äußere Form der Schlitten, siehe dazu auch 5.2.4, kann sich dynamisch verändern. Durch die Abschrägungen bedingt gleiten die Schlitten aneinander vorbei, wenn eine äußere vertikale Kraft vorhanden ist. Im Beispiel bedingt die Abschrägung eine Interferenzkraft nach links (oberer Schlitten) respektive nach rechts (unterer Schlitten). Zusammen mit den auf jeden Schlitten wirkenden Federkräften kann so eine Gesamtkraft berechnet werden. Diese ist ein Maß dafür, wie energieoptimal der Schlitten im aktuellen Schedule angeordnet ist. Überschreitungen der Federgrenzen oder Überlappung von Taskschlitten führen dabei zu sehr viel höheren Kräften als Federauslenkungen innerhalb des gültigen Bereichs.

Zu jedem beliebigen Zeitpunkt stehen dabei nicht nur die im Diagramm unmittelbar übereinander angeordneten Schlitten in Wechselwirkungen. Strenggenommen handelt es sich ja auch nur um eine zweidimensionale Visualisierung eines eindimensionalen Modelles, welches damit auch ungültige Schedules darstellen kann. Da sich in

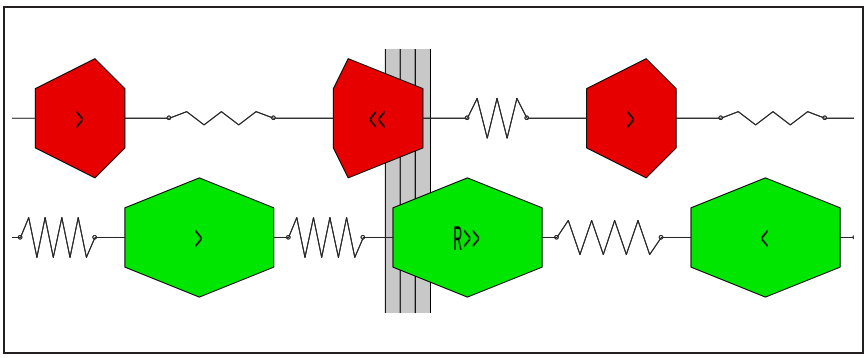

Abbildung 5.5: *Ungültiger Schedule. Zwei Schlitten überlappen im innerhalb des Federsystems direkt korrigierbaren Bereich*

dieser Darstellung nicht nur benachbarte Schlitten beeinflussen können, wird hier von **Hypersledges** gesprochen. Sie stehen mit *jedem* anderen Hypersledge auf gleicher Zeitebene in Kontakt.

5.2.3 Kraft- und Energiekonzept

Die Verwendung von **Feder-** und **Interferenzkräften** führt dazu, daß derselbe Algorithmus sowohl zur Generierung eines gültigen Schedules als auch zur iterativen Erzeugung eines energieoptimalen Planes genutzt werden kann. Die Kräfte bewegen sich dabei in zwei Bereichen: Dem Gebiet der zulässigen und dem der unzulässigen Kräfte (Abbildung 5.6). Wirken auf einen Schlitten unzulässig große Kräfte (etwa bei zeitlicher Überlappung oder Federüberdehnungen), so verursacht dieser Schlitten Constraintverletzungen. Insgesamt gilt damit für den gesamten Schedule, daß dieser genau dann gültig ist, wenn alle Einzelkräfte (und damit auch die Summe der Einzelkräfte) im zulässigen Bereich liegen.

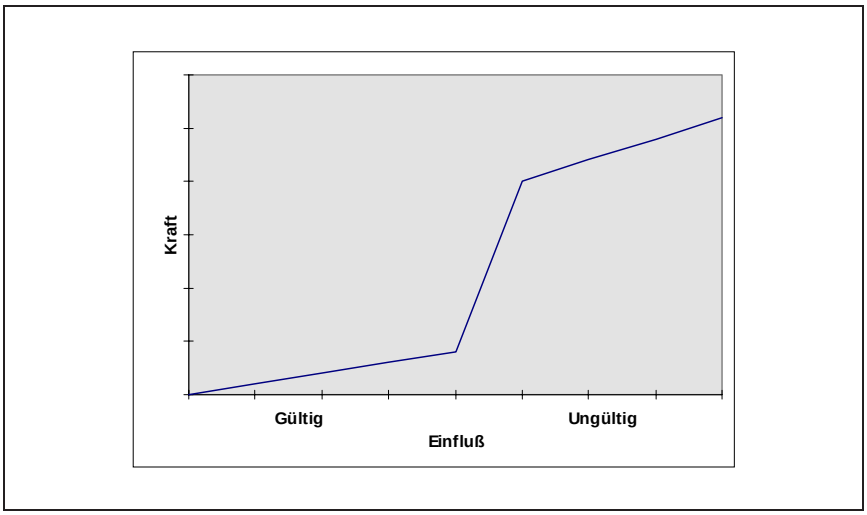

Abbildung 5.6: *Die Bereiche zulässiger und unzulässiger Kräfte im System*

Die Horizontalkräfte, welche auf die einzelnen Schlitten wirken, haben zwei Funktionen: Einerseits stellen sie die Energie des Systems dar, andererseits ermöglichen sie den Schluß auf einen möglichen Gradientenabstieg im Energiefeld.

Die Summe aller ungerichteten Kräfte im System ist ein Maß dafür, wieweit die Schlitten von ihrer Idealposition abweichen. Im gültigen Bereich entspricht dies der in den Federn gespeicherten Energie. Falls der Schedule jedoch insgesamt ungültig ist und die Kräftesumme somit im ungültigen Bereich liegt, läßt sie einen Rückschluß auf die Schwere der Constraintverletzungen zu – geeignete Kräftefunktionen vorausgesetzt. Aufgrund dieser Eigenschaften ist die *Summe der ungerichteten Kräfte* als **Energiefunktion** des Systems geeignet. Diese Vereinfachung entspricht nicht dem physikalischen Vorbild, ermöglicht jedoch eine einfache und effiziente Energiebetrachtung. Eine Minimierung der Energie führt zu einem gültigen Schedule, bei dem auch noch die Start Time Windows "regelmäßig" ausgenutzt sind. Es lassen sich hier kürzeste Schedules, längste Schedules oder auch flexibelste Schedules fordern; nähreres unter 5.2.5.

Bei der Auswahl eines Zufallsschrittes im DSA liegt es nahe, zuerst solche Parameter der Lösung (und damit Positionen von Schlitten) zu variieren, welche einen sehr starken Einfluß auf die Gesamtenergie haben. Die pro Schlitten angreifenden Kräfte gehen summarisch in die Gesamtenergie ein, so daß hier eine Präferenz definiert wird: Schlitten mit hohen angreifenden Kräften werden bevorzugt bewegt. Dadurch werden auch vorrangig Constraintverletzungen bearbeitet, bevor kleinere Optimierungen der Federauslenkungen vorgenommen werden.

5.2.4 Dynamische Form der Schlitten

Zur Beseitigung von zeitlichen Interferenzen läßt sich eine weitere Heuristik formulieren: Die oben vorgestellte Interferenzkraft gibt nicht nur an, wie stark die Interferenz ist (d.h. etwa, wie stark die Schlitten überlappen), sondern auch, in welche Richtung die Kräfte wirken. In Abbildung 5.5 ist durch die abgeschrägte Form intuitiv erkenntlich, daß eine Rechtsverschiebung des unteren Schlittens "sinnvoll" ist.

Andererseits wäre eine Rechtsverschiebung nicht sinnvoll, wenn die Feder zur Rechten bereits vollständig gestaucht wäre, eine Verschiebung also zu einer Constraintverletzung führen würde. In diesem Modell wird daher die Form des Schlittens dynamisch durch die angreifenden Federkräfte bestimmt. Die Schräge zu jeder Seite wird dabei so bemessen, daß im Falle eines "Aneinandervorbeigleitens" die angreifenden Federn nicht den zulässigen Bereich verlassen:

$$\begin{aligned} \text{Space}_{\rightarrow} &= \min\{\text{MaxStretch}_L,\ \text{MaxCompress}_R\} \\ \text{Space}_{\leftarrow} &= \min\{\text{MaxStretch}_R,\ \text{MaxCompress}_L\} \end{aligned}$$

MaxStretch und MaxCompress bezeichnen dabei die bei den Federn noch vorhandenen Spielräume. Aus den maximal zulässigen Bewegungen nach links und rechts kann auf die Länge der Abschrägungen geschlossen werden. Gilt

$$\text{Space}_{\rightarrow} + \text{Space}_{\leftarrow} \leq e,$$

so ist der gesamte Schlitten der Länge e (Execution Time) von Schrägfläche bedeckt, die Schrägflächen (*Ramps*, es wird die Breite der Basis angegeben) werden entsprechend gesetzt:

$$\begin{aligned} \text{Ramp}_R &:= \text{Space}_{\leftarrow} \\ \text{Ramp}_L &:= \text{Space}_{\rightarrow} \end{aligned}$$

Falls der Schlitten durch seine Federn nur wenig eingeschränkt ist, kann der Fall

$$\text{Space}_{\rightarrow} + \text{Space}_{\leftarrow} > e,$$

eintreten. In diesem Falle werden die Schrägflächen in jenem Verhältnis auf den Schlitten verteilt, in dem die Bewegungsräume zueinander stehen:

$$\begin{aligned} \mathrm{Ramp}_R &:= \frac{\mathrm{Space}_{\leftarrow}}{\mathrm{Space}_{\leftarrow}+\mathrm{Space}_{\rightarrow}} \cdot e \\ \mathrm{Ramp}_L &:= \frac{\mathrm{Space}_{\rightarrow}}{\mathrm{Space}_{\leftarrow}+\mathrm{Space}_{\rightarrow}} \cdot e \\ &= e - \mathrm{Ramp}_R \end{aligned}$$

Für einen Schlitten mit einer vertikal angreifenden, punktförmigen Kraft ergeben sich drei mögliche Reaktionen. Die punktförmige Kraft kann z.B. durch einen anderen Schlitten bedingt werden, welcher sich an dieser Stelle befindet. Greift die Kraft auf der linken oder rechten Schrägfläche an, so ergibt sich eine horizontale Verschiebungskraft nach rechts bzw. nach links. Die theoretische Bedeutung dabei ist, daß der Schlitten in die angegebene Richtung solange verschoben werden kann, bis die Kraft nicht mehr auf den Schlitten wirkt, *ohne* dabei die Feder-Constraints zu verletzen. Dies ergibt sich direkt aus der Definition der Schrägflächen. Wird die vertikale Kraft also von einem temporal fixierten Taskschlitten ausgeübt (etwa harte Periodizität), kann der andere Schlitten ausreichend zur Seite gleiten.

Ist die Belastung jedoch an einer nicht-abgeschrägten Stelle des Schlittens zu finden (vergleiche auch 5.8, Kraft 2), kann die Interferenz nicht ohne Verletzung anderer Constraints beseitigt werden. Dieser Fall muß später getrennt behandelt werden, da der Gradient der Zielfunktion an dieser Stelle nicht ermittelt werden kann.

5.2.5 Kraft- und Energiedefinition

An dieser Stelle soll zuerst die exakte Definition der Interferenz- und Federkräften erfolgen.

Interferenzkräfte treten auf, wenn zwei oder mehr Taskschlitten um dieselbe Position im Schedule konkurrieren (siehe Abbildung 5.5). Simuliert werden sie, indem sämtliche Schlitten gleichmäßig mit einer zentrierenden Kraft versehen werden. Befindet sich ein Schlitten mit keinem weiteren Schlitten in zeitlicher Überlappung, heben sich die allseitig angreifenden Kräfte auf[3], siehe Abbildung 5.7.

Findet dagegen eine zeitlichen Überlappung statt, so übt jeder Berührungspunkt –da *Hyper*sledges angenommen werden, ist jeder Überlappungspunkt ein Berührungspunkt– eine punktgerichtete Vertikalkraft auf den bzw. die anderen Schlitten aus. Dieser Berührungspunkt stellt ein einwertiges Lager dar (siehe [Sch92]), d.h. es kann nur *eine* Kraft aufgenommen werden. Diese muß zwangsläufig orthogonal zur Oberfläche verlaufen. In Abbildung 5.8 wird die angreifende Kraft V so in zwei orthogonale Kräfte zerlegt, daß F dabei senkrecht zur Oberfläche steht. Aus dieser Zerlegung kann dann wiederum auf die eigentliche Vortriebskraft H, welche in Horizontalrichtung wirken muß, geschlossen werden.

Zur exakten Berechnung dieser Kraft müssen einige geometrische Sätze zum Einsatz kommen: In Abbildung 5.9 wird die Schräge des Schlittens durch die Weite s und die Höhe h charakterisiert. Die angreifende Vertikalkraft ist durch V und die resultierende Vortriebskraft durch f bezeichnet: Die Dreiecke $D1$ und $D2$ sind ähnlich, da zwei Winkel (Rechter Winkel und α) übereinstimmen (Zweifacher Nebenwinkel). Aus dieser Ähnlichkeit folgt, daß die Seitenlängen im Dreieck $D2$ Vielfache von s und

[3] Genaugenommen stimmt dies nur, wenn die Schlitten zu beiden Seiten gleichmäßig abgeschrägt sind. Die durch asymetrische Schrägungen auftretenden Horizontalkräfte werden jedoch bereits durch die Federkräfte (exakter) modelliert und sind daher hier vernachlässigbar.

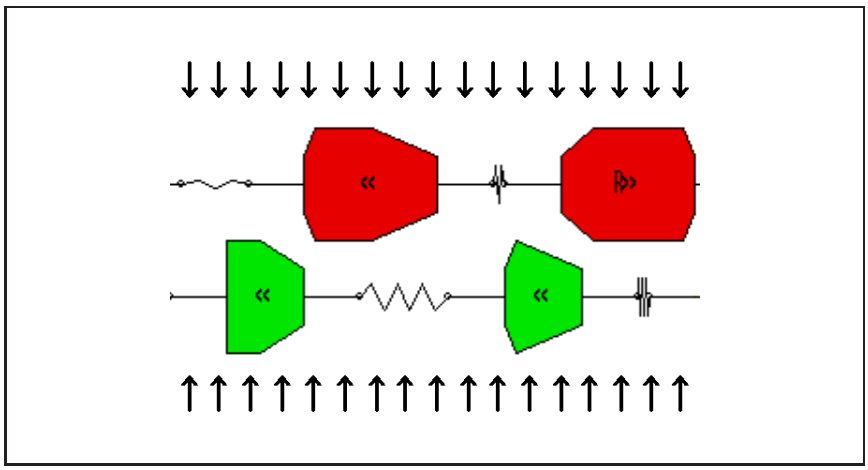

Abbildung 5.7: *Gleichmäßige zentrierende Kraft auf nicht-konkurrierende (links) und konkurrierende Schlittenpaare (rechts). Die Kraft ist dabei nicht von der vertikalen Position des Schlittens abhängig – es handelt sich um ein eindimensionales Modell mit Hyper-Sledges*

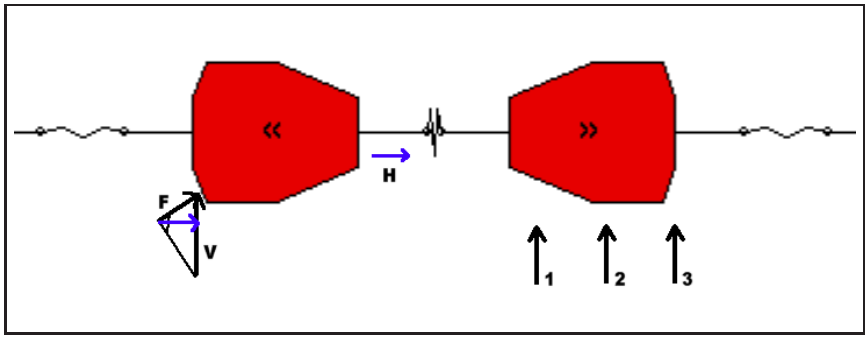

Abbildung 5.8: *Erzeugen der horizontalen Vortriebskraft aus vertikaler Preßkraft*

h sein müssen, hier durch cs und ch dargestellt, $c \in R$. Es gilt nach dem Satz des Pythagoras:

$$\begin{aligned} & V^2 &=& (ch)^2 + (cs)^2 \;=\; c^2h^2 + c^2s^2 \\ \Longrightarrow \quad & c^2 &=& \tfrac{V^2}{h^2+s^2} \end{aligned} \tag{5.1}$$

Der Flächeninhalt von $D2$ läßt sich auf zweierlei Weisen ausdrücken:

$$\begin{aligned} A(D2) &= \tfrac{V \cdot f}{2} \\ &= \tfrac{cs \cdot ch}{2} \;=\; \tfrac{c^2 \cdot s \cdot h}{2} \\ \Longrightarrow \quad Vf &= c^2hs \\ \Longrightarrow \quad f &= \tfrac{c^2hs}{V} \\ \overset{(5.1)}{\Longrightarrow} \quad f &= \tfrac{V \cdot h \cdot s}{h^2+s^2} \end{aligned}$$

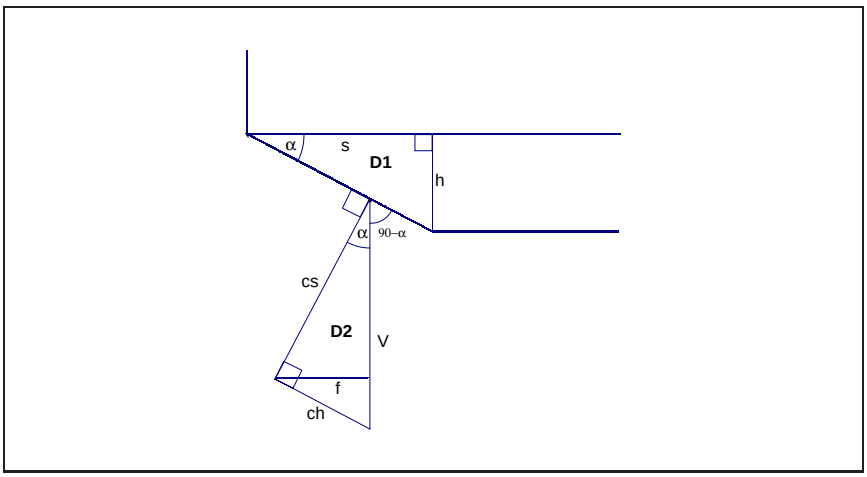

Abbildung 5.9: *Zur Berechnung der Vortriebs- aus der Anpreßkraft*

Die insgesamt ausgeübte Vertialkraft ergibt sich durch Integration über die Berührungsfläche. In diesem Modell wird dabei wieder streng eindimensional integriert, d.h. zwei Schlitten berühren sich genau dort, wo sie zeitlich überlappen – dies verändert jedoch qualitativ nicht das Resultat. Da die Vortriebskraft auf den Schrägflächen proportional zur ausgeübten Preßkraft ist, kann die kumulierte gerichtete Vortriebskraft pro Schlitten leicht durch Integration über beide Schrägflächen berechnet werden. Der zugrundeliegende Schedule ist wegen der Überlappungen klarerweise ungültig, daher wird die berechnete Kraft schließlich mit einem ausreichend großen Faktor skaliert, um das Erreichen des unzulässigen Bereiches (siehe Abbildung 5.6) zu garantieren.

Berühren sich hingegen zwei Schlitten an zwei waagerechten Seitenflächen, so kann der Konflikt nicht durch alleinige Verschiebung beider Schlitten gelöst werden (*Harter* Konflikt). Nach obiger Berechnungsvorschrift wirkt dann *keine* Horizontalkraft. Um jedoch der Modellvorstellung entsprechend durch die wirkenden Kräfte eine Verbesserung der Lösung bewirken zu können, werden Schlitten mit harten Konflikten willkürlich mit horizontalen Kräften beaufschlagt. Diese müssen im unzulässigen Bereich liegen, da eine Constraintverletzung vorliegt. Die Ausrichtung der Kraft ist nicht a-priori festzustellen, da der Gradient der Zielfunktion an dieser Stelle 0 ist. Die ausreichend große Kraft wird daher später zufällig orientiert.

Federkräfte werden zusätzlich durch die am Schlitten befestigten Federn ausgeübt. Für jede Feder ist die momentane Auslenkung gegeben. Zusammen mit den federintrinsischen Parametern (Minimale und maximale Elongation) kann die Federkraft auf drei Weisen definiert werden:

- *Kontrahierende* Feder
- *Zentrierende* Feder
- *Expandierende* Feder

Die nach dem *Hook*schen Gesetz ([Sch92]) berechnete Reaktionskraft ist proportional zur Auslenkung aus der Ruhelage (Abbildung 5.10). Diese Ruhelage befindet sich entweder bei minimaler, maximaler oder mittlerer Elongation. An den Grenzen des zulässigen Bereiches wird die Reaktionskraft zusätzlich additiv und multiplikativ verstärkt, da der ungültige Kräftebereich erreicht werden soll. Hierdurch wird auch die Einhaltung der Federgrenzen durch das Minimierungsproblem erfaßt.

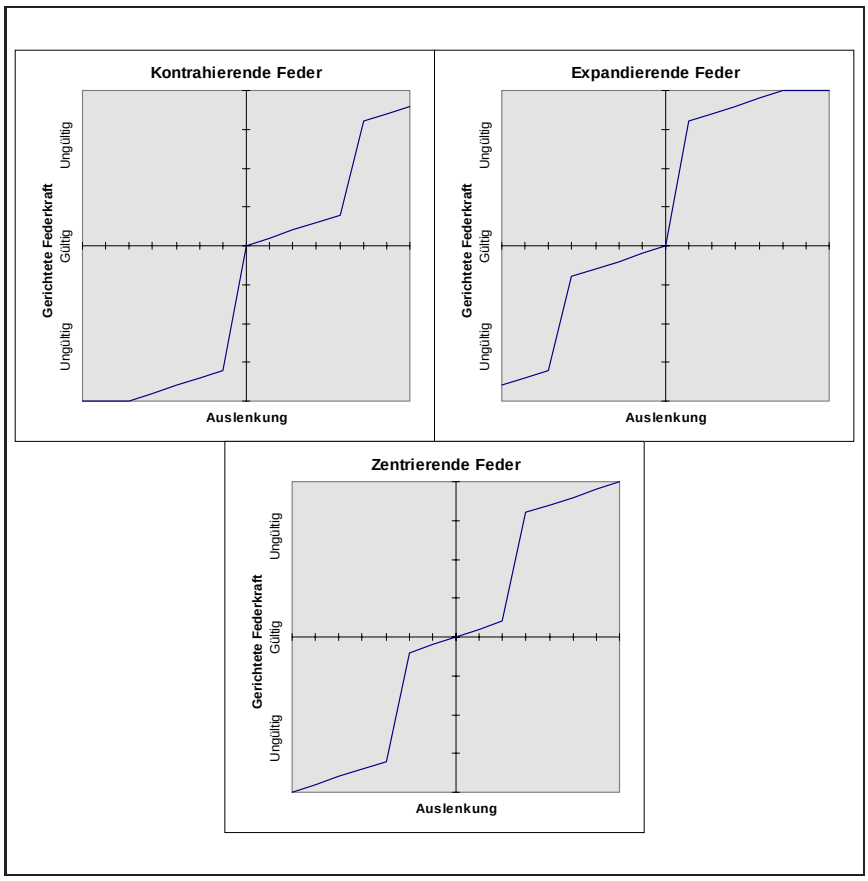

Abbildung 5.10: *Feder-Reaktionskräfte in Abhängigkeit von der Auslenkung einer kontrahierenden, expandierenden bzw. einer selbstzentrierenden Feder. An der spezifizierten Grenze der Dehnbarkeit steigt die Kraft sprunghaft in den ungültigen Bereich an.*

Bezogen auf das eigentliche Scheduling-Problem lassen sich durch die verschiedenen Federkraft-Definitionen grob drei Ziele verfolgen:

- Minimale Prozessorauslastung (Expandierende Federn, Deadlines werden soweit wie möglich ausgenutzt)
- Hohe Flexibilität bezüglich Hinzufügung weiterer periodischer, sporadischer oder einmaliger Tasks (Zentrierende Federn, jeder Schlitten hat maximale Bewegungsfreiheit innerhalb seines Start Time Windows)
- Höchste Ausführungsfrequenz (Kontrahierende Federn, der zeitliche Abstand zwischen zwei Taskinstanzen wird minimiert)

Erfahrungswerte zeigen, daß zur initialen Generierung eines zulässigen Schedules zentrierende Federn zu bevorzugen sind. Diese gewähren maximale Freiräume bei der gegenseitigen Taskbeeinflussung. Zur späteren Postoptimierung kann dann die Wirkungsweise der Federn entsprechend abgeändert werden.

Vorstellbar sind auch beliebig andere Auslenkungs-Kraft-Funktionen, etwa exponentieller Gestalt oder mit Energieminimum an anderen Stellen. Im Rahmen dieser Arbeit soll sich aber auf die drei linearen Grundtypen beschränkt werden.

Kraftübertragung ist ein weiteres Konzept, welches den Lösungsprozeß weiter optimiert. Sind die Schlitten bereits fertig angeordnet soll diese Ordnung nicht mehr verändert werden (s.u., lokal ist die Probe "abgekühlt"), so kann jeder Schlitten eine horizontale Druckkraft auf seine unmittelbaren Nachbarn weitergeben. Liegen etwa drei Schlitten lückenlos nebeneinander und wird auf den letzen Task eine Kraft nach links ausgeübt, etwa um eine Interferenz zu beseitigen, so wird diese Kraft durch die gesamte Schlittenkette propagiert. Dadurch wirkt auch auf das linke Element eine Kraft, so daß die ganze Reihe eine eventuell bestehende Lücke am linken Rand schließen wird, um die Interferenz am rechten Ende zu beseitigen. Auch diese *Force Propagation* entstammt direkt der physikalischen Wirklichkeit, in der dieselbe Kraftübertragung vorliegt.

5.2.6 Directed Annealing durch Trichter

Wie bereits in der Einführung (5.2.1) kurz beschrieben, ist es nicht sinnvoll, einen Schedule an allen Stellen gleichzeitig und mit gleicher Intensität zu optimieren. Kleine Änderungen an einer Stelle des Schedules können starke Auswirkungen auf den Rest haben und diesen sogar invalidieren. Das einfache Simulated-Annealing-Verfahren wird daher derart modifiziert, daß die "Abkühlung" der "Schmelze" gerichtet erfolgt.

Wie unter 5.2.3 beschrieben, stellt die ungerichtete Summe aller Schlittenkräfte ein Maß für die Systemenergie dar. Die beim Simulated Annealing (SA) beaufschlagte Externenergie entspricht der zentrierenden Energie in Form der gleichförmigen externen Kraft (Abbildung 5.7). Eine Abkühlung entspricht somit der steten Erhöhung der Zentrierungskräfte. Wenn Teile des Schedules gültig sind oder gültig werden, verliert die Zentrierungskraft an dieser Stelle ihre Wirkung, da ein solcher Schlitten nicht in Konkurrenz mit anderen Objekten steht und somit zentriert werden kann, um den Trichter zu passieren.

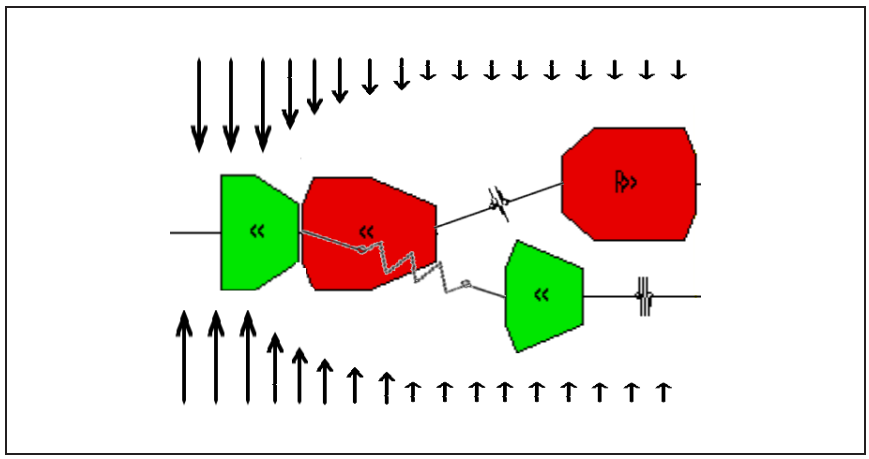

Abbildung 5.11: *Das Konzept des Directed Simulated Annealing durch einen Trichter visualisiert. Zwei Taskschlitten können bei Eintritt in Trichter angeordnet werden.*

Das Konzept des Directed Simulated Annealing (DSA) kann durch einen verschiebbaren Trichter visualisiert werden. Am dünnen Ende ist der Trichter derart schmal, daß nur ein Taskschlitten in der Breite passieren kann –nebeneinanderliegende, zeitlich überlappende Schlitten werden blockiert. Wird ein Trichter über die lose nebeneinander angeordneten Jobreihen (Systeme aus Schlitten und Federn) von links nach rechts

geführt und kommt es nicht zu einer Verklemmung, so wurden die Schlitten erfolgreich temporal arrangiert. Die den Trichter verlassende Taskfolge gibt einen gültigen Schedule an.

Die zentrierende Kraft wird nun in Abhängigkeit von der relativen Position des Trichters definiert. Am Auslaß des Trichters muß diese Kraft unendlich groß sein, da keine Schlittenkollisionen mehr möglich sein dürfen. Weit vor dem Trichter dagegen wird die Seitenkraft als sehr klein angesehen, an diesen Stellen spielen Konflikte keine vorrangige Rolle.

Mit zunehmender Nähe zum Trichter werde die Kraft linear vergrößert, die Temperatur der Probe im Sinne des SA also abgesenkt. Die Vermeidung von Schlittenkollisionen gewinnt dadurch immer höhere Priorität bis –im Erfolgsfalle– die Tasks geordnet den Trichter verlassen. Die Temperatur ist hier fast 0, es sind nur noch Kräfte im zulässigen Bereich gestattet. Dadurch wird gewährleistet, daß die Schlitten sich im durch die Federn beschränkten Bereich weiter bewegen können, dabei jedoch nie die physikalischen Grenzen überschritten oder Kollisionen herbeigeführt werden.

Falls es innerhalb des Trichters zu einer Verklemmung kommt, zwei Schlitten demnach in einer Art und Weise überlappen, die durch einfache Bewegungen einzelner Schlitten nicht behoben werden kann, so ist der Schedule auch in der näheren Umgebung zu variieren. Dazu wird die Temperatur erneut leicht angehoben. Dies ist eine weitere Abwandlung zum SA, da im einfachen Falle keine Zulässigkeitsbedingungen vorliegen. Die Erhöhung der Temperatur erfolgt beim DSA, indem die Abkühlungsfront leicht zurückgefahren wird. Mit dem Bild des Trichters gesprochen: Der Trichter wird ein Stück nach links bewegt, um wieder stärkere Variationen der Schlitten zuzulassen. Durch diese Methode konnten gute Resultate bei der Erlangung zulässiger Lösung erzielt werden. Es handelt sich somit strenggenommen um ein **hybrides Verfahren** aus Directed Simulated Annealing und **Backtracking**.

5.2.7 Simulationsgrundlagen

Die Simulation erfolgt über den aktuellen Schedule (damit die Position der einzelnen Schlitten) und die Trichterposition. Idealerweise müßte für jede Trichterposition eine optimale Anordnung der Schlitten erfolgen. Dies liefe auf das Lösen mehrerer Differentialgleichungen hinaus, was in diesem Zusammenhang nicht praktikabel erscheint. Desweiteren ist für die eigentliche Problemlösung eine energieoptimale Zwischenlösung nicht interessant, da eventuell noch Kollisionen vorkommen können. Außerdem ist die geschlossene Formulierung der Gleichungen nur bedingt möglich, da die Kräfteberechnung mehrere Fallunterscheidungen und auch die dynamische Form der Schlitten beachten muß.

Aus diesem Grunde wird der Weg über iterative Verbesserung des Energieinhaltes gewählt. Pro Schritt wird die Gesamtenergie als Summe der Einzelkräfte berechnet und nur ein Schlitten bewegt.

$$
\begin{aligned}
F_{Fi} &: \text{Federkräfte an Schlitten } i \\
F_{Ii} &: \text{Interferenz- und Propagationskräfte an Schlitten } i \\
E &= |F_{F1}| + |F_{F2}| + |F_{F3}| + \ldots + |F_{I1}| + |F_{I2}| + \ldots
\end{aligned}
$$

Diese Energiefunktion ist linear in sämtlichen Kraftkomponenten. Zur Energieminimierung wird ein Gradientenabstieg favorisiert, dazu müssen hohe Kraftanteile bevorzugt eliminiert werden. Es wird eine Wahrscheinlichkeitsverteilung proportional zu den Kraftanteilen pro Schlitten aufgestellt

$$
p_i = \frac{|F_{Fi}|+|F_{Ii}|}{E} \quad \forall i
$$

und gemäß dieser Verteilung ein Schlitten ausgewählt (**probablistische Auswahl**). Durch die Einteilung der Kräfte in zulässige und unzulässige Kräfte werden somit Constraintverletzungen häufiger ausgebessert als z.B. Federabweichungen von der Nullposition.

Für den gewählten Schlitten kann nun die eigentlich wirkende Horizontalkraft als

$$F \quad = \quad F_{Fj} + F_{Ij}$$

gerichtet berechnet werden. Diese gibt damit auch die Richtung an, in die die Bewegung des Schlittens einen Gradientenabstieg bedeutet, insofern der Gradient nicht Null ist. Ausgehend von dieser Richtungsbewegung können nun neue Kräfte auf diesen Schlitten nach Bewegung um t Zeiteinheiten angeben werden,

$$\begin{array}{rcl} & F & \stackrel{+1}{\longrightarrow} \quad F_1 \stackrel{+1}{\longrightarrow} F_2 \stackrel{+1}{\longrightarrow} F_3 \stackrel{+1}{\longrightarrow} \ldots \\ \Longrightarrow & 0 & \stackrel{+1}{\longrightarrow} \quad (F_1 - F) \stackrel{+1}{\longrightarrow} (F_2 - F) \stackrel{+1}{\longrightarrow} \ldots \end{array} \tag{5.2}$$

woraus sich in (5.2) direkt die lokale Kraftverringerung ergibt. Für mindestens einen Schritt liegt eine tatsächliche Verbesserung vor, da der Schlitten in Richtung der wirkenden Kraft bewegt wurde. Typisch ist jedoch das spätere erneute Zunehmen der Kraft, da etwa Federn an die physikalischen Grenzen stoßen oder Schlitten miteinander kollidieren werden.

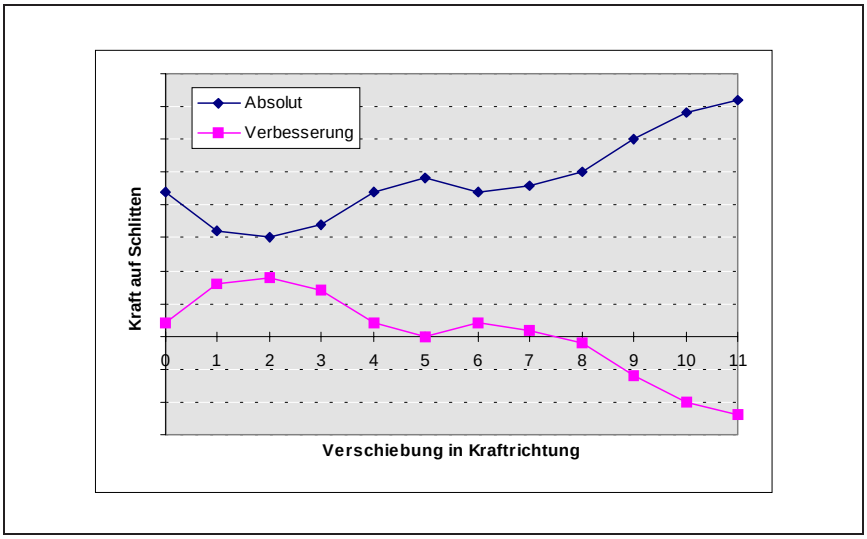

Abbildung 5.12: *Beispiel für die bei Verschiebung eines Schlittens auftretende Veränderung der lokal wirkenden Kraft*

Zu beachten ist dabei, daß nur die Position *eines* Schlittens verändert wird und nur lokal eine iterative Energieverbesserung vorgenommen wird. Durch Wechselwirkungen mit anderen Schlitten kann eine lokale Verbesserung auch zu einer globalen Verschlechterung führen. Durch die probablistische Auswahl wird dem jedoch Rechnung getragen. Dabei werden, wie beim SA üblich, auch energetische Verschlechterungen der Situation mit gewissen Wahrscheinlichkeiten zugelassen.

Innerhalb der lokalen Kraftverbesserung wird wiederum eine probablistische Auswahl der endgültigen Bewegungsdistanz vorgenommen: Je größer die Verbesserung, desto höher die Auswahlwahrscheinlichkeit. Der Suchbereich wird dabei auf die "nahe" Nachbarschaft eingeschränkt, also z.B. Bewegungen nur maximal um eine Schlittenlänge. Ebenso wirken die Nachbarn im Schedule begrenzend, da im Normalfalle ein

Schlitten nicht so weit bewegt werden soll, daß er mit einem anderen überlappt. Eine Ausnahme stellen hier Schlitten dar, auf welche bereits ungültige Kräfte wirken. Zur Beseitigung dieser Interferenzen sind dann auch Bewegungen möglich, die zu anderen Überlappungen führen, da ja die Ausgangsenergie des Schlittens schon sehr hoch ist.

Nach Ausführung der endgültigen Bewegung werden dann schließlich die Formen des Schlittens sowie seiner Nachbarn dynamisch den neuen Kräfteverhältnissen angepaßt. Nach einer globalen Neuberechnung der Energie im System kann die nächste Iteration begonnen werden.

Sinkt die Systemenergie in den zulässigen Bereich, d.h. sind alle Kräfte im System zulässig, so wurde ein gültiger Schedule gefunden. Dieser kann nun entweder verlängert werden oder im Postoptimierungsprozeß versucht werden, die Energie weiter abzusenken.

5.2.8 Dynamische Sledge-Erzeugung und -Entfernung

Obwohl auf die vom Trichter weit entfernten Schlitten nur geringe Kräfte wirken und diese somit den Iterationsprozeß nur marginal beeinflussen, soll die Schlittenanzahl im System möglichst gering gehalten werden. Da jedoch auch die für einen gültigen Schedule minimale Anzahl von Schlitten nicht a-priori bekannt ist, werden die Sledges dynamisch erzeugt und dem Schedule hinzugefügt bzw. bei Bedarf auch wieder entfernt. Dies geschieht z.B. dann, wenn der Trichter sich zu dicht am Ende der Jobsequenz befindet. Üblicherweise wird sichergestellt, daß der gesamte Trichterbereich mit Taskschlitten gefüllt ist.

Weitere Gründe für das dynamische Hinzufügen und Entfernen von Schlitten ergeben sich aus den sogenannten Job-Energien, s.u..

5.2.9 Zyklischer Schedule

Bisher wurden nur endliche Schedules bzw. durch unbeschränkten Generationsprozeß beliebig lange Schedules betrachtet. Für das unendliche Scheduling von zyklischen Tasks ist jedoch die Berechnung von zyklischen Schedules sinnvoller, siehe auch 2.3.

Zur Generation von solchen zyklischen Plänen mittels des DSA ist ein Feedback vom “Ende” des Schedules an den “Anfang” notwendig. Gemäß des Hypersledge-Modells erfolgt dieser Schluß durch Federn: Mit gegebener Zykluslänge können die jeweils ersten und letzten Schlitten jeder Jobreihe durch Feedback-Federn verbunden werden. Die Definition von Zyklus und Feedback verdient besonderes Augenmerk, da eine geringe Änderung am Ende des Zyklus große Auswirkungen auf den angekoppelten Beginn haben kann.

Nach Satz 3.4 läßt sich jeder gültige zyklische Schedule so darstellen, daß er mit einer beliebigen Taskinstanz eines beliebigen Jobs endet. Daher wird an dieser Stelle ein beliebiger Job als **Synchronisations-Job** definiert. Der Schedule bezüglich dieses Jobs spezifiziert eindeutig einen Zyklus. Die erste Jobinstanz ist mit einer den Jobspezifikationen (minimal Releasezeit, maximal Releasezeit zzgl. Start Time Window) entsprechenden Feder am fixen Zyklus-Start zum relativen Zeitpunkt $t = 0$ “befestigt”. Beendet wird der Zyklus mit dem Ende der letzten, zum Job gehörigen Taskinstanz. Durch diese Spezifikation wird garantiert, daß bei einer wiederholten Ausführung des Zyklus der Synchronisations-Job den Constraints genügt. Direkt nach Ende des letzten Tasks wird zum Zeitpunkt $t = 0$ erneut begonnen und die hier arretierte Start-Feder garantiert die Timing-Constraints.

Der so definierte Zyklus verändert sich somit mit der Anzahl der zum Synchronisations-Job gehörenden Anzahl von Instanzen und deren Anordnung im Schedule. Werden dynamisch neue Schlitten hinzugefügt oder entfernt, ändert sich umgehend auch der Zyklus.

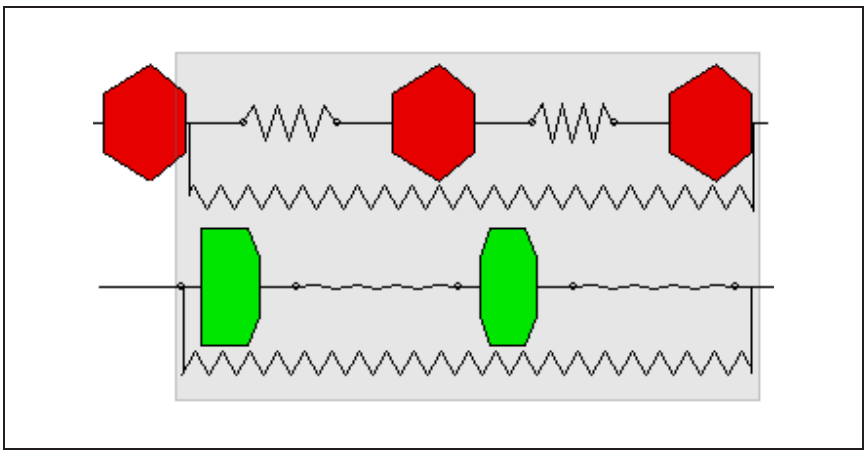

Abbildung 5.13: *Feedback-Federn in einem Zyklus gegebener Länge (Grau hinterlegt). Die Parameter der Feedback-Feder sind durch den Job definiert. Der außerhalb des Zyklus liegende Schlitten zeigt zur Verdeutlichung die Wiederholung der ersten Taskinstanz im Zyklus.*

Bezogen auf den dermaßen parametrisierten Zyklus können die Feedback-Federn der übrigen Jobs dimensioniert werden. Die dehnbare Länge bleibt durch die Länge des Start Time Windows konstant, der nicht-dehnbare Teil überbrückt dagegen rückwärtig die Zykluslänge. Diese minimale Elongation ergibt sich somit als $r-c$, d.h. Release-Zeit abzüglich der Zykluslänge, welches damit zu einer *negativen* Länge führt.

Wegen der durch den Synchronisations-Job festgelegten Zykluslänge entstehen mitunter schwere Constraintverletzungen bezüglich der anderen Jobs, weil etwa Deadlines weit überschritten werden, wenn der Gesamtzyklus verlängert wird. Aus diesem Grunde müssen auch **Job-Energien** berechnet werden. Dazu wird die Gesamtenergie des Systems auf die einzelnen Jobs verteilt. Jede Job-Energie setzen sich damit, genau wie die Systemenergie, additiv aus den absoluten Federkräften zusammen, welche auf zum Job gehörende Taskschlitten wirken. Ist diese Energie für längere Zeit in positiven bzw. negativen unzulässigen Bereich, so sind aktuell zu wenige respektive zu viele Taskinstanzen in den Zyklus eingeplant. Dies ergibt sich, da der überwiegende Teil der Federn zu stark komprimiert oder zu stark expandiert ist. Ebenfalls probablistisch wird hier in Abhängigkeit von der Job-Energie ein dynamisches Hinzufügen und Entfernen von Taskschlitten ermöglicht.

Ebenso müssen sämtliche Jobs einen Einfluß auf die Zykluslänge haben. Zur Bewertung dienen wiederum die Job-Energien. Für sämtliche Nicht-Synchronisations-Tasks werden die Energien aufsummiert und stellen ein Maß für die Güte der aktuellen Schedule-Länge dar. Hohe Energien bedingen eine Ausdehnung des Zyklus, hohe negative Energien eine Stauchung. Im Konzept der Hypersledges wird dieser Einfluß als eine zusätzliche Kraft auf den letzten Schlitten des Synchronisations-Task modelliert. Proportional zur berechneten Job-Energie-Summe wirkt eine gerichtete Kraft auf den Sledge und damit direkt auf das damit gekoppelte Zyklusende.

5.2.10 Fine-Tuning

Der Modellierung des Scheduling-Problemes durch das Hypersledge-Spring-Modell ist ein hohes Maß an experimentellen Untersuchungen vorausgegangen, um eine möglichst gute Modellstruktur, gute Wechselwirkungen sowie gute Berechenbarkeit und damit eine hohe Relevanz sämtlicher verwendeter Komponenten für die Zielerfüllung zu er-

langen. Neben vielen grundsätzlich verschiedenen Modelltypen, Wechselwirkungen und Simulationsmöglichkeiten wurden dabei auch einige Parametervariationen vorgenommen.

Das vorliegende, oben beschriebene Modell basiert bereits auf den daraus gewonnenen Erkenntnissen. Zum Beispiel erwies sich die Kopplung über Federn mit linearer Federkraft gegenüber einer exponentiellen Rückwirkung als geeigneter. Ebenso entstand die Idee der dynamischen Schlittenform oder der Interferenzkräfte erst während der Testphase mit anderen Repräsentationsformen. Daher ist schon die Definition des Modelles als eine Art "Tuning" anzusehen.

Nebenher haben sich einige Parameter des gewählten Modelles als "günstig" für den Simulationsprozeß erwiesen. Dazu gehören etwa das Verhältnis von Feder- zu Interferenzkraft, die Kraft vor und hinter dem Trichter, das Management von dynamischen Zyklenlängen und andere. Erst durch eine gute Wahl dieser Parameter erreicht der Simulator die vorliegende Leistungsfähigkeit.

Einige der oben angegebenen Parameter lassen sich im beiliegenden Simulator (siehe Abschnitt A.1.3) in eigenen Dialog-Fenstern variieren. Damit kann die Grundstruktur des Modells leicht an verschiedene Anforderungen angepaßt werden oder mit anderen Parameterkombinationen experimentiert werden.

5.2.11 Erweiterungsmöglichkeiten

Das vorgestellte Modell bietet noch viel Spielraum für Erweiterungsmöglichkeiten. An dieser Stelle sollen kurz einige Ideen angegeben werden, die die Leistungsfähigkeit der Methode weiter unter Beweis stellen können:

- Erweiterung auf Inter-Job-Constraints durch frei wählbare Federangriffspunkte
- End-to-End-Design durch Hinzufügung absoluter, fester Zeitpunkte und beschränkender Federn, welche einzelne Taskschlitten mit diesen Zeitmarken verknüpfen können (etwa: Zwischenresultat stets 10 Zeiteinheiten nach Zyklen-Beginn liefern)
- Verbreiterung des Trichter-Ausgangs und somit einfache Multiprozessor-Unterstützung, es können dann mehrere Taskschlitten nebeneinander den Trichter passieren
- Mehrere (fertig erstellte) Zyklen kombinieren
- A-priori-Prüfung auf mögliche Nicht-Schedulability, etwa durch Utilisation-Kriterium. Der Iterationsalgorithmus kann das Set dann sofort zurückweisen.

Das Hauptaugenmerk dieser Arbeit liegt jedoch nicht auf einer weitestgehenden Optimierung des Modelles sondern darauf, ein adäquates Modell zu entwickeln und die Eignung zur Lösung des gestellten Problems unter Beweis zu stellen. Sämtliche angegebenen Erweiterungsmöglichkeiten dürfen daher nur als Ausblick auf zukünftige Weiterentwicklungen angesehen werden.

Kapitel 6

Beispiele und experimentelle Ergebnisse

6.1 Zyklen mit multiplen Taskinstanzen

Im Falle von $r_i \equiv 0$, d.h. trivialen Release-Zeiten, läßt sich jeder gültige Zyklus von Tasks so darstellen, daß mindestens ein Task nur genau einmal vertreten ist. Dies beschränkt den Prüfaufwand beim Cycle Checking.

Im nichttrivialen Fall von beliebigen Release-Zeiten gilt dies nicht, was durch folgendes Beispiel gezeigt wird:

$$n = 2; \quad e_1 = e_2 = 1; \quad r_1 = 2, \ \delta_1 = 0; \quad r_2 = 3, \ \delta_2 = 1$$

führt einsichtig und eindeutig zu folgendem minimalen Zyklus (Abbildung 6.1): Anschaulich wird J_1 mit einer festen Frequenz ausgeführt und J_2 startet abwechselnd nach 1 bzw. 2 Instanzen von J_1. Dabei ist hier kein Task mit nur einer Instanz im

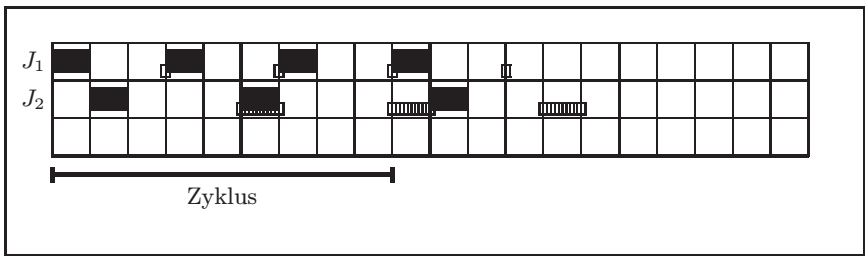

Abbildung 6.1: *Zyklischer Schedule, für den es kein Äquivalent gibt, in dem ein Task nur einfach vertreten ist. Zu beachten ist das Start Time Window von J_1 der Länge 0, die Periodizität des Jobs ist also eindeutig festgelegt.*

Zyklus vertreten, so daß im allgemeinen Falle alle Vorkommen eines Tasks beim Cycle Checking mit einbezogen werden müssen.

6.2 Idle-freie Zyklen multipler Taskinstanzen

Ein ähnlicher Fall wie unter 6.1 kann auch ohne Leerlaufzeiten konstruiert werden, so daß also konsekutive Jobs sofort nach Ende des Vorgängers gestartet werden. Auch in diesem Falle kann kein vereinfachtes Cycle Checking durchgeführt werden.

$$n = 3; \quad e_1 = e_2 = e_3 = 1; \quad r_1 = 3; \quad \delta_1 = 0; \quad r_2 = r_3 = 1; \quad \delta_2 = \delta_3 = 1$$

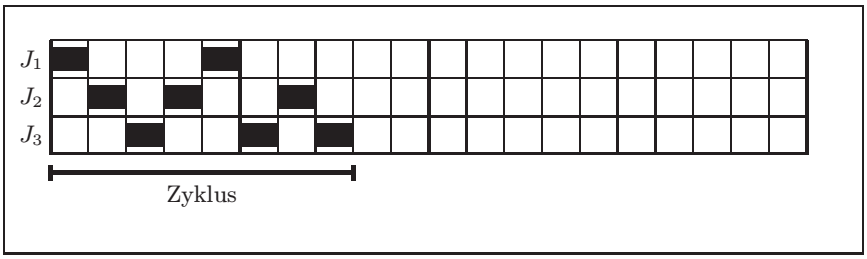

Abbildung 6.2: *Zyklischer lückenloser Schedule, für den es kein Äquivalent gibt, in dem ein Task nur einfach vertreten ist*

6.3 2-Job-Probleme

Zur Überprüfung der in Abschnitt 3.4 vorgestellten Kriterien und Heuristiken sowie zur Ermittlung von deren Leistungsfähigkeit wurden mittels eines Branch-and-Bound-Verfahren (siehe A.2) verschiedene Job-Sets überprüft.

Für die Job-Spezifikation

$$J_1 = (e_1, r_1, \delta_1), \quad J_2 = (e_2, r_2, \delta_2)$$

wurden parametrische Einschränkungen durch

$$\begin{aligned} e_1, e_2 &\in [1, 4] \\ r_1, \delta_1, r_2, \delta_2 &\in [0, 4] \end{aligned}$$

vorgenommen. Es erfolgt ein Vergleich der durch die weiter oben beschriebenen Kriterien gewonnenen Resultate mit der tatsächlichen Schedulability, ermittelt durch den Branch-and-Bound-Algorithmus.

Zur besseren Übersichtlichkeit werden von sämtlichen Tasksets zuerst nur Teilmengen betrachtet. Die Einschränkungen werden dann nach und nach aufgehoben bis schließlich sämtliche $(4^2 \cdot 5^4) = 10000$ Sets in die Berechnung einfließen.

6.3.1 Testsets mit $r_1 = r_2 = 0$

In Satz 3.6 (Seite 28) wird ein notwendiges und hinreichendes Kriterium für 0-Release-Zeiten angegeben. Insgesamt gibt es durch die oben definierte Einschränkung nur 400 Task-Sets, welche die $r_1 = r_2 = 0$ -Bedingung erfüllen. Leicht zu sehen ist in diesem einfachen Fall, daß sämtliche 400 Sets korrekt in planbare und nicht-planbare Mengen separiert werden. In der Tabelle 6.3 ist dazu oben die im Branch-and-Bound ermittelte tatsächliche Schedulability dem Kriterium (links) gegenübergestellt.

		Schedulability	
		TRUE	FALSE
Kriterium	TRUE	100	0
	FALSE	0	300
	N/A	5898	3702

Abbildung 6.3: *Experiementelle Ergebnisse im Falle* $r_1 = r_2 = 0$

Sowohl Kriterium als auch Branch-and-Bound besitzen selbstverständlich boolesche Ergebniswerte (TRUE und FALSE). Da nicht sämtliche möglichen Sets betrachtet werden, müssen die nichtanwendbaren (*N/A, not applicable*) Sets ausgesondert werden. Diese können in der letzten Tabellenzeile getrennt ausgewiesen werden.

Die Güte eines Kriteriums bzw. einer Heuristik ergibt sich also aus drei Teilkomponenten:

- *Großer Definitionsbereich*
  Das Kriterium soll auf möglichst viele (verschiedene) Test-Sets angewandt werden können. Die letzte N/A-Zeile der Tabelle soll demnach möglichst kleine Werte, bevorzugt 0, aufweisen.
- *Notwendigkeit des Kriteriums*
  Ein notwendiges Kriterium muß für alle tatsächlich schedulablen Job-Sets erfüllt sein, d.h. die Implikation "*Schedulability*→*Kriterium*" muß gelten. In der Tabelle soll daher der mittlere Eintrag in der ersten Spalte möglichst klein sein.
- *Hinreichende Güte des Kriteriums*
  Hinreichende Kriterien dürfen nur für solche Sets erfüllt sein, die tatsächlich schedulbar sind. Die Implikation lautet hier "*Kriterium*→*Schedulability*". In der Tabelle soll somit der Wert oben rechts minimal sein.

Eine Zusammenfassung der Kriterien ergibt, daß die Ähnlichkeit mit einer 2×2 Diagonalmatrix im oberen Teil der Tabelle gute Kriterien charakterisiert. Es sollen an dieser Stelle jedoch keine allgemeinen Kenngrößen definiert werden, sondern nur Ergebnisse kurz vorgestellt werden.

6.3.2 Testsets mit $r_1 = 0$

Für den Fall, daß eine Release-Zeit 0 ist, greift Satz 3.9.

		Schedulability	
		TRUE	FALSE
Kriterium	TRUE	788	0
	FALSE	0	1212
	N/A	5210	2790

Abbildung 6.4: *Experimentelle Ergebnisse im Falle* $r_1 = 0$

Auch hier zeigt sich das Kriterium im abgedeckten Bereich sowohl hinreichend als auch notwendig.

Aufgrund der Symmetrie des Problems bezüglich der Release-Zeiten werden natürlich auch solche Test-Sets abgedeckt, bei denen mindestens eine der beiden Releasezeiten 0 ist, also $r_1 = 0 \ \vee \ r_2 = 0$. Von den 10000 möglichen Konstellationen kann das Kriterium dann immerhin auf 36% angewandt werden.

		Schedulability	
		TRUE	FALSE
Kriterium	TRUE	1476	0
	FALSE	0	2124
	N/A	4522	1878

Abbildung 6.5: *Experimentelle Ergebnisse im Falle* $r_1 = 0 \ \vee \ r_2 = 0$

6.3.3 Testsets mit $\delta_1 = \delta_2 = 0$

Basierend auf Satz 3.10 konnte wiederum ein Kriterium implementiert werden:

		Schedulability	
		TRUE	FALSE
Kriterium	TRUE	34	0
	FALSE	0	366
	N/A	5964	3636

Abbildung 6.6: *Experimentelle Ergebnisse im Falle* $\delta_1 = \delta_2 = 0$

6.3.4 Testsets mit $\delta_1 = 0$

Interessanter wird dagegen die Überprüfung der Heuristik aus Satz 3.11. Wiederum werden bei Berücksichtigung der Symmetrie 36% des Problemraumes abgedeckt. Klar erkenntlich ist, daß es sich um ein *notwendiges*, jedoch nicht *hinreichendes* Kriterium handelt. In 148 von 3600 Fällen (symmetrischer Fall aus Abbildung 6.8), d.h. nur circa 4%, ergibt sich ein falsches Resultat.

		Schedulability	
		TRUE	FALSE
Kriterium	TRUE	622	74
	FALSE	0	1304
	N/A	5376	2624

Abbildung 6.7: *Experimentelle Ergebnisse im Falle* $\delta_1 = 0$

		Schedulability	
		TRUE	FALSE
Kriterium	TRUE	1210	146
	FALSE	0	2244
	N/A	4788	1612

Abbildung 6.8: *Experimentelle Ergebnisse im Falle* $\delta_1 = 0 \ \vee \ \delta_2 = 0$

Inkorrekte Resultate liefert das Kriterium etwa bei $J_1 = (1, 4, 2),\ J_2 = (4, 4, 1)$. Dies sind all jene Fälle, in denen die Ausführungszeiten zwar prinzipiell in die Lücken der anderen Ausführung passen, durch kleine relative "Verschiebungen" gegeneinander jedoch im späteren Verlauf Überschneidungen auftreten. Bezogen auf die relativ hohe Trefferquote des Kriteriums lassen sich etwa beim inkrementellen Scheduling von neu hinzukommenden Tasks bereits ca. 62% der Fälle *ohne* nähere Betrachtung zurückweisen. Bei Bearbeitung der verbleibenden 37% wird dann nur in 10% der Fälle ein Mißerfolg eintreten. Absolut auf das gesamte betrachtete Problemfeld bezogen sind das nur ca. 4%.

6.3.5 Allgemeiner Fall

Werden alle gewonnenen Erkenntnisse aus den Sätzen in ein Kriterium kombiniert, wobei jedes Unterkriterium jeweils entsprechend seines Definitionsbereiches verwendet wird (siehe 3.4.7), ergibt sich eine globale Schedulability-Heuristik für 2-Job-Probleme.

Für oben parametrisierte Jobsets ergibt sich eine Trefferquote von 97.5%:

		Schedulability	
		TRUE	FALSE
Kriterium	TRUE	5998	250
	FALSE	0	3752
	N/A	0	0

Abbildung 6.9: *Experimentelle Ergebnisse im allgemeinen Falle*

Um eine gewisse Skalierbarkeit zu zeigen, werden die Beschränkungen an die Job-Parameter nun gröber gefaßt:

$$\begin{aligned} e_1, e_2 &\in [1, 10] \\ r_1, \delta_1, r_2, \delta_2 &\in [0, 10] \end{aligned}$$

		Schedulability	
		TRUE	FALSE
Kriterium	TRUE	907388	61728
	FALSE	0	494984
	N/A	0	0

Abbildung 6.10: *Experimentelle Ergebnisse im allgemeinen Falle, großes Test-Set*

Von den etwa 1.46 Millionen Job-Sets, die in diesem Experiment betrachtet wurden, hat das Kriterium immerhin 1.4 Millionen Sets korrekt klassifiziert. Dies ergibt eine Trefferquote von fast 96%. Insgesamtgesehen ist die Verwendung des kombinierten Kriteriums, dessen Auswertung nur einen Zeitbedarf von $O(1)$ besitzt, bei 2-Job-Problemen mit relativen Timing-Constraints sehr empfehlenswert. Die Separation in planbare und nichtplanbare Gruppen erfolgt annehmbar zuverlässig.

6.4 Hypersledge-Spring-Modell

Zur experiementellen Bewertung und Beurteilung des in Abschnitt 5.2.2 definierten Hypersledge-Spring-Modells werden an dieser Stelle einige Problem-Sets simuliert und die Ergebnisse entsprechend visualisiert.

Simulationsgrundlage ist das unter A.1 vorgestellte Tool, mit welchem auch die gezeigten Schedules erzeugt wurden. Auf beiliegender Daten-CD finden sich sowohl das Simulations-Tool als auch die zugrundeliegenden Taskspezifikationen.

6.4.1 Einfache Schedules

Minimaler Zyklus

Für ein einfaches 4-Job-Problem mit den Parametern

$$\begin{aligned} J_1 &= (2, 6, 28) \\ J_2 &= (3, 7, 30) \\ J_3 &= (7, 9, 26) \\ J_4 &= (3, 6, 30) \end{aligned} \tag{6.1}$$

ergibt sich nach etwa 20 Simulationsschritten der in Abbildung 6.11 gezeigte Schedule. Es handelt sich dabei um einen *minimalen* Schedule, d.h. es sind so wenig Taskinstanzen wie möglich verplant. Die Auslenkung der Federn ist durchschnittlich, so daß auch die zeitliche Länge des Zyklus durchschnittlich ist. Eine geringe Abweichung von der optimalen Konfiguration ist durch die verbleibende Systemenergie von $E = 30$ zu erkennen. Für die meisten Probleme ist jedoch die völlige Vermeidung von Restenergie nicht möglich, da dies nur dann der Fall ist, wenn *keine* Feder eine Rückstellkraft ausübt – die Jobs müßten dann etwa immer genau in der Mitte ihrer Start Time Windows gestartet werden.

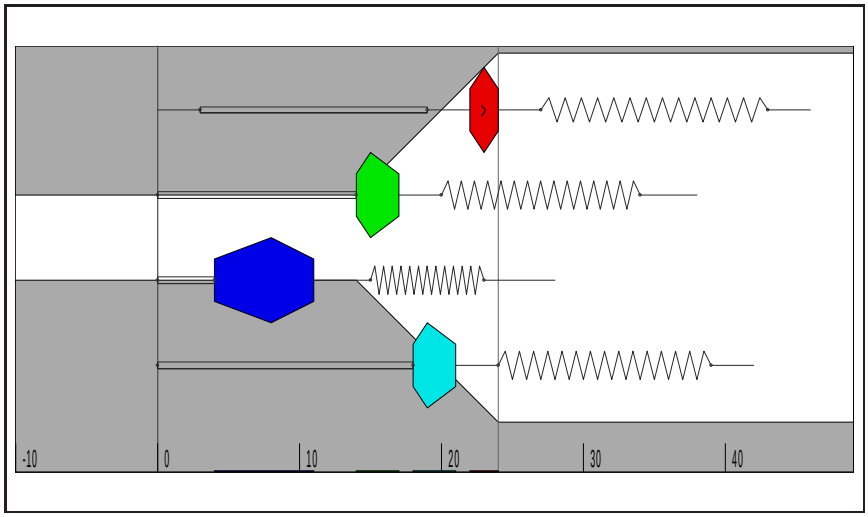

Abbildung 6.11: *Task-Set* `bsp-1.tsk` *(Rest-Systemenergie: 30)*

Redundante Verlängerung des Zyklus

Durch entsprechend lange Simulation kann der Zyklus künstlich verlängert werden. Nach etwa 100 Simulationsschritten ergibt sich folgender Schedule. Zu beachten ist dabei, daß die Anzahl der Task-Instanzen nicht konstant ist: In 6.11 war jede Instanz einfach vertreten, hier ist der erste Task 4-mal eingeplant, die restlichen drei jedoch nur 3-mal.

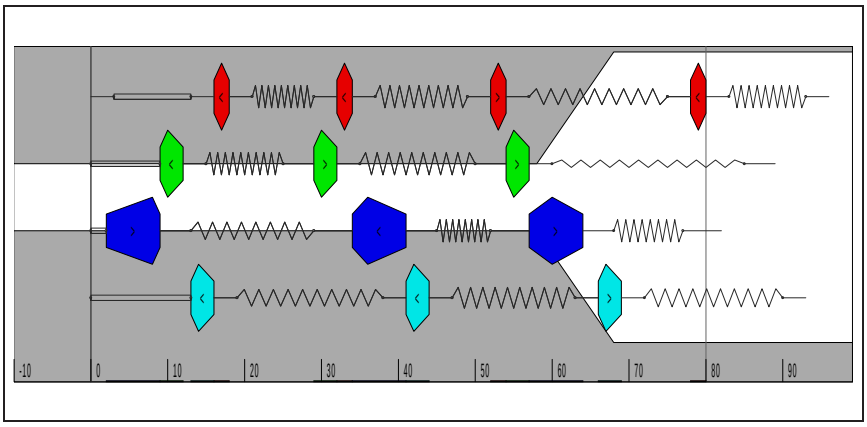

Abbildung 6.12: *Task-Set* `bsp-1.tsk`*, längerer Zyklus (Verbleibende Rest-Systemenergie: 24)*

Eine weitere Verlängerung der Simulationszeit erweitert den Zyklus immer mehr, so daß auch *ohne* explizite Definition eines Zyklus unendlich lange Schedules erzeugt werden könnten.

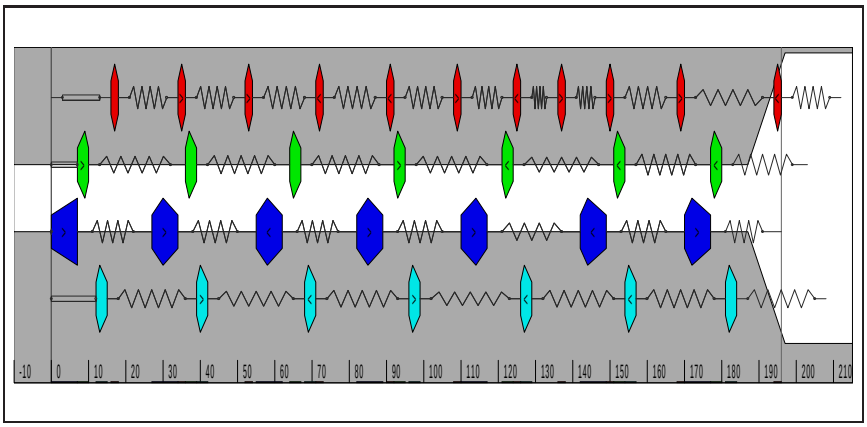

Abbildung 6.13: *Task-Set* `bsp-1.tsk`*, willkürlich verlängerter Zyklus (Rest-Systemenergie: 53)*

6.4.2 Große Anzahl an Jobs

Die Anzahl der Jobs kann stark vergrößert werden. Im folgenden werden Beispiele mit 7 respektive 9 Jobs gezeigt, wobei die weitere Hinzufügung von Jobs nur durch den Speicherplatz und den in der Grafik zur Verfügung stehenden Platz limitiert wird.

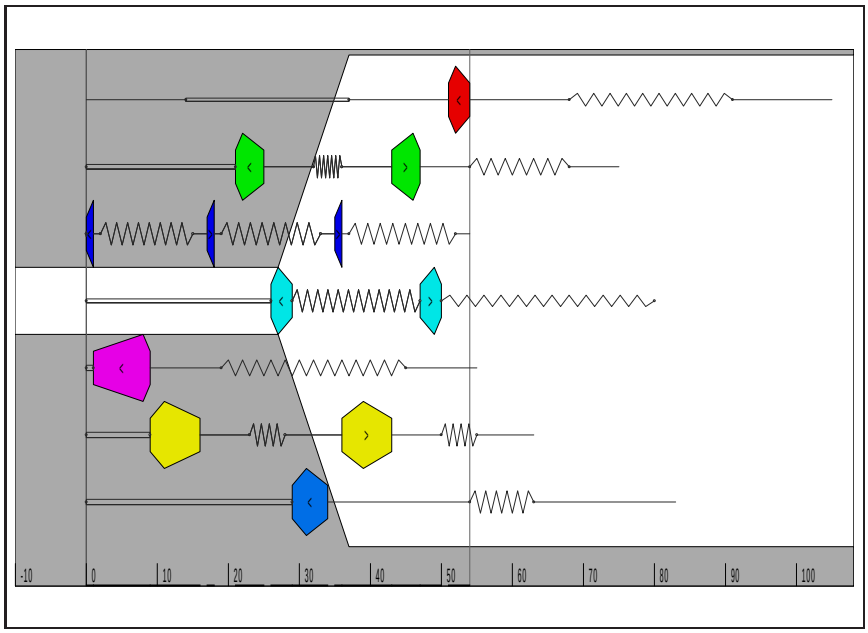

Abbildung 6.14: *Task-Set* `bsp-2.tsk` *mit 7 Jobs, stark unterschiedliche Periodizität (Rest-Systemenergie: 27)*

Bemerkenswert ist in dem gezeigten Falle auch die stark unterschiedliche Periodizität: Einige Jobs sind nur mit einer Instanz vertreten, wohingegen J_3 schon mit drei Ausführungen eingeplant werden muß, um alle Constraints einzuhalten. Außerdem ist der entstandene Schedule relativ dicht, so daß im Simulationsprozeß nur wenig Freiraum für andere Kombinationen bestand. Entsprechend waren in diesem Falle auch circa 120 Simulationsschritte erforderlich, um das gezeigte Ergebnis zu generieren.

Durch Modifikation der Execution-Time für den vorletzten Job kann der Schedule weiter verdichtet werden. Alleine J_6 okkupiert bereits ca. 20% der verfügbaren Rechenzeit im Schedule.

Eine Erweiterung auf 9 Jobs ist in Abbildung 6.16 gezeigt, 14 Jobs dagegen in 6.17.

Bei Verwendung beiliegender CD-Rom können die Simulationen, die zu oben gezeigten Ergebnissen geführt haben, leicht selbst nachvollzugen werden. Es ist dabei nur notwendig, die referenzierten Beispiel-Dateien zu laden und die Simulation zu starten.

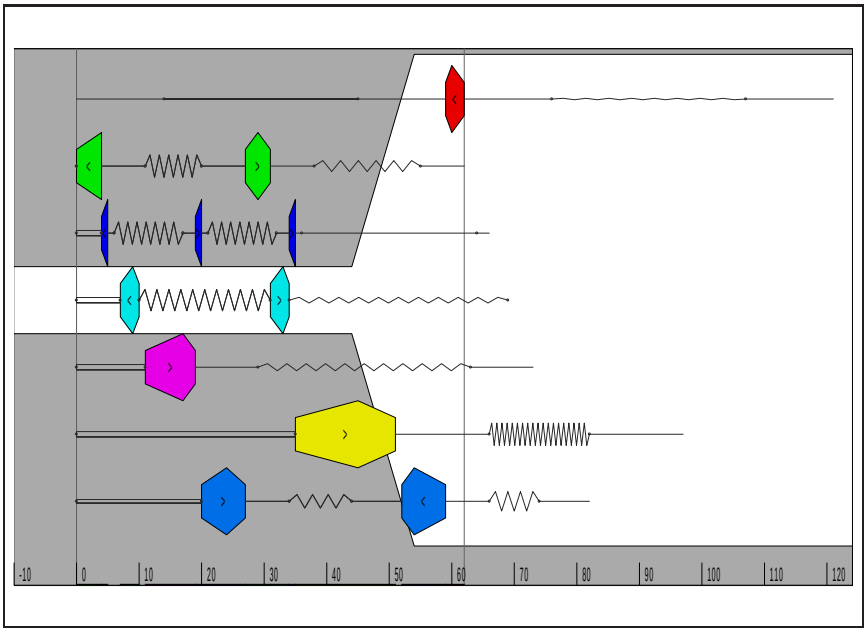

Abbildung 6.15: *Task-Set* `bsp-2.tsk`*, modifiziert (Rest-Systemenergie: 280)*

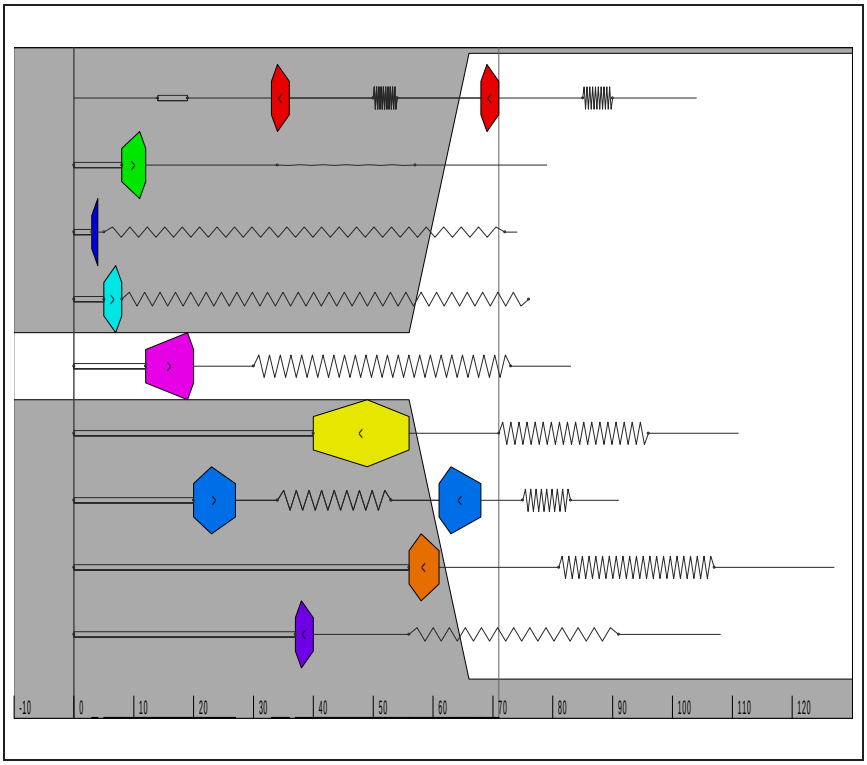

Abbildung 6.16: *Task-Set* `bsp-3.tsk`*, 9 Jobs (Rest-Systemenergie: 29)*

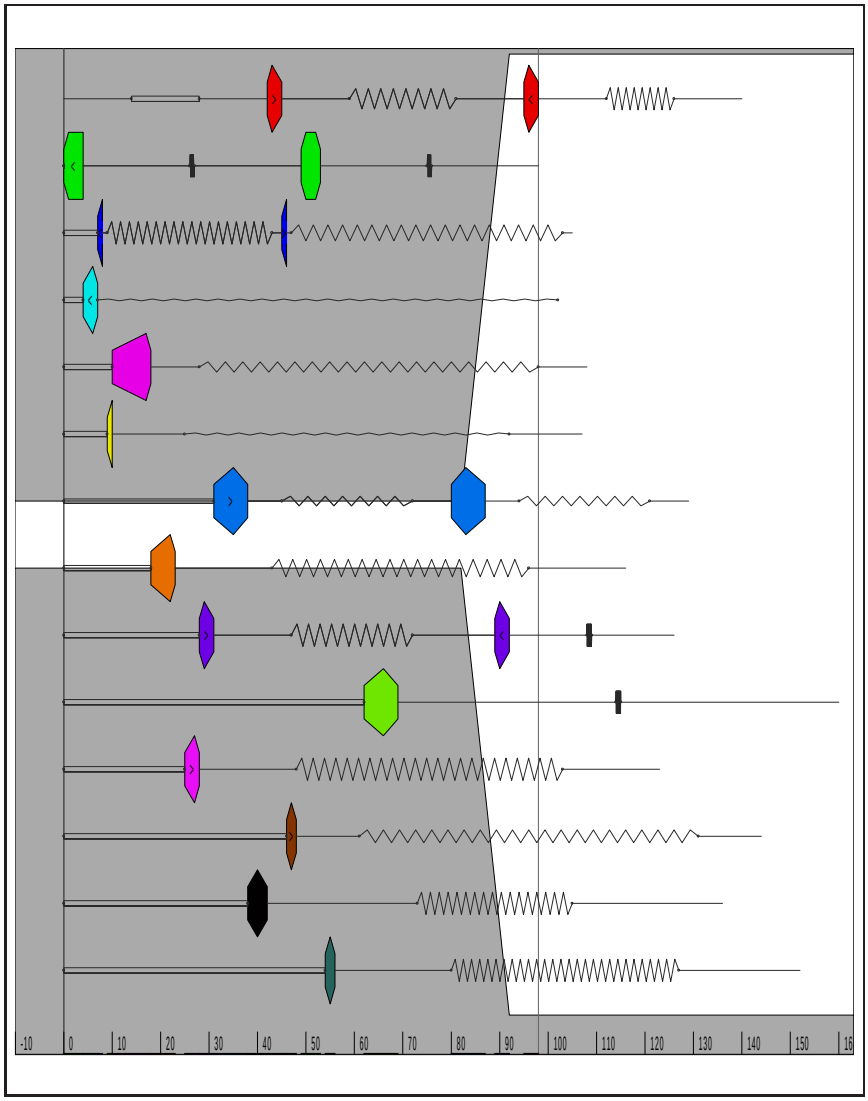

Abbildung 6.17: *Task-Set* `bsp-4.tsk`*, 14 Jobs (Rest-Systemenergie: 7)*

6.4.3 Trichter-Variation

Wird der Trichter stark verkürzt und die Vertikalkraft vor dem Trichter ausgeblendet, arbeitet die Simulation sehr "kurzsichtig". Nur auf solche Taskschlitten, die den Trichter erreicht bzw. passiert haben, wirkt eine Interferenzkraft. Weit vor der Öffnung des Trichters wird somit eine zeitliche Überlappung nicht beachtet – alleine die Federn bewirken eine periodische Anordnung der Schlitten.

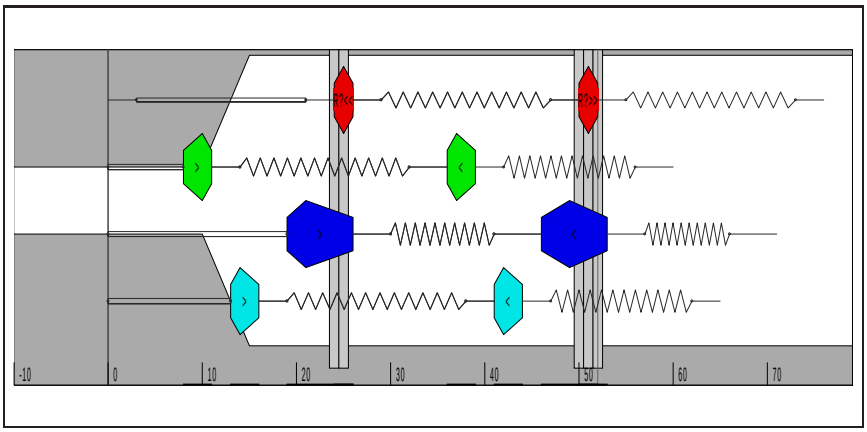

Abbildung 6.18: *Task-Set* `bsp-5.tsk`*, sehr kurzer Trichter ohne äußere Kraft (Rest-Systemenergie: Ungültiger Bereich)*

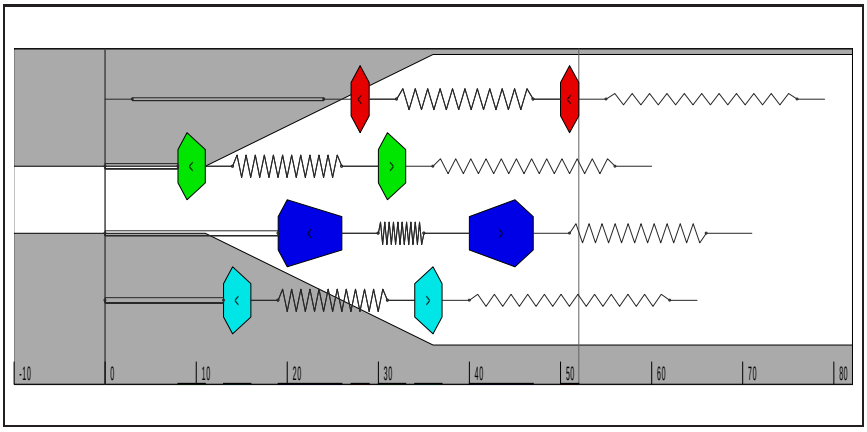

Abbildung 6.19: *Task-Set* `bsp-5.tsk` *modifiziert, durchschnittliche Trichterlänge ohne äußere Kraft (Rest-Systemenergie: 80)*

Wird der Trichter dagegen verlängert und somit auch die Auswirkung einer Schlittenüberlappung weiter hinten beachtet, ergeben sich sehr viel schneller gültige Schedules.

6.4.4 Post-Optimierung

Um die Post-Optimierungsmöglichkeiten zu zeigen, wird ausgehend vom in 6.1 spezifizierten Problem ein leicht verlängerter Schedule generiert.

Zentrierendes Federmodell

Die initiale Erzeugung erfolgt –wie üblich– mit einem zentrierenden Federmodell:

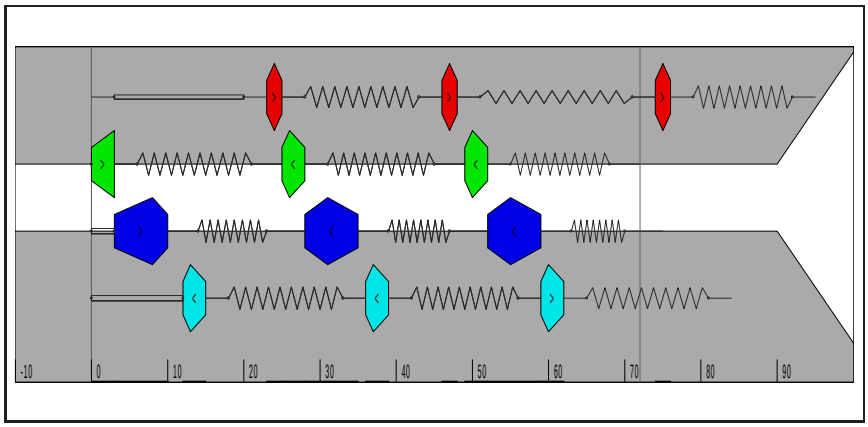

Abbildung 6.20: *Task-Set* `bsp-6.tsk`*, willkürlich verlängerter Zyklus (Rest-Systemenergie: 80)*

Kontrahierendes Federmodell

Bei einer Umstellung der Federcharakteristik auf kontrahierende Federn ergibt sich bei gleichbleibender Zyklenfestlegung eine sehr viel geringere zeitliche Zyklenlänge. Es ist klar zu erkennen, wie hier durch einige Release-Constraints diese Länge nach unten beschränkt wird. Aus einem Zyklus der Länge 72 ergibt sich nach einigen Iterationen ein Zyklus der Länge 52.

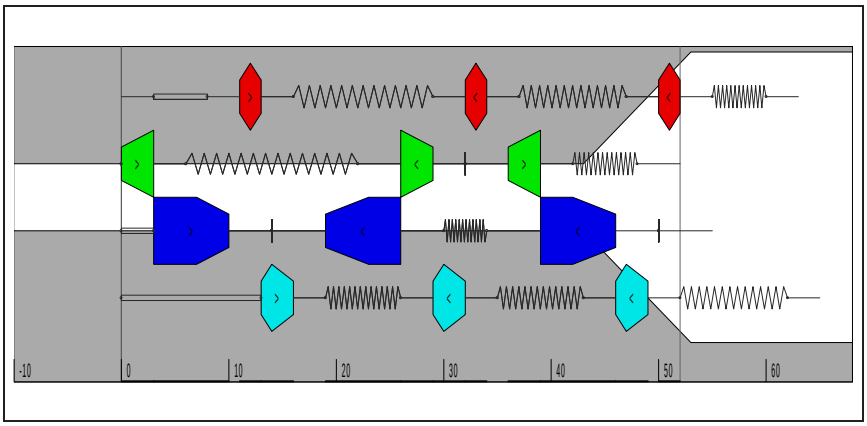

Abbildung 6.21: *Task-Set* `bsp-6.tsk`*, jeweils kontrahierende Federn (Rest-Systemenergie: 580)*

Expandierendes Federmodell

Analog kann durch expandierende Federcharakteristika eine zeitliche Streckung des Schedules erreicht werden. Die dabei erreichte Zyklenlänge beträgt 98, wobei die Anzahl der verplanten Job-Instanzen konstant gehalten wird:

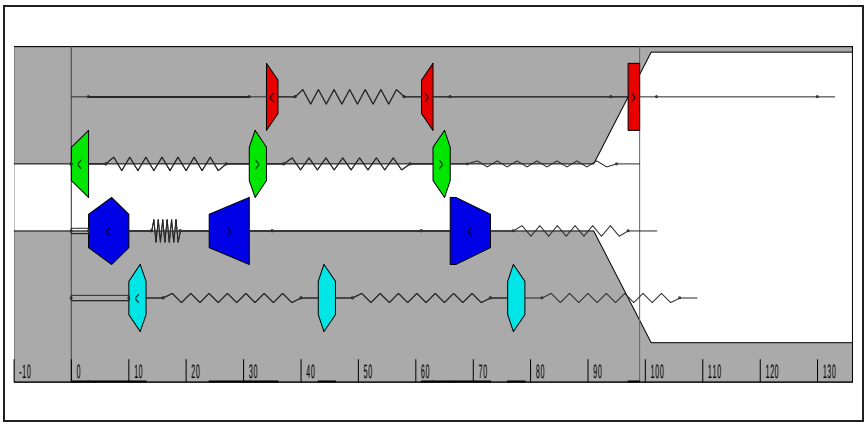

Abbildung 6.22: *Task-Set* `bsp-6.tsk`*, diesmal expandierende Federn (Rest-Systemenergie: 520)*

6.4.5 Flexibilität des Schedules

Das Hypersledge-Spring-Modell ist bezüglich der nachträglichen Änderung von Taskparametern relativ flexibel. Ausgehend von dem einfachen Scheduling-Problem in Abbildung 6.23 sollen einzelne Job-Parameter variiert werden und die jeweils entstehenden neuen Schedules dargestellt werden.

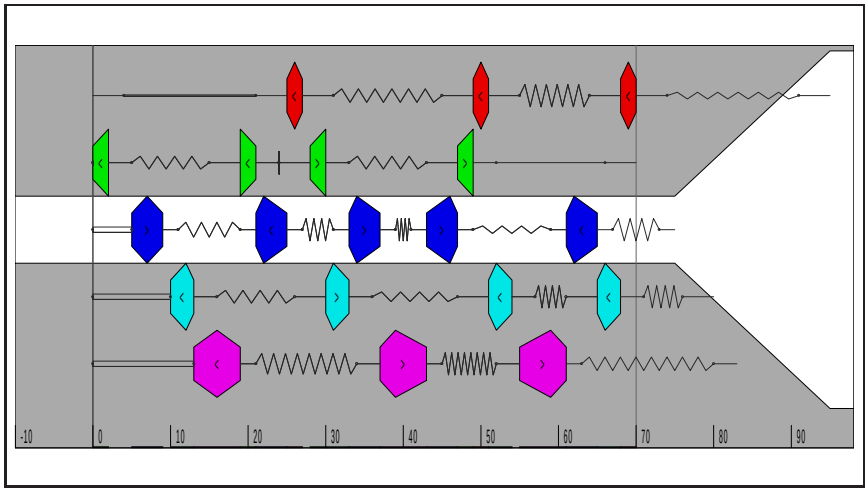

Abbildung 6.23: *Task-Set* `bsp-7.tsk`*, langer Zyklus (Rest-Systemenergie: 72)*

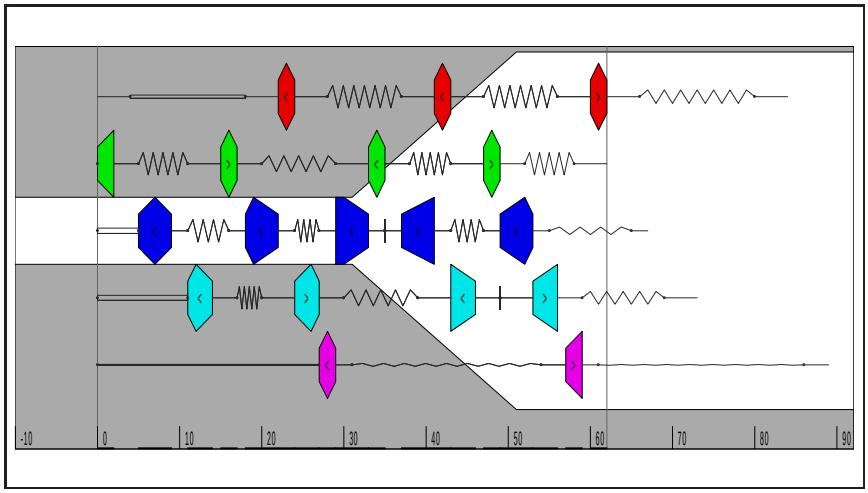

Abbildung 6.24: *Task-Set* `bsp-7.tsk`*,* $e_5 := 2$ *(Unterste Taskinstanz, war vorher* $e_5 = 6$*. (Rest-Systemenergie: 72)*

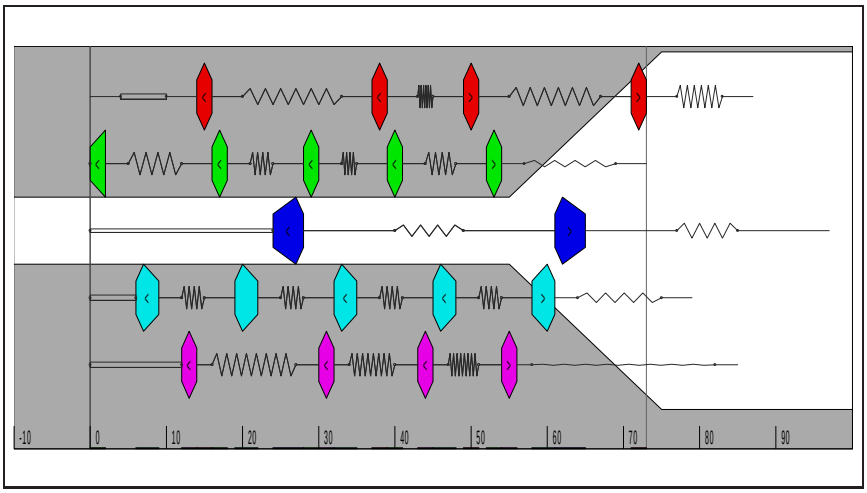

Abbildung 6.25: *Task-Set* `bsp-7.tsk`, $r_3 := 24$ *(Dritte Taskinstanz von oben, war vorher* $r_3 = 4$*, Rest-Systemenergie: 123)*

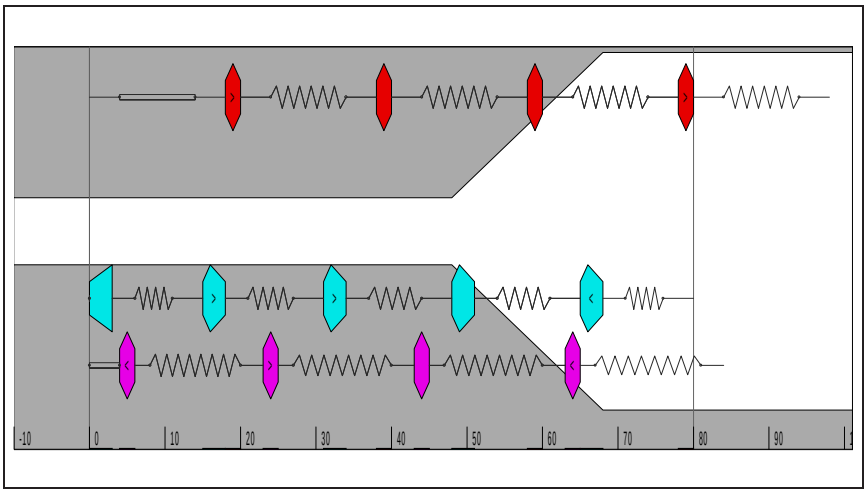

Abbildung 6.26: *Task-Set* `bsp-7.tsk`*, Entfernung der Jobs* J_2 *und* J_3 *(Rest-Systemenergie: 77)*

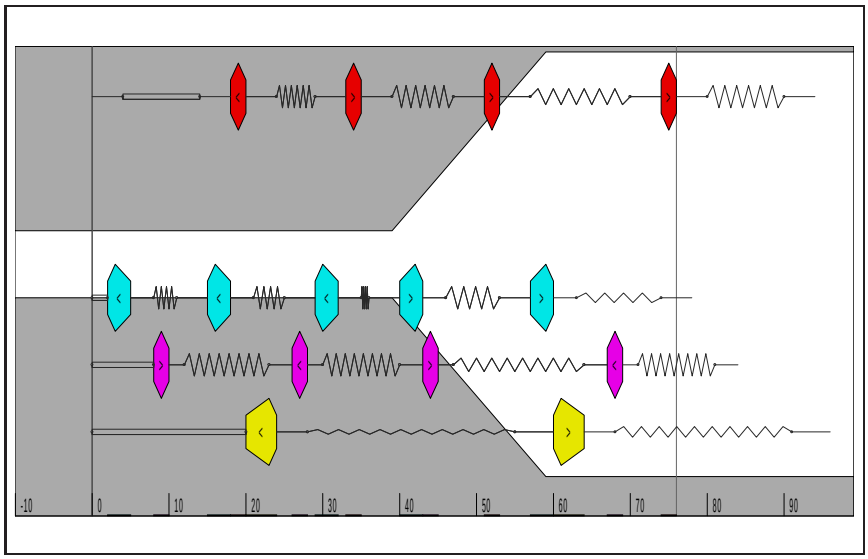

Abbildung 6.27: *Task-Set* `bsp-7.tsk`*, Hinzufügung eines neuen Jobs (Rest-Systemenergie: 77)*

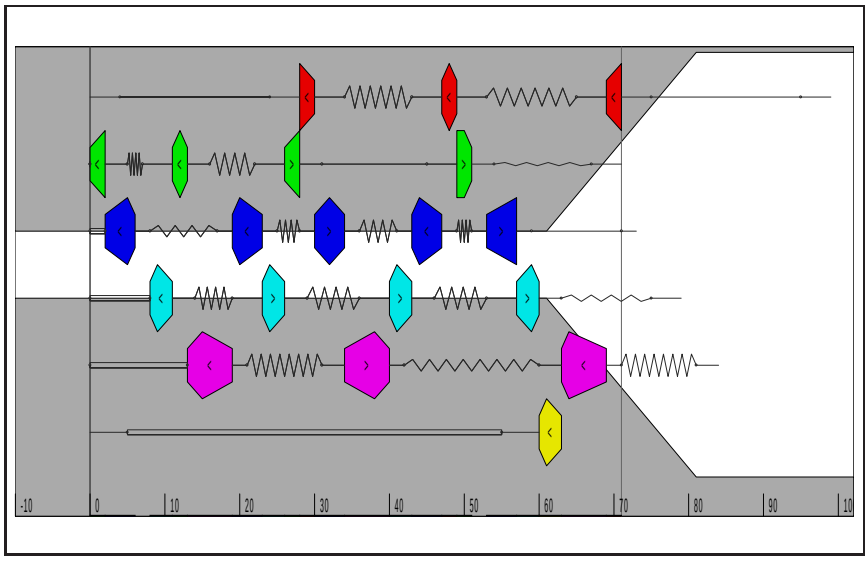

Abbildung 6.28: *Task-Set* `bsp-7.tsk`*, Alternatives Hinzufügung eines einmaligen Tasks im Zeitintervall* $(10, 90)$ *(Rest-Systemenergie: 77)*

Wichtig ist dabei, daß bei einer Änderung normalerweise nicht der gesamte Schedule neu generiert werden muß, sondern daß auf vorliegenden Resultaten aufgebaut werden kann. Zu erkennen ist dies an der Ähnlichkeit der einzelnen Abbildungen; jene Tasks, die von der Modifikation nicht betroffen wurden, bleiben im wesentlichen unberührt. Dennoch ist eine Variation der Positionen und auch eine Umordnung möglich.

6.4.6 Fixierung der Zykluslänge

Je nach Problemstellung kann es erforderlich sein, eine vorgegebene Zyklenlänge zu erreichen. Um "*das* Problem" zu lösen, ist in diesem Beispiel eine Zyklenlänge von 42 fest voreingestellt. Nach 40 Iterationsschritten liegt eine gültige Lösung (Abbildung 6.30) vor.

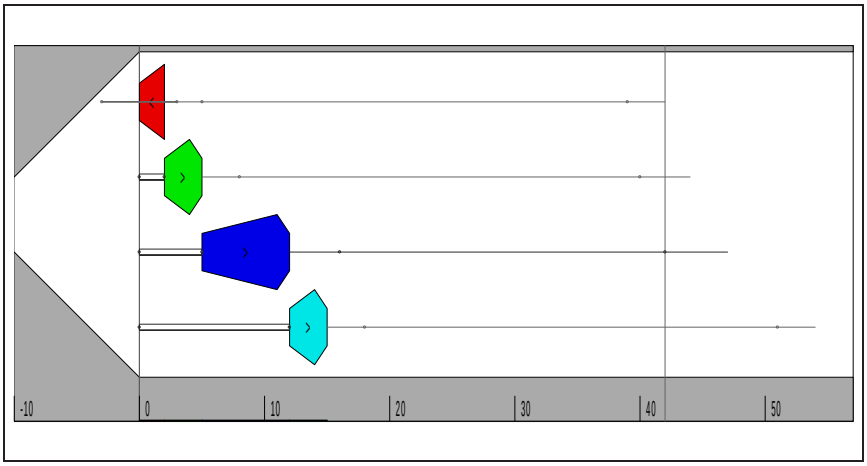

Abbildung 6.29: *Task-Set* `bsp-1.tsk`*, fixierte Zyklenlänge bei* $t = 42$ *(Rest-Systemenergie: Ungültiger Bereich)*

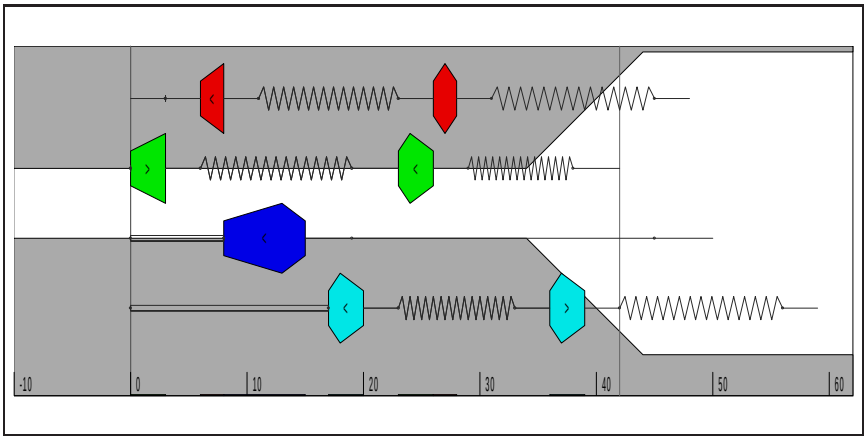

Abbildung 6.30: *Task-Set* `bsp-1.tsk`*, fixierte Zyklenlänge bei* $t = 42$ *(Rest-Systemenergie: 4)*

Kapitel 7

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit galt es, das Scheduling von unbeschränkten Taskfolgen mit relativen Timing-Constraints eingehender zu behandeln, verwandte Lösungsmethoden zusammenzutragen und zu bewerten sowie darauf aufbauend neue, spezialisierte Verfahren und Modelle zu entwickeln.

Nach einer eingehenden Problembeschreibung wurden einige Scheduling-Aufgaben für praxisrelevante Echtzeitsysteme vorgestellt. Das besondere Interesse an unendlichen Schedules sowie die Notwendigkeit von relativen Timing-Constraints konnte damit belegt werden.

Anschließend wurden drei in der Literatur vorgestellte Lösungsverfahren eingehender beschrieben:

- Static Cyclic Scheduling
- Parametric Dispatching
- Branch-and-Bound-Algorithmus mit Slack Time Vectors
-

Am Rande wurde auch kurz auf eine IDA^* Suchmethode eingegangen. Diese Verfahren weisen verschiedene Limitationen auf, dazu gehören etwa:

- Gültige Schedules werden unter Umständen nicht gefunden
- Keine Unterstützung von unendlichen Schedules
- Unflexible Ausführungspläne
- Hohe Komplexität der Planungsalgorithmen

Zur Beseitigung dieser Einschränkungen wurden zwei verschiedene Lösungsstrategien verfolgt. Als Grundlage dafür wurden zyklische Schedules gewählt, welche zur Repräsentation von unendlichen Taskfolgen geeignet sind. Es wurden zuerst einige theoretische Betrachtungen angestellt, welche Anforderungen an gültige Zyklen gestellt werden können und dürfen. Dabei konnte für 2-Job-Probleme ein notwendiges Kriterium für die Schedulability hergeleitet werden, welches eine Erkennungsquote von etwa 96% besitzt. Dieses läßt sich auch auf das inkrementelle Hinzufügen von Jobs zu bestehenden zyklischen Schedules anwenden.

Basierend auf den theoretischen Grundlagen wurde der Branch-and-Bound-Algorithmus aus [Eck99] um die Verwendbarkeit von Release-Constraints erweitert. Für das neue Verfahren konnten die Korrektheit und die Vollständigkeit gezeigt werden. Trotz

der Erweiterung auf Release-Zeiten konnte die Komplexität des Algorithmus größenordnungsmäßig konstant gehalten werden. Leider ist sie jedoch –wie bei vollständigen Suchverfahren üblich– relativ hoch (NP), so daß die Skalierbarkeit der Methode nicht gewährleistet ist. Große Task-Sets können nur bedingt mit annehmbarem Aufwand geplant werden.

Aus diesem Grunde wurde ein zweiter, iterativer Ansatz entwickelt. Grundlage dafür ist das Hypersledge-Spring-Modell, eine neuartige Repräsentation von Schedules. Taskinstanzen werden durch Schlitten repräsentiert, welche sich frei auf der Zeitachse bewegen können; relative und absolute Constraints werden durch Federn modelliert, Ressourcenbeschränkungen dagegen durch die räumliche Ausdehnung und Verdrängung der Schlitten. Der Zugang zu dem Modell ist sehr intuitiv und entstammt direkt der Physik, auch der Simulationsprozeß macht hier viele Anleihen. Es handelt sich um einen hybriden Ansatz, das *Directed Simulated Annealing*. Die dabei generierten Schedules können je nach Wunsch entweder endlich oder zyklisch aufgebaut sein und zeichnen sich durch eine sehr hohe Flexibilität aus:

- Hinzufügen von neuen Jobs
- Verwendbarkeit aperiodischer Jobs
- Veränderung von Job-Parametern zur Laufzeit, etwa der Job-Ausführungszeiten
- Kombinierbarkeit von mehreren Schedules

Ebenso gewährt die gute Visualisierbarkeit des Modelles eine intuitive Einsicht in limitierende Faktoren: Jobs, die etwa durch zu geringe zeitliche Spielräume den Scheduling-Prozeß behindern, können leicht identifiziert werden.

Die Leistungsfähigkeit sowohl der vorgestellten theoretischen Kriterien als auch des iterativen Modells wurde schließlich durch die Bearbeitung repräsentativer Scheduling-Probleme unter Beweis gestellt. Es zeigte sich, daß die Simulation flexibel auf geänderte Job-Parameter reagiert und nicht darauf angewiesen ist, bei einer kleinen Änderung den gesamten Schedule neu zu erzeugen. Ebenso lassen sich offline einfach neue Tasks hinzufügen oder entfernen, die Simulation erreicht schnell wieder gültige Schedules.

Unter allen möglichen Ausführungsplänen wird durch den geeigneten Aufbau des zugrundeliegenden Modelles eine Optimierung ermöglicht. Je nach Konzept lassen sich Zyklen minimaler oder maximaler Länge erzeugen. Dies gewährleistet etwa einen hohen Datendurchsatz bzw. maximale Freiräume für aperiodische Tasks. Zur Laufzeit lassen sich diese in den Schedule einbauen, wenn die Schlitten genügend Platz bieten. Die Überprüfung auf maximale Freiräume läßt sich durch einfache Untersuchungen erkennen, wobei auch die Verschiebung von Tasks im zulässigen Bereich zugelassen wird.

Insgesamt konnte gezeigt werden, daß es sich bei dem neuen Modell um eine flexible und intuitive Art und Weise der Schedule-Darstellung handelt. Der vorgestellte Simulator besitzt bereits eine gute Leistungsfähigkeit, kann jedoch weiter optimiert werden. Das Modell selbst ist anpassungsfähig genug, um etwa absolute Deadlines, intra-Task-Constraints oder eine Mehrprozessorunterstützung inkorporieren zu können. Es muß daher gehofft werden, daß in Zukunft neben den traditionellen, vollständigen Scheduling-Verfahren auch die iterativen Planungsmethoden weiter untersucht werden.

Anhang A

Anwenderdokumentation

A.1 Hypersledge-Spring-Simulator

Im Rahmen der Anfertigung dieser Arbeit ist ein visuelles Simulations-Tool auf Basis des in Abschnitt 5.2.2 definierten Hypersledge-Spring-Modells entstanden. Unter **Windows 95/98/NT** lassen sich zur Laufzeit Tasksets in weiten Grenzen spezifizieren und der Lösungsprozeß des Directed Simulated Annealing visuell überwachen. Dabei können diverse Simulationsparameter auch während der laufenden Simulation verändert, Probleme abgespeichert und erneut aufgerufen oder ausgedruckt werden.

Das Tool dient dabei vorwiegend dazu, die Funktionalität des zugrundeliegenden Modells zu testen und Aussagen über die Leistungsfähigkeit zu geben oder Parametervariationen zu vereinfachen. Es wurde dabei nicht so ausgelegt, daß zu möglichst vielen Task-Sets gültige Schedules gefunden werden bzw. daß Heuristika widersprüchliche Sets sofort zurückweisen. Daher können Fälle auftreten, in denen existierende gültige Schedules nicht oder nur nach sehr langen Simulationszeiten gefunden werden. Diese Möglichkeit ist jedoch nicht sehr wahrscheinlich, in den meisten Fällen –besonders bei insgesamt geringer Utilisation, siehe 3.4.2– wird schnell ein gültiger Plan erreicht.

Beim Start des Simulators durch **"hssim"** wird ein zuerst leerer Schedule präsentiert. In diesem Fenster kann später die Simulation verfolgt werden.

A.1.1 Die Menüstruktur

Über das Menü können diverse Aktionen ausgelöst und Konfigurationen vorgenommen werden:

- **File**
  - **Load Problem-Set** Von einem beliebigen Datenträger kann ein zuvor abgespeichertes Job-Set samt der zugehörigen Konfigurationsparameter geladen werden. Nach dem Einlesen wird ein initialer Schedule angezeigt, welcher nicht notwendigerweise gültig ist. Diese erste Approximation ergibt sich, wenn für jeden Job genau eine Taskinstanz verplant wird und diese genau zur Mitte ihres Starttime-Windows ausgeführt werden. Zur Durchführung des DSA ist solch eine Ausgangslösung zwangsweise erforderlich, sie muß jedoch keinesfalls zulässig sein.
  - **Save Problem-Set** Analog kann die aktuelle Konfiguration abgespeichert werden. Der gerade angezeigte Simulationsfortschritt gehört dabei nicht zum Problem sondern kann vielmehr als aktueller Lösungsversuch betrachtet werden. Da das qualitative Verhalten des DSA nicht von einer

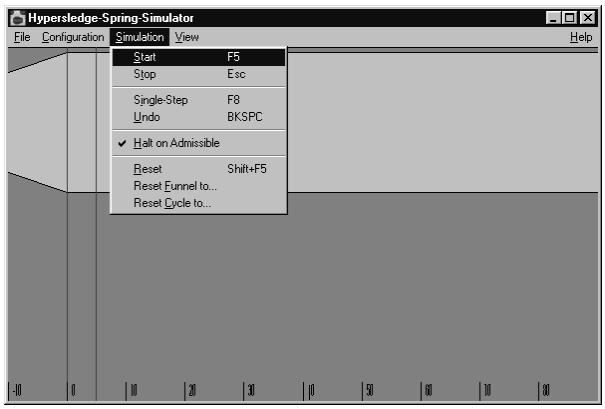

Abbildung A.1: *Das Startfenster des Hypersledge-Spring-Simulators*

spezifischen Ausgangs- oder Zwischenlösung abhängig sein sollte, kann auf eine Speicherung dieser Daten verzichtet werden.

- **Reset Problem-Set** Sämtliche Konfigurationseinstellungen sowie das gesamte Task-Set werden auf a-priori festgelegte Standardeinstellungen zurückgesetzt. Dies ist besonders zur Erstellung von neuen Beispielen sinnvoll, welche keinen "Ballast" der vorhergehenden Einstellung tragen sollen.
- **Print Schedule** Druckt den aktuell berechneten Schedule auf einem beliebigen System-Drucker aus. Die Darstellung erfolgt in etwa wie angezeigt, es wird jedoch der gesamte Schedule und nicht nur der gerade im Fenster angezeigte Teil ausgedruckt. Die Farben werden dabei druckergerecht angepaßt.
- **Exit** Beendet das Simulationsprogramm. Vor Beendigung sollte, falls gewünscht, das aktuelle Problem-Set zur späteren Weiterbearbeitung abgespeichert werden.

- **Configuration**
  - **Task-Set** Der Menüpunkt öffnet ein weiterführendes Konfigurationsfenster, welches unter A.1.2 weiter unten beschrieben wird.
  - **Simulation** Genauso kann der Simulationsvorgang in einem separaten Dialogfenster beeinflußt werden, siehe A.1.3.
  - **Dynamic Sledge Add/Remove** Dieser Schalter im Menü legt fest, ob Schlitten dynamisch zum Schedule hinzugefügt oder entfernt werden dürfen. Dies ist zum Beispiel dann zwangsweise erforderlich, wenn die Periodizität der einzelnen Jobs stark differiert. Wie unter 5.2.8 beschrieben wird anhand mehrerer Heuristiken festgestellt, ob ein Generieren oder Entfernen von Taskinstanzen den Schedule verbessern kann, also die Systemenergie absenkt.
    Ist diese Option nicht angewählt, wird ausgehend von den aktuell verplanten Schlitten eine weitere Simulation vorgenommen. Dies ist etwa dann sinnvoll, wenn mit verschiedenen Optimierungszielen experimentiert werden soll. Dabei kann ein gültiger Schedule z.B. auf kürzeste oder längste Zykluszeiten hin weiter optimiert werden, ohne durch das Ändern der

Schlittenzahl komplett verschiedene Schedules zu erzeugen.
Der aktuelle Zustand der Schalter wird dabei im Menü stets durch kleine Haken links vom Menüpunkt angezeigt. Ein vorhandener Haken bedeutet dabei, daß die Option momentan aktiv ist.

- **Cyclic Schedules** Diese Option legt fest, ob mit Zyklen gearbeitet werden soll. Für das Scheduling von nicht-unendlichen Taskfolgen oder zur Bewertung des einfachen DSA ist es oft sinnvoll, auf die Zyklen-Bildung zu verzichten. In diesem Falle werden keine Feedback-Federn von der letzten zur ersten Taskinstanz eines jeden Jobs eingefügt. Eine entsprechende Konfiguration vorausgesetzt kann aber auch so der Trichter kontinuierlich über die Job-Sets bewegt werden. Bei entsprechend schneller Abarbeitung der Simulation ist auch so das dynamische Erstellen von unendlichen Schedules möglich.
  Durch den Verzicht auf die Feedback-Federn vereinfacht sich das Scheduling-Problem stark. Es läßt sich noch besser durch das DSA lösen, da einem strikt orientierten Problem ein strikt orientiertes Verfahren gegenübersteht. Erst durch die Feedback-Federn können Änderungen am Ende des Zyklus Rückwirkungen auf den Anfang haben und somit eine globale Variation des Schedules erzwingen.
- **Dynamic Cycle Length** Normalerweise wird während des Simulationsprozesses die Zyklus-Länge als eigener Parameter in die Optimierung mit einbezogen, siehe 5.2.9. Wenn jedoch durch äußere Constraints oder durch vorhergegangene Simulation bereits eine feste Zyklus-Länge vorgegeben ist, läßt sich die automatische Zyklenbestimmung deaktivieren. In diesem Falle wird der gerade aktuelle Zyklus beibehalten und nur eine Variation der Task-Instanzen innerhalb dieses Zyklus vorgenommen.
  Auch diese Option ist zum Bewerten der einzelnen Optimierungsparameter geeignet, da sich Nebeneffekte ausblenden lassen. Die Kräftekopplung und die Auswirkung auf die Bewegung der Schlitten läßt sich bei fester Zyklenlänge sehr viel leichter beobachten und deuten.
- **Move Funnel**[1] Option, welche festlegt, ob beim DSA eine Abkühlung der Probe erfolgen soll. Durch Deselektion kann der Trichter an der aktuell eingenommenen Position fixiert werden. Besondere Relevanz besitzt diese Option, um mit verschiedenen Vertikalkräften zu experimentieren und zu sehen, inwieweit sich Feder- und Interferenzkräfte überlagern und welche Reaktionen erfolgen. Sinnvoll mag es in diesem Falle sein, den Trichter so zu positionieren, daß der gesamte Zyklus innerhalb oder außerhalb des bearbeiteten Bereiches liegt, siehe dazu **Reset Funnel Position**.

- **Simulation**

  - **Start** Startet die Simulation mit den aktuellen Spezifikationen bzw. setzt die Simulation fort. Nach jedem Schritt wird der angezeigte Schedule aktualisiert und auf Zulässigkeit geprüft. Wenn der Benutzer es wünscht, wird bei Erzeugen eines zulässigen Plans die Simulation abgebrochen und eine Meldung angezeigt. Durch erneuten Start kann sie jedoch beliebig fortgesetzt werden.
  - **Stop** Die Simulation wird nach Beendigung des aktuellen Schrittes angehalten. In dem pausierten Zustand ist es besonders einfach, Zusatzinformationen über den aktuellen Schedule zu erlangen (siehe A.1.4) oder einen Ausdruck anzufertigen.

[1] **funnel** ['f∧nl] *n* Trichter *m*

- **Single-Step** Im Zustand der angehaltenen Simulation kann durch Wahl dieses Menüpunktes gezielt ein Schritt ausgeführt werden. Dabei läßt sich gut beobachten, wie pro Schritt nur ein Schlitten gewählt und bewegt wird und wie sich die Schlittenkräfte dabei verändern.
- **Undo** Macht den letzten Simulationsschritt, sofern möglich, rückgängig. Besonders komplexe Operationen wie z.B. die Entfernung von Schlitten aus dem Schedule, welche jedoch selten ausgeführt werden, können dabei nicht rückgängig gemacht werden.
- **Halt on Admissible**[2] Schalter legt fest, ob die Simulation bei Erreichen einer zulässigen Lösung mit einer entsprechenden Hinweismeldung angehalten werden soll. Besonders zur initialen Generierung von gültigen Schedules ist diese Option hilfreich, bei der späteren Optimierung (etwa mit geänderten Federtypen) sollte sie dagegen abgewählt werden.
- **Reset** Setzt den Schedule auf den initialen Minimal-Plan zurück. Dabei ist pro Job genau eine Taskinstanz eingeplant. Diese ist zeitlich so arrangiert, daß sie eine mittlere Periodizität aufweist, also zur Mitte des Start Time Windows gestartet wird.
- **Reset Funnel to** Öffnet eine weitere Dialogbox, in welcher die aktuelle Position des Trichters geändert werden kann. Beispielsweise bei der Post-Optimierung mit verschiedenen Federtypen ist es sinnvoll, manuell in die Positionierung des Trichters einzugreifen. Je nach Status der Option **Move Funnel** wird der Trichter anschließend weiterbewegt oder fixiert.
- **Reset Cycle to** Analog kann der aktive Zyklus geändert werden. Vorausgesetzt, daß die Zyklenlänge nicht automatisch bestimmt wird (Option **Dynamic Cycle Length**), kann das Ende des Zyklus beliebig festgesetzt werden. Ist zum Auswahlzeitpunkt die Option **Dynamic Cycle Length** aktiv, wird diese auf Wunsch automatisch deselektiert.
  Mit Hilfe dieses Menüpunktes läßt sich etwa einer durch Umweltbedingungen geforderten fixen Zykluslänge Rechnung tragen. Außerdem können die ausgeführten Optimierungsschritte bei fester Zyklendefinition leichter nachvollzogen werden.

- **View**
  - **Visible Simulation** Diese Option legt fest, ob nach jedem Simulationsschritt der aktive Schedule angezeigt werden soll. Normalerweise ist es nicht sinnvoll, auf diese Ausgabe zu verzichten, langsamere Computer mit betagteren Grafikkarten können jedoch teilweise durch den häufigen Bildaufbau ausgebremst werden. Im unsichtbaren Simulationsmodus wird der Fortschritt durch einen einfachen Balken angezeigt. Bei Unterbrechung der Simulation wird dann der aktive Schedule angezeigt.
  - **Scroll Window** Dient zur Verschiebung des sichtbaren Schedule-Ausschnitts. Bei längeren Schedules oder längeren Zyklen kann nur ein Teil des Planes im Fenster angezeigt werden. Um den Ausschnitt zu verschieben, wird ein horizontaler Scroll-Balken eingeblendet. Bei Verschiebung des Reglers wird der Bildinhalt automatisch angepaßt. Im normalen Simulationsmodus dagegen wird der Ausschnitt automatisch so verschoben, daß der gerade aktive Teil des Schedules sichtbar wird.
  - **View Thesis** Zeigt dieses Dokument online mit Hilfe des **Acrobat Readers**[3] an.

[2] **admission** *n* Einlaß *m*, Zulassung *f*
[3] Freeware, siehe http://www.adobe.com

- **Help** Dient zur Anzeige einiger versionsrelevanter Informationen zum Programm und zum Aufruf dieses Dokumentes in der Online-Version.

A.1.2 Konfiguration des Task-Sets

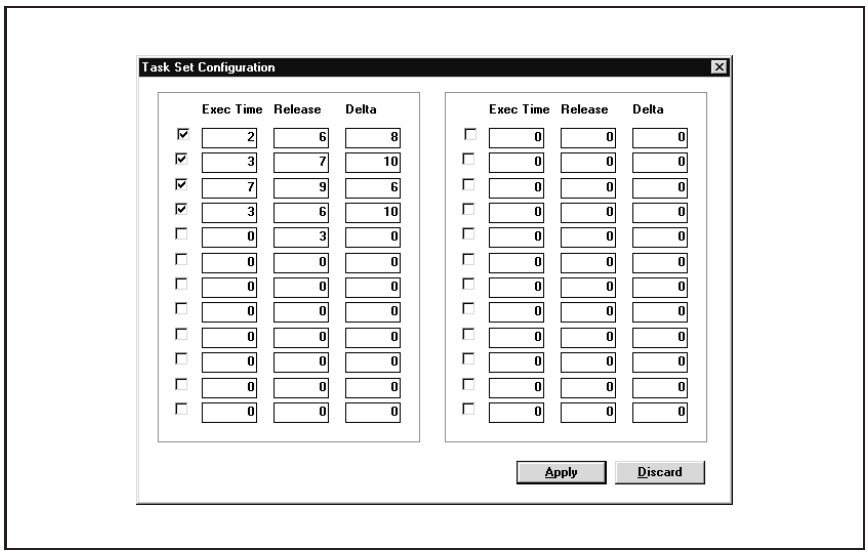

Abbildung A.2: *Konfiguration des Task-Sets im Hypersledge-Spring-Simulator*

Bis zu 24 Jobs können mit Hilfe des in Abbildung A.2 gezeigten Dialoges konfiguriert werden. Für jeden Job wird dabei angegeben, ob er aktiv ist (Haken im linken Aktivierungsfeld) und welche Zeitconstraints er besitzt. Dazu gehören die Ausführungszeit (Execution Time, links), die Release-Zeit (mittig) und die Länge des Start Time Windows (rechts, zur Definition siehe auch 1.4).

In den Planungsprozeß einbezogen werden dabei nur die aktivierten Tasks. Dies kann dazu verwendet werden, einzelne Jobs kurzzeitig aus der Optimierung herauszunehmen und dadurch die geforderte Prozessorauslastung zu verringern. Bei der Speicherung oder dem Laden von Problem-Sets werden sämtliche Jobs, also auch die nichtaktiven, berücksichtigt.

Die Umsetzung der Jobs in den Schedule-Graph erfolgt in der Definitionsreihenfolge. Der erste definierte Job spezifiziert also die oberste Schlittenreihe im Graphen, sofern der Job aktiv ist. Ansonsten wird der erste aktivierte Job in der ersten Reihe angezeigt.

Wenn der Nutzer die Set-Spezifikationen ändert, während schon ein Schedule in Bearbeitung ist, wird dieser so wenig wie möglich verändert. Dies heißt, daß sämtliche unveränderten Jobs im Schedule verbleiben. Nur jene Jobs, deren Parametrisierung geändert wurde bzw. die neu in das Set aufgenommen wurden, werden im Schedule entfernt und anschließend neu erzeugt. Dadurch ist beispielsweise zu zeigen, daß mit Hilfe des Hypersledge-Spring-Modells formalisierte Schedules unkritisch gegenüber der Hinzufügung neuer Jobs sind.

Weiterhin ist es möglich, einmalige Tasks zu spezifizieren. In diesem Falle ist einfach die Releasezeit oder die Deadline mit negativem Vorzeichen anzugeben. Der Task mit der spezifizierten Ausführungszeit e_i wird dann innerhalb des durch $[|r_i|, \ldots, |r_i|+|d_i|]$ definierten Intervalls gestartet. Dadurch sind auch (in diesem Fall jedoch absolute) Zeitschranken an derartige Instanzen möglich.

A.1.3 Konfiguration der Simulation

Zur Konfiguration einiger Simulationsparameter dient der in Abbildung A.3 dargestellte Dialog.

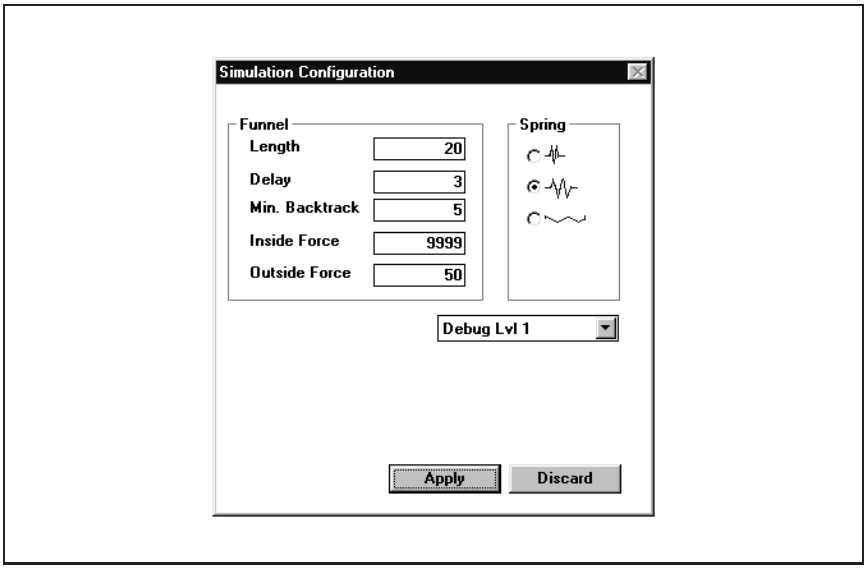

Abbildung A.3: *Die Konfiguration der wichtigsten Simulationsparameter erfolgt in einer eigenen Dialogbox*

Im linken Teil gilt es, den Trichter zu spezifizieren: Mit der Länge (**Length**) ist der zeitliche Abstand zwischen dem engsten und dem weitesten Stück des Trichters bezeichnet. Je größer die Länge des Trichters gewählt wird, desto breiter ist die Abkühlungsfront des DSA und desto früher wird begonnen, zeitliche Interferenzen zu berücksichtigen. Für die Wahl dieses Parameters gibt es keine Empfehlung, je nach Aufgabenstellung und Problemstruktur eignen sich kurze und manchmal auch entartete Trichter (Länge 0). Im allgemeinen sind jedoch Werte in der Größenordnung der durchschnittlichen Jobperiodizität sinnvoll, da dann für jeden Job normalerweise zwei Taskinstanzen im Trichtergebiet liegen.

Hinter der **Delay**-Zeit verbirgt sich die relative Geschwindigkeit des Trichters in Bezug auf die Simulationsgeschwindigkeit. Bei einem Wert von 3 wird der Trichter nach jeweils drei Simulationsschritten bewegt. Dies dient dazu, pro Trichterposition einen energetisch relativ günstigen Schedule zu erreichen. Auch hier gibt es keine generellen Empfehlungen an die Parameterwahl, der Wert sollte weder zu groß noch zu klein sein. Praktikabel ist es, die Anzahl der verwendeten Jobs zu verwenden. So kann pro Task im Durchschnitt eine Taskinstanz verschoben werden.

Wie schon unter 5.2.6 dargestellt ist, wird bei einer auftretenden Verklemmung von Taskschlitten der Trichter wieder leicht zurückgefahren. Dies gibt den Schlitten wieder eine größere Bewegungsfreiheit, so daß das Problem beseitigt werden kann. Die Größe **Min. Backtrack** gibt an, um wieviele Zeiteinheiten der Trichter bei einer Verklemmung mindestens zurückgeschoben werden muß. Die Angabe sollte groß genug sein, um wieder Variationen im lokalen Umfeld der Problemschlitten zu ermöglichen, aber nicht so groß, daß der Schedule global verändert wird. Idealerweise stellt man hier die durchschnittliche Periodizität der Jobs oder die durchschnittliche Execution Time ein.

Die Kräfteangaben **Inside Force** und **Outside Force** spezifizieren die Vertikalkräfte am linken und rechten Ende des Trichters (siehe Abbildung 5.11). Die Kräfteverteilung im Inneren des Trichters wird linear interpoliert. Sinnvoll ist es hier, im Inneren des Trichters höhere Kräfte auftreten zu lassen als im Äußeren, da nur dies dem Prinzip des DSA entspricht. Die absolute Größe der Kräfte ist im allgemeinen jedoch unkritisch.

Im rechten Teil der Dialogbox läßt sich das Verhalten der Federn einstellen (vergleiche Abbildung 5.10). Diese können ein kontrahierendes, zentrierendes oder expandierendes Verhalten aufweisen. Zur Erlangung von gültigen Ausgangsschedules hat sich die Wahl der zentrierenden Federn als sinnvoll herausgestellt. Für spätere Optimierungen (kurze oder lange Zyklen) muß das Federverhalten hier variiert werden.

Zur internen Nutzung ist eine Einstellmöglichkeit für das **Debug Level** vorgesehen. Diese Wahl beeinflußt, in welchem Maße Hinweistexte in den Statusbericht ausgegeben werden. Im allgemeinen sollten hier keine Einstellungen vorgenommen werden, um den Rechner nicht mit der Ausgabe von Debug-Meldungen zusätzlich zu belasten.

A.1.4 Der angezeigte Schedule

Während der Simulation werden laufend die aktuellen Schedules graphisch dargestellt. Die im Fenster unter A.4 dargestellten Elemente teilen sich in folgende Gruppen auf:

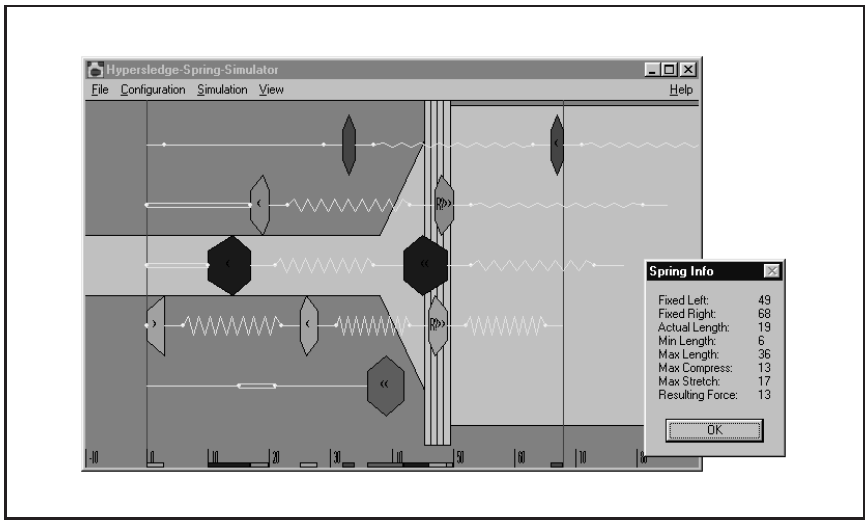

Abbildung A.4: *Anzeige des aktiven Schedules. Taskschlitten sind mit der aktiven Form dargestellt, Federn entsprechend der Spezifikationen. Zusätzlich eine Info-Box mit Informationen zur äußersten Feder unten rechts.*

- **Taskschlitten** Zu den zentralen Elementen gehören die Taskschlitten. Dies sind die normalerweise achteckigen Gebilde in verschiedenen Farben. Horizontal auf gleicher Ebene angeordnete (gleichfarbige) Schlitten gehören zu demselben Job. Die Form der Schlitten richtet sich nach der aktuellen Auslenkung der befestigten Federn, siehe 5.2.4. Im Innern der Schlitten ist graphisch die aktuell wirkende Kraft eingezeichnet. Je nach Stärke der Kraft wird sie durch die Symbole “<<”, “<”, “>” oder “>>” dargestellt. Bei harten Interferenzen wird ein “R” vorangestellt; dies bedeutet, daß die Richtung der Kraft nicht bekannt ist sondern zufällig ermittelt werden muß.

- **Federn** Zwischen den Taskschlitten sind die Federn angeordnet. Sie teilen sich auf in den unflexiblen Bereich (horizontale Linie) und den flexiblen Bereich (Zickzack-Linie). Der unflexible Bereich stellt die Release-Zeit dar, d.h. die minimale Länge der Feder. Aus der Länge der Zickzack-Linie ergibt sich näherungsweise die maximale Auslenkung und damit die Länge des Start Time Windows. Je weiter die Feder gedehnt wird, desto weiter wird die Feder gespreizt dargestellt. Bei Erreichen der maximalen Dehnbarkeit wird insbesondere aus der Zickzack- eine Horizontallinie. Visuell ist somit der zulässige Bereich jeder einzelnen Feder gut zu erkennen.
  Werden Federn außerhalb ihres zulässigen Bereiches eingesetzt, liegt also eine Constraintverletzung vor, so wird die Feder rot hervorgehoben. Bei der Kraftberechnung innerhalb des Simulationsprozesses muß dann eine unzulässige Kraft errechnet werden.
  Eine spezielle Form von Federn stellen die sogenannten **Begrenzer** dar. Diese besitzen ähnlich wie die Federn minimale und maximale Auslenkung, üben aber innerhalb des zulässigen Bereiches keine Kraft aus. Dargestellt werden sie ähnlich den Federn durch zylindrische Elemente, welche den flexiblen Teil der Feder ersetzen. Solche Begrenzer finden etwa zur Verankerung von sporadischen oder einmaligen Tasks in das ihnen zugeteilte Startfenster oder zur Fixierung der jeweils ersten Taskinstanzen Verwendung.

- **Interferenzen** An den Stellen, wo sich mindestens zwei Hyperschlitten zeitlich überlappen, liegt eine Interferenz vor. Diese kann entweder hart oder weich sein (siehe 5.2.4). Eine harte Interferenz kann nicht allein durch Verschiebung der beteiligten Schlitten beseitigt werden und erfordert somit stärkere Veränderungen im Plan. In der Darstellung werden die Interferenzen durch farbige, vertikale Balken dargestellt, welche die Überlappungszeiten angeben. Harte Interferenzen werden dabei durch rote, weiche durch gelbe Balken angezeigt.

- **Zyklus-Länge** Der aktuelle Zyklus ist links und rechts durch zwei dünne, vertikale Linien angezeigt. An diesen Stellen erfolgt das Zusammenfügen von linkem und rechtem Teil des Planes. Die Feedback-Federn müssen somit über das rechte Zyklusende hinausragen und von diesem denselben Abstand haben, wie dies die ersten Taskinstanzen vom Zyklenstart definieren. In obiger Abbildung ist dies einfach ersichtlich.

- **Schlitten-Informationen** Durch einen Doppelklick auf einen beliebigen angezeigten Schlitten lassen sich nähere Informationen zu dieser Taskinstanz erhalten. Neben der absoluten Position lassen sich auch die wirkenden Einzelkräfte ablesen. Diese Angaben sind jedoch vorwiegend zur Bewertung und Optimierung des zugrundeliegenden Modelles konzipiert und sollten normalerweise nicht relevant sein.

- **Federn** Analog lassen sich durch Doppelklick auf eine Feder nähere Informationen einblenden (siehe dazu auch obiges Fenster). Es werden die Befestigungspunkte, die minimale und maximale Auslenkung sowie die resultierende, gerichtete Kraft ausgegeben. Wiederum dienen diese Angaben vorwiegend zur Entwicklung und Bewertung des Modelles.

- **Zeitbalken** Am unteren Ende des Fensters ist zur Orientierung eine Zeitskala eingeblendet. Nicht nur die absoluten Zeitangaben werden hier vermerkt, auch wird farbig eingezeichnet, zu welchen Zeiten welcher Task eingeplant ist. Aus dieser Darstellung lassen sich einfach Idle-Zeiten des Schedules und die Utilisation des Systems erkennen.

A.2 Branch-And-Bound

Zur Überprüfung des unter 4.2 beschriebenen Prinzips des "Relative Starttime Vector" im Branch-and-Bound-Verfahren sowie zur Erlangung der in 6.3 vorgestellten experimentellen Ergebnisse wurden einige Hilfsprogramme erstellt. Diese sind jedoch vorrangig für den erfahrenen *C++*-Programmierer konzipiert, es fehlen Benutzerschnittstellen und intuitive Ein- sowie Ausgaberoutinen. Sämtliche Konfiguration ist im Source-Code selbst vorzunehmen.

Dennoch erlauben die vorhandenen Module das zyklische Scheduling von beliebigen Task Sets mittels des Branch-and-Bound-Ansatzes auf vielen Unix-Systemen. Im folgenden sollen nur kurz die einzelnen Dateien vorgestellt werden, der versierte Programmierer kann basierend auf den Source-Files sicherlich schnell eigene Anwendungen erstellen.

- **Makefile**
  Unix-kompatibles Makefile, erzeugt die benötigen ausführbaren Programme. Angepaßt an Solaris 5.
- **sched.c**
  Enthält den eigentlichen Branch-and-Bound-Algorithmus. Das Job-Set wird mittels einer *struct* definiert. Algorithmus vergibt explizite Idle-Zeiten im Schedule. Ausgabe der zugehörigen zyklischen Sequenz, wenn gefunden. Zu Testzwecken maximale Länge des Zyklus beschränkbar.
- **sched.h**
  Spezifikation der *struct* und der Ein- und Ausgabevariablen.
- **sched_test.c**
  Einfaches Testprogramm, führt Branch-and-Bound mit voreingestelltem Test-Set durch. Anpassung der Parameter im Source möglich.
- **2tasks.c**
  Für zwei Tasks werden die drei spezifizierenden Parameter innerhalb vorgegebener Intervalle variiert. Prüfung von vorgegebenen Heuristiken oder Kriterien und Gegenüberstellung mit tatsächlicher Schedulability, welche durch einen Branch-and-Bound-Durchlauf ermittelt wird. Liefert Tabellen nach Art von 6.3.

Anhang B

Klassifikation der Methoden

Die in verschiedenen Kapiteln verwendete Klassifikation der einzelnen Verfahren wird in Abbildung B.1 vergleichend zusammengefaßt. Details zu den angegebenen Methoden finden sich in den entsprechenden Abschnitten dieser Arbeit oder bei den eigentlichen Urhebern.

		Static Cyclic Scheduling ([CA94])	Parametric Dispatching ([SGA93])	Slack Time Vector ([Eck99])	Relative Starttime Vector (4.2)	Hyperslede-Spring-Model (5.2.2)
	Seite	13	16	18	38	49
Funktionsprinzip	Offline	●	●	●	●	●
	Online		●			○
Echtzeitverhalten	Hard RTS	●	●	●	●	●
	Soft RTS					
Ausführungsumgebung	Single Proc.	●	●	●	●	●
	Multi Proc.					○
Tasktypen	Periodisch	●	●	●	●	●
	Aperiodisch			○		●
Ausführungszeiten	Exakt	●		●	●	●
	Beschränkt		●			
Release-Zeiten	Absolut		●		○	○
	Relativ	●	●		●	●
Deadlines	Absolut		●		○	○
	Relativ	●	●	●	●	●
Unterstützung Aperiodischer Tasks	Offline			○	○	○
	Online		●			●

Abbildung B.1: *Klassifikation sämtlicher im Detail betrachteter Methoden*

Anhang C

Daten-CD-Rom

Beiliegende CD-Rom enthält unter anderem sämtliche Dokumente und Programme, die während der Erstellung dieser Arbeit angefertigt wurden. Dazu gehören insbesondere:

- Dieses Dokument im Postscript sowie im direkt einseh- und durchsuchbaren Adobe-PDF-Format[1]
- Hypersledge-Spring-Simulator
- Branch-and-Bound-Algorithmus
- Sämtliche Source-Dateien für Programme und Dokumente
- Adobe Acrobat Runtime-Version

Falls die Daten-CD in dieser Ausgabe fehlen sollte, verwenden Sie bitte die unter

`http://wilko.hein.net`

verfügbaren Daten und Programme.

[1] Siehe: http://www.adobe.com. Konvertierung durch [Bab]

Danksagung

An dieser Stelle möchte ich Herrn Prof. Klaus Ecker danken, der sowohl das Thema dieser Arbeit gestellt als auch zu dem Entstehen in vielen fachlichen und interessanten Gesprächen beigetragen hat. Ebensolcher Dank gilt Herrn Prof. Ingbert Kupka, der mir –besonders im theoretischen Teil– hilfreich und kritisch zur Seite gestanden hat. Beide waren prompt zu kurzen Gesprächen bereit, die diese Arbeit sehr vorangetrieben haben.

Danken möchte ich auch meiner Familie, den Freunden und besonders Francesca für moralische Unterstützung, aufmunternde Worte und dafür, daß sie stets für mich da waren. Danke!

Abbildungsverzeichnis

Literaturverzeichnis

[BESW93] Blazewicz, J.; Ecker, K. H.; Schmidt, G.; Weglarz, J., *"Scheduling in Computer and Manufacturing Systems"*, Berlin, 1993

[Bab] Babinszki, A., *"Net Distillery"*, Babinszki World, `http://www.babinsz` `↪ki.com/distiller/`

[Cho97] Choi, S. (Directed by Agrawala, A. K.), *"Dynamic Time-Based Scheduling for Hard Real-Time Systems"*, Dissertation, UMD CS-TR-3850, University of Maryland, 1997

[CA94] Cheng, S.-T.; Agrawala, A. K., *"Scheduling of Periodic Tasks with Relative Timing Constraints"*, UMD CS-TR-3392, University of Maryland, 1994

[CA95] Cheng, S.-T.; Agrawala, A. K., *"Scheduling of Periodic Tasks with Relative Timing Constraints"*, UMD CS-TR-3402, University of Maryland, 1995

[CA97] Choi, S.; Agrawala, A. K., *"Scheduling Aperiodic and Sporadic Tasks in Hard Real-Time Systems"*, UMD CS-TR-3794, University of Maryland, 1997

[CA97b] Choi, S.; Agrawala, A. K., *"Dynamic Dispatching of Cyclic Real-Time Tasks with Relative Constraints"*, UMIACS-TR-97-30, University of Maryland, 1997

[CDHC94] Carpenter, T.; Driscoll, K.; Hoyme, K.; Carciofifi, J., *"Arinc 659 scheduling: Problem Definition"*, In: Proceedings of IEEE Real-Time Systems Symposium, San Juan, 1994

[D94] n.n., *"DUDEN – Die deutsche Rechtschreibung"*, Dudenverlag Mannheim, 1994

[DD95] Domschke, W.; Drexel, A., *"Einführung in Operations Research"*, Springer-Verlag, Berlin, 1995

[Eck92] Ecker, K., *"Algorithmic Methods for Real-Time Scheduling"*, Informatik-Berichte des Instituts für Informatik der TU Clausthal, 1992

[Eck99] Ecker, K. H., *"Repeated Execution of Non-Preemptive Tasks with Relative Timing Constraints"*, Submitted to Real-Time-Systems, 1999

[FAQNLP] n.n., *"sci.opt-research Nonlinear Programming 'FAQ' (Frequently Asked Questions)"*, Online-Dokument, URL `http://www.isa.utl.pt/` `↪matemati/mestrado/io/nlp.html` o.ä.

[FAQRT] n.n., *"Comp.RealTime Newsgroup 'FAQ' (Frequently Asked Questions)"*, Online-Dokument, URL `http://www.realtime-info.be/encyc/` `↪techno/publi/faq/rtfaq.htm` o.ä.

[FK92] Fohler, G.; Koza, C., *"Heuristic Scheduling for Distributed Hard Real-Time Systems"*, Institut für Technische Informatik Universität Wien (Hg.), 1992

[GH94] Gerber, R.; Hong, S., *"Compiling Real-Time Programs with Timing Constraint Refinement and Structural Code Motion"*, UMD CS-TR-3323, UMIACS-TR-94-90, University of Maryland, 1994

[GKHS95] Gerber, R.; Kang, D.; Hong, S.; Saksena, M., *"End-to-End Design of Real-Time Systems"*, UMIACS-TR-95-61, University of Maryland, 1995

[GPS95] Gerber, R.; Pugh, W.; Saksena, M., *"Parametric Dispatching of Hard Realtime Tasks"*, IEEE Transactions on Computers, 1995

[IBM98] IBM Corporation, *"Strategies for Solving Mathematical Programming Problems"*, Online-Dokument, URL `http://www.research.ibm.com/` `↪osl/ekkgc6.html`

[K89] Kucera, A., *"The Compact Dictionary of Exacr Science and Technology"*, Oscar Brandstetter Verlag, Wiesbaden, 1989

[KGS98] Kang, D.-I.; Gerber, R.; Saksena, M., *"Parametric Design Synthesis of Distrubuted Embedded Systems"*, UMD CS-TR-3885, UMIACS-TR-98-18, University of Maryland, 1998

[Kop95] Helmut Kopka, *"LaTeX Einführung"*, 2. Auflage, Addison Wesley, 1995

[P90] n.n., *"Pons Standardwörterbuch Deutsch-Englisch, Englisch-Deutsch"*, Klett-Verlag Stuttgart, 1990

[PS91] Puschner, P. P.; Schedl, A. v., *"Computing Maximum Task Execution Times – A Graph-Based Approach"*, Journal of Real-Time Systems, 13(1):67-91, 1997

[RN95] Russel, S.; Norvig, P., *"Artificial Intelligence – A Modern Approach"*, Prentice Hall, New Jersey, 1995

[Sch92] Schmidt, L., *"Leitfaden zur Vorlesung 'Technische Mechanik"'*, TU Clausthal, Clausthaler Förderkreis für angewandte Mechanik im Montanwesen e.V., 1992

[S94] Saksena, M. C. (Directed by: Agrawala, K.; Gerber, R.), *"Parametric Scheduling for Hard Real-Time Systems"*, Dissertation, UMD CS-TR-3321, UMIACS-TR-94-88, University of Maryland, 1994

[Sak98] Saksena, M., *"Real-Time System Design: A Temporal Perspective"*, Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE), May 1998

[SBL94] Sun, J.; Bettati, R.; Liu, J. W.-S., *"An End-to-End Approach to Schedule Tasks with Shared Resources in Multiprocessor Systems"*, IEEE Wirkshop on Real-Time Operating Systems and Software, 1994

[SGA93] Saksena, M. C.; Gerber, R.; Agrawala, A. K., *"Scheduling with Relative Timing Constraints"*, University of Maryland, 1993