Fakultät 6.2 - Informatik
Universität des Saarlandes

Ressourcenoptimierung von Workflow Problemen

Diplomarbeit

Angefertigt unter der Leitung von Prof. Dr. Gert Smolka
Zweitgutachter Prof. Dr. Gerhard Weikum

Betreuung durch Dr. Christian Schulte und Dr. Tobias Müller

MARTIN HOMIK

25. April 2002

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Saarbrücken, den 25.04.2002

Martin Homik

Zusammenfassung

Aktuelle Workflow Management Systeme sind in der Lage, die Durchlaufzeit einzelner Prozesse schon in der Planungsphase zu optimieren. Sie vernachlässigen jedoch die Optimierung von Organisation und Infrastruktur. Dies gilt insbesondere für flexible Prozesse mit kontinuierlicher Versorgung.

Eine Optimierung von Organisation und Infrastruktur wird aufgrund von zunehmender Komplexität notwendig. Dies betrifft vor allem die Wirtschaft, in der Zeit und Kapital eine wesentliche Rolle spielen.

Diese Arbeit stellt das mathematische Modell und die Implementierung von *Woop* vor. *Woop* löst Optimierungsprobleme für flexible Prozesse mit kontinuierlicher Versorgung unter Anwendung der Constraint Programmierung. Optimale Lösungen werden effizient, exakt und generisch approximiert.

Die Implementierung basiert auf dem System Mozart und der Programmiersprache Oz.

Danksagung

Mein größter Dank gebührt meinen Eltern, die mir das Studium ermöglicht haben, und mich während der gesamten Studienzeit unterstützen.

Weiterhin danke ich meinen Betreuern Christian Schulte und Tobias Müller, die mir dir Tür zur Constraint Programmierung weit geöffnet und damit mein Interesse geweckt haben. Außerdem danke ich ihnen für die zahlreichen Tipps rund um Mozart, Constraints und Diplomarbeit. Ferner danke ich Prof. Dr. Gert Smolka und dem gesamten Lehrstuhl für das hervorragende Arbeitsumfeld.

Weiterer Dank geht an Daniel Bobbert, Thorsten Brunklaus und Esther Niederberger für ihre Ideen und Korrekturen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Diese Arbeit stellt das mathematische Modell und die Implementierung von *Woop* vor. *Woop* löst Optimierungsprobleme für flexible Prozesse mit kontinuierlicher Versorgung unter Anwendung der Constraint Programmierung. Optimale Lösungen werden effizient, exakt und generisch approximiert.

1.1 Workflow Management

Effiziente Geschäftsprozesse in Unternehmen erfordern informationstechnische Infrastrukturen. Ein Workflow Management System (kurz: WFMS) koordiniert die zu einem Geschäftsprozeß gehörenden Aktivitäten gemäß einer Ablaufspezifikation für den vorliegenden Prozeßtyp. Die Bearbeitung von Aktivitäten übernehmen Workflow Teilnehmer.

Moderne Unternehmen sind darauf angewiesen, daß ein WFMS eine große Zahl (*kontinuierliche Versorgung*) von Geschäftsprozessen verschiedenster Typen (*flexible Prozesse*) verläßlich und ohne sichtliche Verzögerung ausführt. Dabei beschreiben drei essentielle Punkte jeden Prozeß: Prozeßlogik, Organisation und Infrastruktur.

Die *Prozeßlogik* gibt den Ausführungszeitpunkt einer Aktivität vor. Dagegen legt die *Organisation* fest, welche Aktivität von welchem Teilnehmertyp ausgeführt wird. Prozeßlogik und Organisation garantieren die korrekte Arbeitsweise des WFMSs.

Die *Infrastruktur* beschreibt, wieviele Teilnehmer eines bestimmten Typs notwendig sind. Die Infrastruktur eines WFMS ist maßgebend für Invest und Durchsatz.

Aus Sicht der Investoren besteht das wichtigste Planungsproblem des Workflow Management darin, für eine gegebene Menge von Ablaufspezifikationen und für einen Mindestdurchsatz die kostengünstigste Organisation und Infrastruktur zu finden. Hierbei darf nur auf Teilnehmertypen aus einer gestellten Menge zurückgegriffen werden.

1.2 Motivation

Für die Planer von WFM-Systemen ist es eine schwierige Aufgabe, eine kostengünstige Infrastruktur und Organisation zu finden. Umso mühsamer gestaltet sich bei komplexen Prozessen die Suche nach der optimalen Lösung.

Neben der Kostenoptimierung fordert die Wirtschaft in der Praxis weitere Konzepte:

Betriebsmodus. Es wird zwischen zwei Modi unterschieden: separat und vermischt. Im *separaten* Betriebsmodus startet das WFMS ausschließlich Prozesse eines bestimmten Typs. Im *vermischten* Betriebsmodus hingegen werden Prozesse verschiedener Typen im vorgegebenen Verhältnis gestartet.

Gerichteter Datenfluß. Ein Objekt, das nach einer Ablaufspezifikation bearbeitet wird, hat eine feste Sequenz von Teilnehmertypen zu durchlaufen. D.h. der Transport von Objekten wird gesteuert. Dadurch ist eine exakte Bestimmung der effektiven Taktzeit, Organisation und Infrastruktur möglich.

Attributierte Ressourcen. Teilnehmertypen erlangen durch Eigenschaften (Fähigkeiten, lokale Zeitangaben, Stabilität, Kosten, Werkzeuge etc.) präzisere Züge. Jede Eigenschaft übt Einfluß auf Organisation und Infrastruktur aus, was die Komplexität der Problemstellung steigert. Den größten Anteil an der Komplexität haben die Fähigkeiten. Ein Teilnehmertyp ist nicht notwendigerweise ein Spezialist. Die Fähigkeiten der Teilnehmertypen dürfen beliebig überlappen.

Zur Beschreibung der genannten Problemstellung und zur Entwicklung eines Lösungsverfahrens bedarf es eines genauen, mathematischen Modells, das als Basis ein erweiterbares Grundmodell hat.

1.3 Beitrag der Arbeit

Diese Arbeit stellt eine neue Klasse von Planungsproblemen vor. Es handelt sich um die Optimierung von Organisation und Infrastruktur flexibler Prozesse mit kontinuierlicher Versorgung.

Trotz eines breiten Anwendungsspektrums wird das Planungsproblem erstmals in dieser Arbeit mathematisch abstrahiert und als ein Workflow Problem modelliert. Dabei wird zunächst ein Grundmodell vorgestellt, das sukzessiv um präzise Informationen (lokale Zeit, Werkzeuge, vermischte Betriebsart) erweitert wird. Das mathematische Grundmodell erlaubt genauere, formale Untersuchungen des Problems.

Als Lösungstechnik wird die Constraint Programmierung gewählt. Sie erlaubt eine direkte Umsetzung des mathematischen Modells. Die Umsetzung beinhaltet sowohl aus dem Modell hervorgegangene Constraints als auch spezielle, auf das Problem zugeschnittene Heuristiken. Die Implementierung trägt den Namen *Woop (Workflow Optimizer)* [16] und basiert auf dem System Mozart [23] und der Sprache Oz [31].

Woop ist in der Lage, korrekte und exakte Lösungen generisch aus partiellen Informationen zu berechnen. Das Verfahren ist effizient, skalierbar und approximiert schnell optimale Lösungen.

Darüber hinaus enthält *Woop* interaktive Komponenten, die den Anwender dabei unterstützen, Einfluß auf die Lösungsverfahren auszuüben. Durch benutzerdefinierte Modellierungsanweisungen werden dem System partielle Informationen über den Lösungsraum mitgeteilt. Ferner kann der Anwender verschiedene Heuristiken erstellen und testen.

1.4 Aufbau der Arbeit

Das nachfolgende Kapitel 2 stellt das vorliegende Optimierungsproblem flexibler Geschäftsprozesse mit kontinuierlicher Versorgung vor. Kapitel 3 gibt eine Übersicht zur Programmierung mit Constraints. Es bildet die Grundlage für die nachfolgenden Kapitel. Anschließend wird in Kapitel 4 ein mathematisches Modell für das Problem eingeführt. In Kapitel 5 werden Eigenschaften des Problems und des Modells untersucht, um gegebenfalls weitere Einschränkungen des Suchraums zu erfassen. Das Thema Suche und Heuristik wird detailiert in Kapitel 6 behandelt. Die Kernpunkte der Implementierung von *Woop* werden in Kapitel 7 behandelt. In Kapitel 8 werden die vorgestellten Aspekte evaluiert. Das letzte Kapitel faßt die Ergebnisse zusammen. Abschließend werden offene Fragen und der Ausblick diskutiert.

Kapitel 2

Workflow Problem Spezifikation

Der Begriff *Workflow* wird von der Workflow Management Coalition (WfMC) [33] als teilweise oder vollständige Automatisierung eines Geschäftsprozesses definiert, während der Dokumente, Informationen oder Aufgaben von einem Teilnehmer (einer Ressource) zum anderen gereicht werden, um sie nach einer Menge von prozeduralen Regeln zu bearbeiten.

In diesem Kapitel wird das zu untersuchende Problem vorgestellt. Es handelt sich um ein Optimierungsproblem für einen Workflow, der aus flexiblen Geschäftprozessen mit kontinuierlicher Versorgung besteht. Die hier benutzten Begriffe sind an die standardisierten Definitionen der WfMC angelehnt und werden teilweise erweitert.

Dieses Kapitel beginnt zunächst mit der Erklärung der Grundbegriffe von Geschäftsprozessen. Insbesondere wird das Workflow Problem mit kontinuierlicher Versorgung von klassischen Schedulingproblemen abgegrenzt. Anschließend wird erklärt, wie eine Menge von Prozessen im Datenfluß eingesetzt wird, und wie jeder Ablaufplan eines Prozesses zu lesen ist. In jedem der folgenden Abschnitte werden Teile des Lösungsweges skizziert.

2.1 Grundbegriffe

Ein *Geschäftsprozeß (Business Process)* beschreibt die in einer realen Welt miteinander vernetzen Teilprozesse bzw. Aktivitäten, die gemeinsam eine übergeordnete Aufgabe lösen. Beispiele hierfür sind Reklamation oder Bestellannahme in einem Call Center. Ein Bestellvorgang besteht aus mehreren aufeinander folgenden Arbeitsschritten, die von Personen, Maschinen oder Software ausgeführt werden.

Die Abbildung eines Geschäftsprozesses in eine für ein Workflow Management System verständliche Darstellung wird als *Prozeßdefinition (Process Definition)* bezeichnet. Diese Modellierung beinhaltet alle stattfindenden Aktivitäten, einen Zeitplan bezüglich der Aktivitäten, sowie alle Ressourcen, die diese Aktivitäten ausführen. Die

Prozeßdefinition wird an ein Workflowsystem übergeben, welches mit dieser Information Instanzen der Prozeßdefinition ausführen kann. Eine Prozeßdefinition, die ausschließlich auf Computern eingesetzt wird, heißt *Workflowdefinition* und ihre Instanz einfach *Workflow* .

Jeder Geschäftsprozeß besteht aus drei orthogonal aufgebauten Ebenen. Leymann und Roller legen die Ebenen in [19] wie folgt fest:

1. **Prozeßlogik**
   Welche Aktivität soll zu welchem Zeitpunkt ausgeführt werden?
2. **Organisation**
   Wer führt welche Aktivität aus?
3. **Infrastruktur**
   Welche Ressourcen sind in welcher Anzahl notwendig?

Diese Arbeit widmet sich der Aufgabe, Prozeßdefinitionen aus partiellen Informationen gemäß der formulierten Ziele und Kriterien zu bestimmen. Zu den wichtigsten Eingangsinformationen gehören relative Ablaufvorgaben sowie Ressourcentypen.

Unter einem Ablauf versteht man eine relative Reihenfolge von Aktivitäten. Im Gegensatz zu einem Zeitplan, in dem der Ausführungszeitpunkt jeder Aktivität bekannt ist, legt der relative Ablauf zeitliche Rahmenbedingungen fest, die durch Relationen zwischen Aktivitäten ausgedrückt werden. Im folgenden wird von einem *Ablauf* gesprochen, wenn die relative Reihenfolge von Aktivitäten gemeint ist, und es wird von einem *Zeitplan* gesprochen, wenn die exakten Ausführungszeiten gemeint sind.

Den Zeitpunkt und die Art der Prozeßausführung bestimmt der *Modus*. Er legt fest, wieviele verschiedene Prozeßdefinitionen gleichzeitig instanziiert werden dürfen. Ablauf und Modus bilden zusammen den *Datenfluß*.

Beispiel 2.1.1. In einem Call Center für Druckaufträge wird eine Bestellung per Telefon von einem Operator entgegengenommen. Anschließend wird sie an die Druckabteilung weitergeleitet, wo ein Drucker den Ausdruck übernimmt. Zuletzt stellt der Lieferbote die Ware zu. Die Aufgabe besteht darin, die kostengünstigste Prozeßdefinition zu bestimmen.

In diesem Beispiel werden drei Ressourcen angegeben: Operator, Kopierer und Lieferbote. Der Lieferbote ist ein Mensch, der Drucker eine Maschine und der Operator kann entweder ein Mensch oder eine mit Sprachtechnologie ausgestattet Software sein. Die nachfolgende Tabelle hält die Ressourcen mitsamt ihren Kosten fest.

T	Ressource	Kosten
1	Start	0
2	Operator (Software)	2500
3	Operator (Mensch)	3000
4	Drucker	3500
5	Lieferbote	1500
6	Ende	0

Die Ressourcen für Start und Ende verursachen keine Kosten. Sie simulieren Ressourcen, die eine Start- und eine Endaktivität ausführen und dienen lediglich als Orientierungspunkte. Diese Ressourcen werden auch *Sonderressourcen* genannt. Desweiteren erkennt man, daß die Auswahl zwischen einem Menschen und einem Programm als Operator gegeben ist. Hierbei ist das Programm günstiger, da es keine monatlichen Gehaltskosten verursacht. Allerdings sollte der Mensch zumindest aufgrund der Vielfalt seiner Fähigkeiten den Vorzug erhalten. Die endgültige Wahl wird entsprechend der geforderten Aktivitäten getroffen.

Die Aktivitäten im Beispiel unterliegen einer typischen, relativen Reihenfolge. Die nachfolgende Tabelle listet alle geforderten Aktivitäten mitsamt ihrer Kodierung (Spalte *A*) auf. In der Prozeßdefinition werden üblicherweise ganze Zahlen als Namen für Aktivitäten verwendet. Zu jeder Aktivität wird zusätzlich angegeben, welche Aktivitäten zuvor ausgeführt werden müssen, und welche Ressource (Spalte *T*) diese ausführen kann.

A	Aktivität	Vorgänger	T
0	Start	—	1
11	Lieferboten anfordern	0	2,3
12	Druckbestellung aufnehmen	0	2,3
13	Anreise Lieferbote	11	5
14	Drucken	12	4
15	Cover erstellen	12	3
16	Sortieren & Binden	14,15	4
18	Lieferung	13,16	5
19	Fahrzeug überprüfen	16	5
100	Ende	18,19	6

Analog zu der Start- und Endressource liegt die Start- und Endaktivität vor. Die Startaktivität hat keinen Vorgänger. Sowohl ein Mensch als auch ein Programm kann in der Rolle des Operators Lieferboten anfordern und Druckbestellungen entgegennehmen. Aber nur ein menschlicher Operator ist in der Lage ein passendes, fantasievolles Cover zu entwerfen. Daraus ist zu schließen, daß mindestens ein menschlicher Operator in der Prozeßdefinition vorkommen muß, da alle aufgeführten Aktivitäten ausgeführt werden müssen. Ob die Prozeßdefinition nur menschliche oder beide Operatortypen enthält, hängt davon ab, welches Ziel und welche Vorgaben gesetzt wurden.

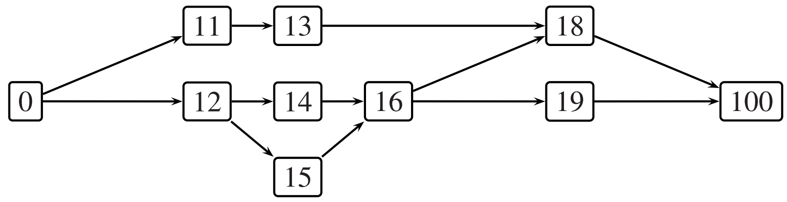


Abbildung 2.1: Ablauf: Bestellannahme einer Druckware

Zur Veranschaulichung eines Ablaufs werden *Flußdiagramme* oder allgemeiner ausgedrückt, gerichtete, azyklische und zusammenhängende *Graphen* benutzt. Die Knoten eines Graphen entsprechen Aktivitäten, während Kanten als Vorrangsrelationen interpretiert werden. Eine Kante von a nach b bedeutet, daß Aktivität a unbedingt vor Aktivität b ausgeführt werden soll. Jeder Graph enthält genau eine Startaktivität (Knoten ohne Vorgänger) und genau eine Endaktivität (Knoten ohne Nachfolger). Zum Beispiel zeigt Abbildung 2.1 einen Graphen der den relativen Aktivitätenfluß der Bestellannahme aus Beispiel 2.1.1 darstellt.

Das Beispiel ist einfach gehalten. Die Fähigkeiten der Ressourcen überlappen sich kaum. Damit ist es leicht, eine konkrete Organisation anzugeben. Bei größeren Prozessen und flexibleren Ressourcen sind die Überlappungen wesentlich größer. Dort fällt die Ermittlung einer guten Organisation schwieriger aus. Es ergeben sich vielfältige Kombinationsmöglichkeiten, deren Berechnung ohne Computerunterstützung unmöglich ist.

Eine *Prozeßinstanz (Process Instance)* ist eine zustandsbehaftete, konkrete Ausführung einer Prozeßdefinition. Der Zustand wechselt nach jeder Beendigung einer Aktivität, und gibt Auskunft über das Voranschreiten der Instanz. Innerhalb einer Prozeßinstanz wird ein *Objekt (Object)* von Ressource zu Ressource gereicht und durch Aktivitäten verändert. Der aktuelle Zustand der Instanz ist am Objekt nachvollziehbar. Das Objekt entspricht dem Produkt eines Geschäftsprozesses, beispielsweise der Lieferung einer Druckware.

2.1.1 Kontinuierliche Versorgung

Bekannte Schedulingprobleme wie zum Brückenbau [5, 10] oder JobShop [2, 11, 17, 5, 10] suchen nach Lösungen, die insbesondere die Prozeßlogik optimieren. Hierbei ist lediglich die Durchlaufzeit eines einzelnen Objektes relevant. Die *Durchlaufzeit* ist die Zeit, die die Bearbeitung eines vollständigen Prozesses in Anspruch nimmt. Es wird demnach versucht, eine passende Organisation anzugeben, die eine optimale Prozeßlogik erfüllt. Problematisch wird es, wenn die Organisation und die Infrastruktur kontinuierlich mit neuen Objekten versorgt werden. Dann ist die Frage nach dem Durchsatz von Objekten viel interessanter, d.h. wieviele Objekte können pro Zeitraum einen kompletten Prozeß durchlaufen?

Die kontinuierliche Versorgung kann nur so schnell wie die langsamste Ressource sein (*bottleneck* Problematik). Eine Ressource ist nicht unbedingt im Sinne der Geschwindigkeit langsam. Vielmehr muß darauf geachtet werden, ob sie langandauernde Aktivitäten besitzt, und ob die Organisation der Ressource übermäßig viele Aktivitäten zugewiesen hat. Übertragen auf das Beispiel 2.1.1 heißt dies, daß die Auslieferung einer Drucksache wesentlich mehr Zeit in Anspruch nimmt als beispielsweise der Druck selbst. Es wird also die Bearbeitungszeit eines Teilnehmers für eine Menge von Aktivitäten betrachtet. Setzt man die Bearbeitungszeit ins Verhältnis zur Anzahl der Teilnehmer, so ergibt dies die *effektive Taktzeit*. Sie ist das kürzeste Zeitintervall zwischen zwei Prozeßinstanzen.

Angenommen, es liegt eine Prozeßdefinition vor, in der eine Ressource T die maximale Taktzeit von 500 Minuten aufweist. Dann wird alle 500 Minuten ein neuer Prozeß instanziiert. Steht der Auftraggeber unter Zeitdruck und will alle 300 Minuten einen Prozeß starten, dann würde spätestens an der Ressource T ein Stau entstehen, der fortwährend wachsen würde. Folgerichtig muß die Taktzeit für T vermindert werden. Hierzu stehen zwei Möglichkeiten zur Verfügung:

Umverteilung von Aktivitäten. Falls möglich, sollten Aktivitäten auf andere Ressourcen umverteilt werden. In diesem Fall ist eine Überlappung von Aktivitäten verschiedener Ressourcen sehr hilfreich. Weniger beschäftigte Ressourcen füllen nun ihre Wartezeiten mit den Aktivitäten überladener Ressourcen.

Aufstocken von Ressourcen. Ist eine Umverteilung nicht möglich, oder führt sie zu keinem besseren Ergebnis, muß eine weitere Ressource desselben Typs wie T hinzugefügt werden. Damit entlastet T_2 die Ressource T_1. Die Taktzeit würde sich für T_1 auf 250 Minuten halbieren.

Ziel ist eine Ausbalancierung aller Taktzeiten. Das Ergebnis der Ausbalancierung ist eine veränderte Infrastruktur. Durch die Hinzunahme einer weiteren Ressource wird der Infrastruktur ein weiterer Aspekt hinzugefügt: die Erhöhung der Kosten. Die Kosten sind die wichtigsten Faktoren in der Wirtschaft und müssen so gering wie möglich gehalten werden. Sowohl die Infrastruktur als auch die Kosten sind Punkte, die von typischen Schedulingproblemen nicht bedacht werden. Im Gegensatz dazu entwerfen die Schedulingalgorithmen einen exakten Zeitplan für die Problemstellung. Die hier vorgestellte Klasse von Problemen begnügt sich mit einem relativen Ablauf. Es ist unwesentlich, in welcher Reihenfolge eine Ressource ihre Aktivitäten ausführt. Die Taktzeiten und die Kosten sind davon völlig unabhängig, womit eine Festlegung auf einen Zeitplan wegfällt.

Ohne auf die exakte Ermittlung der Infrastruktur zum oben genannten Beispiel 2.1.1 einzugehen, die in Kapitel 4 näher erläutert wird, soll dennoch eine geeignete Lösung angeben werden. Abbildung 2.2 gibt eine vollständige Organisation und Infrastruktur an. Demnach werden drei Operatoren benötigt sowie zwei Drucker und fünf Lieferboten.

2.1.2 Aktivitäten

Eine *Aktivität (Activity, Job)* ist ein logischer, atomarer Schritt innerhalb einer Prozeßdefinition. Jede Aktivität wird in einer Prozeßinstanz von einer Ressource genau einmal ausgeführt. Ziel ist es, eine gute Organisation anzugeben, die zu günstigen Kosten und zu einer minimalen, effektiven Taktzeit führt. Beide Größen werden durch die Bearbeitungszeit einer Aktivität sowie die Werkzeuge einer Aktivität beeinflußt.

Die Zeit, die die Ausführung einer Aktivität in Anspruch nimmt, wird *Bearbeitungszeit (Processing Time)* genannt. Die Summe aller Bearbeitungszeiten in einem Block ist ein wesentlicher Bestandteil der *Blockbearbeitungszeit*, welche großen Einfluß auf die Anzahl von Teilnehmern hat (siehe Abschnitt 2.4). Hierbei wird zwischen globaler und

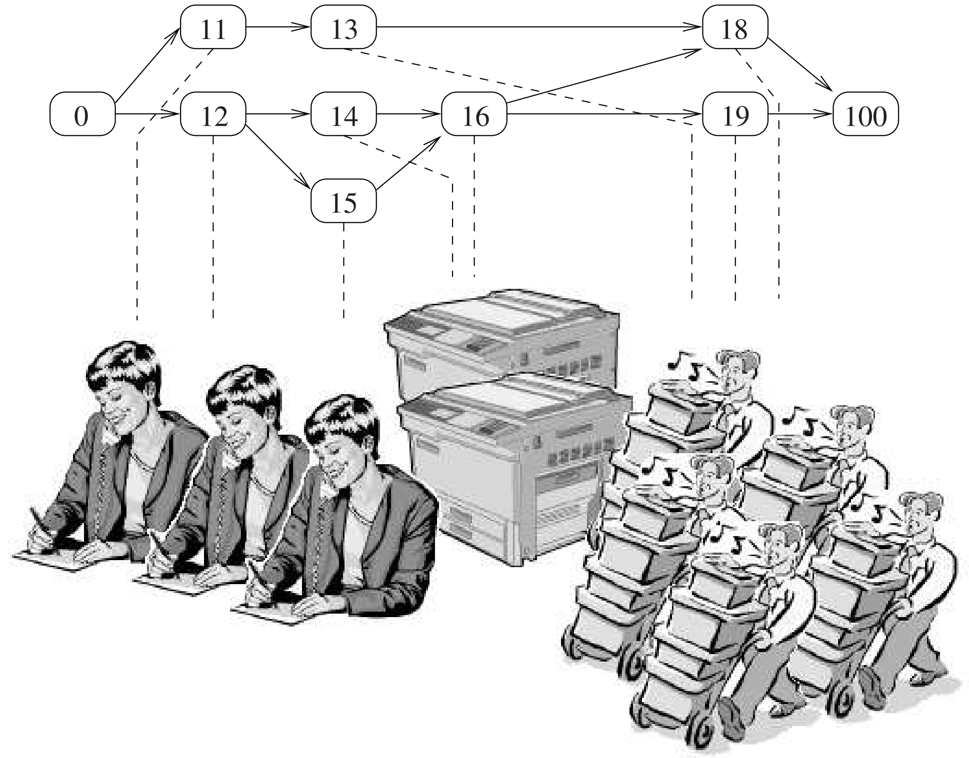


Abbildung 2.2: Organisation und Infrastruktur: Bestellannahme einer Druckware

lokaler Bearbeitungszeit unterschieden, wobei vor der Planung eine Zeitart festzulegen ist.

Globale Bearbeitungszeit. Jede Aktivität wird, gleich von welcher Ressource, immer in der gleichen Zeit bearbeitet. Man betrachte hierzu erneut das Beispiel 2.1.1. Demnach würde die Aktivität `Lieferboten anfordern` von beiden Operatoren Software und Mensch die gleiche Zeit in Anspruch nehmen.

Lokale Bearbeitungszeit. Diese Angabe ist ressourcenspezifisch. Je nachdem welche Ressource eingesetzt wird, ändert sich die Bearbeitungszeit. Diese Angabe ist der Realität näher, da eine Software automatisch ein Signal absetzen kann, wohingegen der Mensch lange Telefonierzeiten braucht. Der Nachteil der gewonnenen Präzision ist die Unbestimmtheit der Bearbeitungszeit. Obwohl der relative Zeitpunkt der Aktivität möglicherweise einfach zu bestimmen ist, muß der Ressourcentyp nicht unbedingt bekannt sein. Demnach kann nur ein Bereich der Bearbeitungszeit angegeben werden: der zwischen der kürzesten und längsten Dauer.

Hat eine Ressource zwei Aktivitäten zu bearbeiten, benötigt auch der Wechsel von der einen Aktivität zur anderen eine Zeitspanne, *Übergangszeit (Transition Time)* genannt. Die Übergangszeit wird als eine Konstante angenommen. Dies reduziert die Komplexität, da nicht zwischen allen möglichen Aktivitätspaaren unterschieden wird. In der

Regel sind die Abweichungen von der durchschnittlichen Übergangszeit zu gering, um signifikant zu sein. Auch die Summe der Übergangszeiten in einem Block ist ein Teil der Blockbearbeitungszeit.

Eine Erweiterung von Aktivitäten ist die Angabe der zu benutzenden *Werkzeuge (Activity Tools*). Eine Aktivität wird von einer Ressource nur dann ausgeführt, wenn ihr Werkzeugmagazin die Werkzeuge, die für die Aktivität benötigt werden, bereitstellt. Werkzeuge sind demnach eine weitere Definition für Aktivitäten und steigern damit die Komplexität des Problems. Da die Aktivitäten atomar sind, bleiben die Werkzeuganwendungen ebenfalls atomar. Die Anwendungsreihenfolge der Werkzeuge ist nicht relevant. Weitere Informationen zu Werkzeugen werden im nächsten Abschnitt vorgestellt.

Aktivitäten können zu einer Aktivitätengruppe (*Activity Block*) bezogen auf bestimmte Eigenschaften, wie zum Beispiel die Zugehörigkeit zu einem gemeinsamen Ressourcentyp, zusammengefaßt werden. Aus dem oben genannten Beispiel 2.1.1 gehen die Gruppen $\{0\}$, $\{11, 12, 15\}$, $\{14, 16\}$, $\{13, 18, 19\}$ und $\{100\}$ hervor. Diese Gruppen werden auch *Blöcke* genannt.

2.1.3 Teilnehmer

Die WfMC verwendet für eine Ressource das Synonym *Workflow Teilnehmer (Workflow Participent)*. Im folgenden werden beide Begriffe gleichwertig benutzt. Workflow Teilnehmer können sowohl menschliche, maschinelle als auch softwaretechnische (Intelligente Agenten) Ressourcen sein. Jeder Teilnehmer hat einen bestimmten Typ, der durch eine Reihe von Eigenschaften bestimmt wird. Hierzu zählen insbesondere Einsatzfähigkeiten, Kosten und Stabilität.

Unter *Einsatzfähigkeiten* versteht man die prinzipiell möglichen Aktivitäten, die ein Teilnehmertyp anbietet. In der Realität wird oftmals nur eine Teilmenge dieser Aktivitäten eingesetzt. Die Wahl eines geeigneten Teilnehmertypen hängt von einer zugrundeliegenden Menge von Aktivitäten ab. Es kann nur der Typ zum Einsatz kommen, der alle Aktivitäten dieser Menge durch seine Einsatzfähigkeiten abdeckt.

Zur Ausführung von Aktivitäten werden Werkzeuge benötigt. Dabei darf eine Aktivität auch mehrere Werkzeuge verlangen. Zwei Aktivitäten a_1 und a_2 dürfen auf gemeinsame Werkzeuge zugreifen, wenn beide ein gleiches Werkzeug verwenden. Damit existiert eine *Überlappung* von Werkzeugen.

Jeder Teilnehmertyp hat ein beschränktes *Werkzeugmagazin*. Dies hat zur Folge, daß ein kleines Magazin nur wenige Werkzeuge aufnehmen kann, und somit nur eine kleine Auswahl der Fähigkeiten zur Verfügung stellt. Damit ist die Auswahl von Aktivitäten wichtig. Da alle Aktivitäten ein gemeinsames Werkzeugmagazin teilen, ist eine hohe Überlappung wünschenswert, weil mehr Platz für weitere Werkzeuge und Aktivitäten vorhanden wäre. Ein Werkzeugmagazin kann man mit einem Schreibtisch vergleichen, auf dem begrenzt Platz ist, so daß nur wenige Utensilien darauf gelagert werden, und nur die eingesetzt werden, die für die Aufgaben notwendig sind. Ein Werkzeug auf dem Schreibtisch kann der Computer sein. Er wird sowohl für die

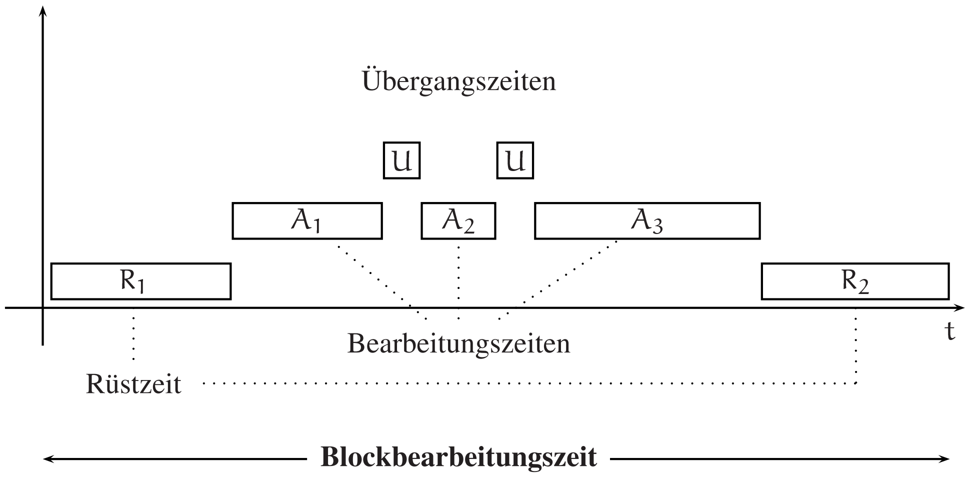


Abbildung 2.3: Zeitablauf

Aktivität `Aufnahme von Kundendaten` als auch für die Aktivität `interne E-Mail Benachrichtigung` verwendet. Durch gemeinsame Nutzung des Computers wird der Fall ausgeschlossen, daß für jede Aktivität ein eigener Computer angeschafft wird.

Eine Instanz einer Aktivität heißt aus Sicht des Teilnehmers, der sie ausführt, *Arbeit (Work Item)*. Ein Teilnehmer kann nur eine Arbeit gleichzeitig verrichten. Stehen mehrere Arbeiten an, so müssen diese an eine *Warteliste (Worklist)* angehängt werden.

Die Annahme eines Objektes zur Bearbeitung kostet Zeit, *Rüstzeit* (*Setup Time*) genannt. Neben der Rüstzeit fallen für jede Aktivität Bearbeitungszeiten an (lokal/global). Den Zeitablauf eines Objektes während der Bearbeitung durch einen Teilnehmer zeigt Abbildung 2.3. Die Bearbeitungszeit eines Objektes in einem Block heißt *Blockbearbeitungszeit* (abgekürzt *bbz*). Die Rüstzeit setzt sich aus den beiden Zeiten R_1 und R_2 zusammen, die während der Annahme und der Abgabe eines Objektes anfallen. Nach der Annahme werden alle dem Teilnehmer zugewiesenen Aktivitäten am Objekt ausgeführt, ohne dabei das Objekt zwischenzeitlich abzugeben. Die Dauer des Wechsels von einer Aktivität auf die nächste ist schließlich die Übergangszeit.

Ein Teilnehmer kann unter Belastung ausfallen. Dadurch treten zwei Probleme auf.

Stillstand. Gibt es keine Ersatzteilnehmer, kann es zu einem Workflow Stillstand kommen.

Stau. Können andere Teilnehmer die Arbeiten eines ausgefallenen Teilnehmers nicht in der geforderten Zeit ausführen, kommt es zu einem Stau. Dies führt zu einer erhöhten Durchlaufzeit (bzw. effektiven Taktzeit).

Beide Probleme werden durch den Einsatz zusätzlicher Teilnehmer gelöst. Die Wahrscheinlichkeit von Ausfällen wird durch eine prozentuale Stabilitätsangabe eines Teilnehmertyps ausgedrückt. Zuverlässige Teilnehmertypen haben eine hohe Stabilität, nahe bei 100 %. Unzuverlässige Teilnehmertypen haben dagegen eine Stabilität nahe bei

0 %. Die Stabilitätsangabe wird direkt zur Berechnung der notwendigen Teilnehmerzahl benutzt. Für weitere Informationen wird auf Abschnitt 4.1.4 verwiesen.

Eine andere Art des Ausfalls ist der Ausfall eines Werkzeugs. Hier gilt eine ähnliche Argumentation wie beim Ausfall eines Teilnehmers. Ein Werkzeugersatz wird *Schwesterwerkzeug* genannt. Die Anzahl der Schwesterwerkzeuge ist vorgegeben. Dies hat extremen Einfluß auf das Werkzeugmagazin. Angenommen, es existiert ein Teilnehmer mit einem Werkzeugmagazin von 30 Plätzen und es gilt, daß jedes Werkzeug im Magazin ein Schwesterwerkzeug hat. Dann sind nur noch 15 Plätze für verschiedene Werkzeuge frei. Wird die Zahl der Schwesterwerkzeuge auf zwei erhöht, so kommen nur noch 10 verschiedene Werkzeuge in Frage.

Auch die Teilnehmer können zu einer gemeinsamen Gruppe eines gemeinsamen Typs (*Worklflow Participant Block*) zusammengefaßt werden. Sie bilden gewissermaßen einen großen Teilnehmer und besitzen eine gemeinsame Warteliste. Liegt noch Arbeit vor und ist ein Teilnehmer bereit, akzeptiert er die Arbeit. Durch solche Gruppenbildungen werden Taktzeit und Ausfallrisiko, welches sonst einen kompletten Workflow blockieren kann, verringert. Die Abbildung 2.10 auf Seite 28 zeigt eine andere Darstellung der Infrastruktur zum oben genannten Beispiel 2.1.1. Nach wie vor besteht es aus fünf Blöcken. Der erste und der letzte Block sind Sonderblöcke, die den Start- bzw. Endteilnehmer darstellen, und kommen daher jeweils einmal vor. Dagegen besteht der zweite Block aus drei, der dritte aus zwei und der vierte aus fünf Teilnehmern.

Die Aufgabe besteht darin, eine passende Infrastruktur zu finden. Zu jedem Block soll jeweils der beste Teilnehmertyp in ausreichender Anzahl ermittelt werden. Für vertiefende Modell- und Lösungsbetrachtungen soll auf das Kapitel 4 verwiesen werden.

2.2 Datenfluß

In diesem Abschnitt werden zwei Eigenschaften des Datenflusses betrachtet: Ablauf und Modus. Der Ablauf gibt die relative Ausführungsreihenfolge von Aktivitäten vor, d.h. wie ein Prozeß durchlaufen wird. Dagegen bestimmt der Modus, wie eine Menge von Prozeßdefinitionen zu behandeln ist. So wird zunächst auf die grafische Repräsentation eines Ablaufs eingegangen, und dann erklärt, welche Ablaufstrategien es gibt. Anschließend werden zwei Modi vorgestellt, die in dieser Anwendung zum Einsatz kommen.

2.2.1 Ablauf

Die Darstellung von Abläufen ermöglichen gerichtete, azyklische und zusammenhängende Graphen. Alle im Graphen vorkommenden Knoten repräsentieren Aktivitäten. Eine Kante ist eine Vorrangsrelation zwischen zwei Aktivitäten, wobei die Quelle der Kante die Vorgängeraktivität und das Ziel die Nachfolgeraktivität ist. Für eine Kante von $\mathfrak{a}$ nach $\mathfrak{b}$ schreiben wir $\mathfrak{a} \to \mathfrak{b}$. Jeder Ablauf enthält genau eine Startaktivität (Knoten ohne Vorgänger) und genau eine Endaktivität (Knoten ohne Nachfolger).

Abbildung 2.4: AND Split / AND Join

Abbildung 2.5: OR Split / OR Join

In diesem Zusammenhang heißt dieser Graph auch *Vorranggraph*. Ein typischer Vorranggraph wurde bereits vorgestellt (siehe Abbildung 2.1 auf Seite 17). Dieser Graph enthält eine Kante von Knoten 12 zu Knoten 14, wobei 12 die Vorgängeraktivität und 14 die Nachfolgeraktivität ist. Diese Kante bedeutet, daß die Aktivität 12 unbedingt vor 14 in der zeitlichen Abfolge bearbeitet werden soll. Aktivitätenpaare, die in keiner Relation zueinander stehen, sind zeitlich unabhängig. Ein Beispiel für diesen Fall sind die Knoten 11 und 12, die auf keinem gemeinsamen Pfad liegen.

Die WfMC schlägt zwei Ausführungsstrategien vor: *Parallel Routing* und *Sequential Routing*. Parallel Routing erlaubt die gleichzeitige Bearbeitung mehrerer Aktivitäten an einem Objekt. Im Gegensatz dazu fordert Sequential Routing die sequentielle Anwendung von Aktivitäten. Ein Prozeß kann beide Strategien beinhalten. Beispielsweise besteht die Reklamation zunächst aus sequentiellen Schritten, wie Aufnahme von Kundendaten und der Beschreibung des Produkts. Die Anfragen über den Verbleib des Produkts werden anschließend parallel auf mehrere Lager verteilt.

In der Darstellung des Parallel Routing werden AND-Splits und AND-Joins (Abbildung 2.4) verwendet. Ein AND-Split verzweigt in mehrere nebenläufige Teilprozesse und kommt am AND-Join wieder zusammen. Dazwischen laufen beliebige Teilprozesse ab.

Desweiteren existieren für Parallel ebenso wie für Sequential Routing Entscheidungsmöglichkeiten (Kontrollfluß), dargestellt durch OR-Splits und OR-Joins (Abbildung 2.5). Je nachdem welcher Fall beim OR-Split eintritt, werden die entsprechenden Aktivitäten in einem Teilprozess ausgeführt. Dies beinhaltet, daß die Aktivitäten, die zu den konträren Teilprozessen gehören, überhaupt nicht bearbeitet werden. Mehrere Teilprozesse münden wieder in einen OR-Join.

Das Workflow Problem enthält sehr einfache Abläufe, die von einem strikten sequentiellen Routing ausgehen. Damit entfallen AND-Splits und AND-Joins. Zu jedem Zeitpunkt wird ein Objekt maximal von einer Aktivität bearbeitet.

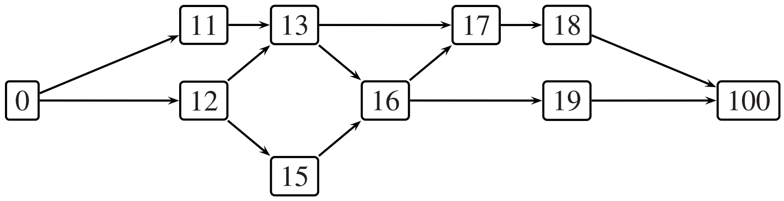


Abbildung 2.6: Ablauf: Reklamation einer Druckware

Darüber hinaus wird verlangt, daß jede Aktivität im Ablauf irgendwann ausgeführt wird. Dies schließt OR-Splits und OR-Joins ebenfalls aus. Allerdings können diese simuliert werden, indem für jede mögliche Alternative eines OR-Splits ein eigener Ablauf angelegt wird. Die Menge der daraus entstandenen Abläufe geht in den Lösungsprozeß ein. Wie dieser Vorgang abläuft, wird im nächsten Abschnitt erklärt.

2.2.2 Modus

Der Problemstellung liegt eine Menge von Abläufen vor. Eine Lösung, insbesondere eine Infrastruktur, muß die Relationen aller Abläufe erfüllen. Auf die reale Welt übertragen heißt dies beispielsweise, daß in einem Call Center eine Bestellung und eine Reklamation von denselben Teilnehmern durchgeführt werden kann.

Für die Bearbeitung mehrerer Prozeßdefinitionen stehen zwei Modi zur Verfügung: der separate und der vermischte Datenfluß.

Im *separaten Datenfluß* darf zu jedem Zeitpunkt maximal eine Prozeßdefinition aktiv vertreten sein, d.h. eine Menge von Teilnehmern nimmt entweder nur Bestellungen oder nur Reklamationen entgegen. Der Graph eines separaten Datenflusses bzw. Ablaufs entsteht aus der Vereinigung der Knoten und Kanten aller Ablaufgraphen der Problemstellung.

Angenommen, Beispiel 2.1.1 wird durch einen weiteren Ablauf, wie in Abbildung 2.6 dargestellt, ergänzt. Die Interpretation des neuen Ablaufs soll hier nicht von Belang sein. Es soll sich um eine allgemeine Form der Reklamation von Druckwaren handeln. So erhält man aus beiden Abläufen einen Graphen für den separaten Datenfluß, indem die Vereinigung wie beschrieben gebildet wird. Abbildung 2.7 zeigt das Resultat.

Dagegen ist im *vermischten Datenfluß* die gleichzeitige Bearbeitung mehrerer Prozeßdefinitionen erlaubt. Der Workflow enthält Objekte aller Prozeßdefinitionen, deren

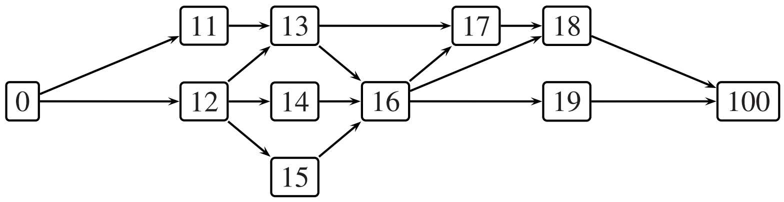


Abbildung 2.7: Beispiel Druckware: separater Datenfluß

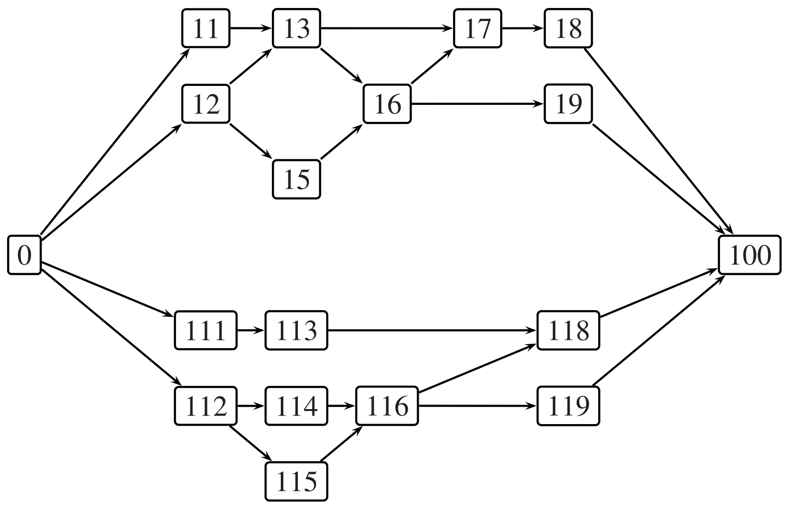


Abbildung 2.8: Beispiel Druckware: vermischter Datenfluß

Zahl einem vorgegebenen Verhältnis entspricht.

Um einen Graphen für den vermischten Datenfluß zu erhalten, muß man in zwei Schritten vorgehen. Erst werden alle Aktivitätsnamen so umbenannt, daß zwei Abläufe nie gemeinsame Aktivitätsnamen besitzen. Start- und Endaktivität bleiben hierbei unverändert. Semantisch gesehen ändert die Umbenennung nichts, da Aktivitäten über Werkzeuge definiert werden. Nach der Umbenennung wird wieder eine Vereinigung aller Abläufe gebildet.

Abbildung 2.8 zeigt einen Graphen für den vermischten Datenfluß zum oben genannten Beispiel der Druckware. Anhand der Graphen für den separaten und vermischten Datenfluß kann die Notwendigkeit der Umbenennung demonstriert werden. Beim separaten Datenfluß wird immer nur nach einer Prozeßdefinition gearbeitet. Es ist zu jedem Zeitpunkt klar, welche Aktivitäten erlaubt und welche verboten sind. Das Workflow System muß zwischen Prozeßdefinitionen umschalten, wenn es neue Aufgaben zu erfüllen hat. Beim vermischten Datenfluß sind mehrere Prozeßdefinitionen gleichzeitig aktiv. Bildet man beispielsweise die Vereinigung aller Abläufe ohne vorherige Umbenennungen, besteht die Gefahr, daß zusätzliche, ungewollte Kanten entstehen. Am Beispiel der Bestellung sieht man, daß die Aktivitäten 12 und 13 voneinander unabhängig sind. Durch die Vereinigung wird jedoch eine Kante $12 \rightarrow 13$ aus der Reklamation hinzugefügt, die die Freiheit der Platzierung von Aktivitäten einschränkt.

2.3 Partitionierung

Ein Mittel zur Bestimmung der Organisation ist die *Partitionierung*. Die Partitionierung ist ein Vorgang, bei dem die Aktivitäten eines separaten bzw. vermischten Datenflusses in paarweise disjunkte Mengen $S_1, S_2, \ldots, S_n$ aufgeteilt werden. Bei diesen Mengen spricht man auch von *Blöcken*. Zu jeder Aktivitätsmenge eines Blocks i gehört eine Infrastruktur, die aus Teilnehmern gleichen Typs bestehen. Jeder Teilnehmer des Blocks i führt nach Erhalt eines Objektes nur die Aktivitäten des Blocks i aus. Neben der Organisation ist ein gerichteter Datenfluß (Abschnitt 2.3.1) ein weiteres Ziel der Partitionierung. Dieser zentrale Vorgang hat größten Einfluß auf Prozeßlogik, Organisation und Infrastruktur.

Die Prozeßlogik wird durch die Abläufe der Problemstellung bestimmt. Jede Kante eines Ablaufs gibt die relative Ausführungsreihenfolge vor. Mit Hilfe der Kanten wird bestimmt, welche Aktivität welcher Menge zugewiesen werden soll, und führt damit eine Ordnung auf den Mengen S_i ein. Es gilt die Regel: ist $a_1 \rightarrow a_2$ eine Kante, die besagt, daß Aktivität a_1 vor a_2 ausgeführt werden soll, so liegen die Aktivitäten entweder in einer gemeinsamen Menge S_i, oder Aktivität a_1 liegt in einer Menge S_i und Aktivität a_2 liegt in einer Menge S_j mit $i < j \leq n$.

Die Partitionierung wird anhand eines gefärbten Graphen veranschaulicht. Aktivitäten, die in einer gemeinsamen Menge liegen, sind einheitlich gefärbt. Die Abbildung 2.9 zeigt einen gefärbten Graphen für den separaten Datenfluß des Beispiels 2.1.1 (siehe Seite 16). Der Graph wurde in fünf Mengen partitioniert. Die Aktivitäten 0 und 100 bilden trotz gleicher Färbung zwei Mengen, um deren Sonderrolle für Start und Ende zu demonstrieren. Die Menge S_2 besteht aus den Aktivitäten 11, 12, 15; die Menge S_3 aus 14 und 16; und schließlich die Menge S_4 aus 13, 17, 18 und 19.

Es muß beachtet werden, daß die hier vorgestellte Zuordnung der Aktivitäten ebenfalls relativen Charakter hat. Das heißt, daß eine Aktivität nicht notwendigerweise auf eine Menge festgelegt werden kann. Es ist wahrscheinlicher, daß eine Aktivität mehreren Mengen zugewiesen werden kann. Dies hat zur Folge, daß alle Möglichkeiten der Partitionierung überprüft werden müssen, indem dem System sukzessive weitere partielle Informationen gegeben werden. Dieser Vorgang heißt *Suche* und wird im Kapitel 6 detailliert vorgestellt.

Auch mit einer teilweisen Partitionierung kann eine geeignete Organisation angegeben werden. Für jede Menge muß untersucht werden, welcher Teilnehmer die geforderten Aktivitäten prinzipiell zur Verfügung stellt. Sind mehrere Möglichkeiten gegeben, so wählt man den günstigsten Teilnehmertyp. Auch die eingesetzten Teilnehmertypen können im Graph dargestellt werden. Die Färbung der Knoten zeigt nicht nur die Angehörigkeit zu einer gemeinsamen Aktivitätenmenge sondern auch zu einem gemeinsamen Teilnehmertyp an.

Sind die Mengen und ihre Teilnehmertypen bekannt, ist es möglich, eine Aussage

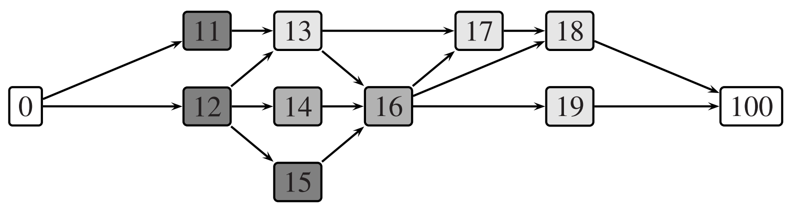


Abbildung 2.9: Beispiel Druckauftrag: Partitionierung separater Datenfluß

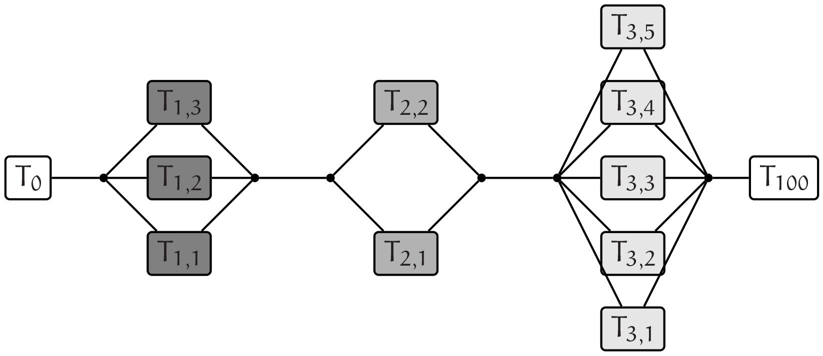


Abbildung 2.10: Anordnung Teilnehmerblöcke

über die Infrastruktur zu geben. Aus der Summe der Bearbeitungszeiten der Aktivitäten wird die Blockbearbeitungszeit pro Block gebildet, die ein wesentliches Maß für die Anzahl der notwendigen Teilnehmer im jeweiligen Block ist. Eine passende Infrastruktur bezüglich des separaten Datenflusses zum oben genannten Beispiel (Abbildung 2.7) zeigt Abbildung 2.10. Man erkennt eine Einteilung des Graphen in fünf Blöcke. Jeder Block ist einheitlich gefärbt und entspricht einem Teilnehmertyp. Die Färbung der Infrastruktur reflektiert die Färbung des Ablaufgraphen. Die Aktivitäten 11, 12 und 15 der Menge S_2 werden vom Teilnehmertyp T_1, der dreifach vorkommt, ausgeführt. Zufällig handelt es sich hier um die gleiche Infrastruktur wie für die Bestellannahme. Dies ist ein Indiz dafür, daß zwischen Bestellung und Reklamation nur geringe Unterschiede vorliegen.

2.3.1 Gerichteter Datenfluß

Die Partitionierung ist das Instrument, das einen gerichteten Datenfluß gewährleistet. Durch sie kann eine geeignete Teilnehmerordnung (Abbildung 2.10) angeben werden. Diese Ordnung ist so zu interpretieren, daß kein Objekt, welches von einem beliebigen Teilnehmer T aktuell bearbeitet wird, im nächsten Arbeitsschritt an einen Vorgängerteilnehmer von T weitergereicht wird. Der nächste Arbeitsschritt kann somit nur von T selbst oder von einem Nachfolger ausgeführt werden.

Der gerichtete Datenfluß gibt eine bessere Abschätzung der effektiven Taktzeit an, da sich alle Objekte nur in eine Richtung bewegen. Gäbe es diese Ordnung nicht, wäre es ungewiß, wieviel Zeit ein Objekt außerhalb seiner Ressourcen (Transport, Wartestellung) verbringt.

2.3.2 Sonderaktivitäten

Sonderaktivitäten, wie Start und Ende (0 und 100), können simuliert werden, indem ihre Bearbeitungszeiten und die Kosten ihrer speziellen Teilnehmertypen auf Null gesetzt werden. Somit spielen sie in der Ermittlung der Gesamtkosten keine Rolle. Weitere Sonderaktivitäten können überall in Ablaufsplänen platziert werden. Sie bieten

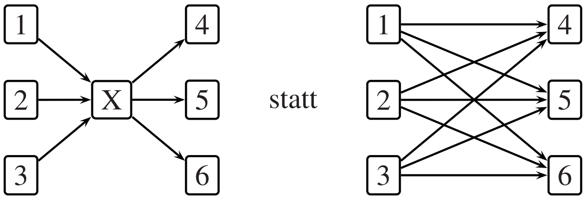


Abbildung 2.11: Mehr Übersicht durch Sonderaktivitäten

Einsatzmöglichkeiten als Sammelknoten für Graphen, die für eine bessere Überschaubarkeit sorgen. Insbesondere eignen sie sich zur Vereinfachung von Graphen (siehe Abbildung 2.11).

2.4 Infrastruktur

Die Infrastruktur macht den Hauptanteil des Gesamtinvests aus. Der Gesamtinvest besteht aus den Kosten pro Block. Zu jedem Block wird ein Teilnehmertyp in bestimmter Anzahl ermittelt. Der Teilnehmertyp hängt hauptsächlich von der Partitionierung ab. Die Anzahl der Teilnehmer pro Block ist von mehreren Variablen abhängig. Hierzu zählen die Größen maximale Taktzeit, Blockbearbeitungszeit, Stabilität des Teilnehmers und Verlust.

Maximale Taktzeit. Durch die kontinuierliche Versorgung wird die Taktzeit vermindert. Eine weitere Verminderung der Taktzeit erfordert mehr Teilnehmer. Je kleiner die vorgegebene, maximale Taktzeit, umso höher fällt der Invest aus.

Blockbearbeitungszeit. Die Blockbearbeitungszeit besteht hauptsächlich aus den Bearbeitungszeiten der Aktivitäten in diesem Block. Eine größere Blockbearbeitungszeit bedingt eine höhere Taktzeit des Blocks, was wiederum zu einer Aufstockung der Teilnehmer führt. Abbildung 2.12 zeigt die Veränderung der Teilnehmerzahl in Abhängigkeit der Blockbearbeitungszeit.

Stabilität. Je stabiler der Teilnehmertyp, desto weniger Ausfälle können langfristig vorkommen. Weniger Ausfälle bedeuten weniger Ersatzteilnehmer.

Verlust. Der Verlust deckt alle übriggebliebenen Fehlerquellen. Je geringer der Verlust, umso weniger Teilnehmer sind notwendig, um ein Objekt in der beabsichtigten maximalen Taktzeit zu bearbeiten.

Man sieht, daß die Zusammenhänge in vielfacher Abhängigkeit zueinander stehen. Während die maximale Taktzeit und der Verlust bekannte Größen sind, wird die Blockbearbeitungszeit und die Stabilität erst zur Laufzeit ermittelt. Beide Größen hängen von der Partitionierung ab. Sie gibt vor, welche Bearbeitungszeiten bzw. Aktivitäten in einem Block vorliegen. Erst danach können Teilnehmer und ihre Anzahl ermittelt werden. Die Abbildung 2.13 zeigt die Vernetzung der Informationen an. Eine gerichtete Kante sagt aus, daß die Quelle ein Parameter für das Ziel ist. So hängt beispielsweise die Blockbearbeitungszeit von der vorliegenden Partitionierung ab. Eine Ausnahme bildet die Beziehung zwischen den Teilnehmertypen und der Partitionierung. Hier

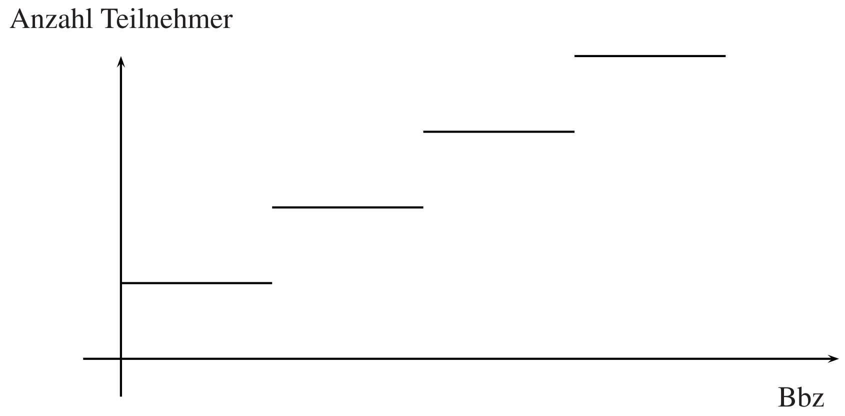


Abbildung 2.12: Anzahl Teilnehmer steigt mit der Blockbearbeitungszeit an.

besteht eine gegenseitige Beeinflussung. Die einzigen Konstanten sind der maximale Takt und der eingerechnete Verlust. Alle anderen Angaben sind Variablen, die zur Berechnung des Gesamtinvest beitragen. Die Änderung einer Variable kann zu einer Veränderung einer anderen führen. Wird beispielsweise die Anzahl der Teilnehmer in einem Block festgelegt, so hat dies Auswirkungen auf die Blockbearbeitungszeit und dementsprechend auch auf die Partitionierung.

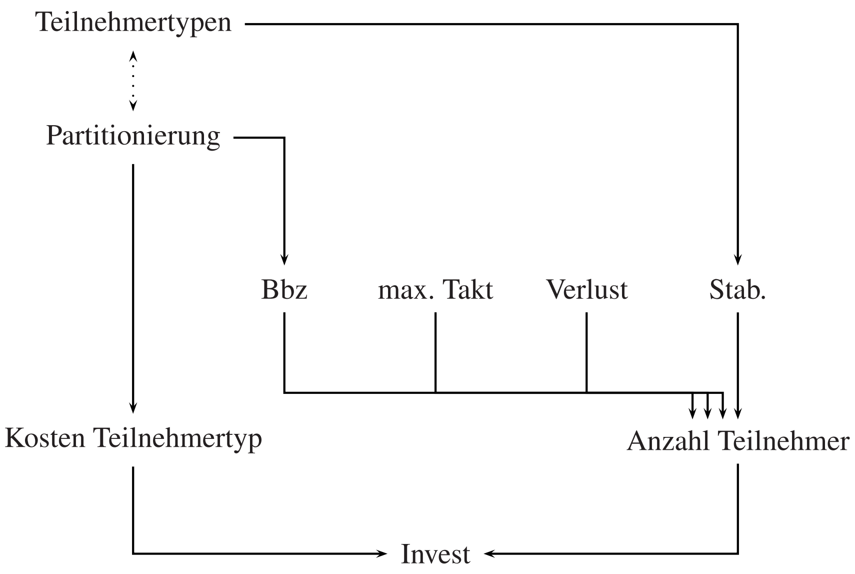


Abbildung 2.13: Parameter für die Investberechnung

Kapitel 3

Constraint Programmierung

Dieses Kapitel stellt Constraint Programmierung (kurz: CP) vor. Es bildet das Fundament für die nachfolgenden Kapitel. Constraint Programmierung eignet sich insbesondere zur Lösung von kombinatorischen Problemen. Darunter fallen klassische Probleme, wie beispielsweise Puzzle, Ressourcenplanung, oder Zuschnittsprobleme [11, 30].

Constraints sind prädikatenlogische Formeln, die Relationen zwischen Variablen ausdrücken. Ziel ist, durch Constraints eine Belegung für die in den Constraints auftretenden Variablen zu finden. Die Belegung muß mit den Constraints konsistent sein. Solange keine Belegung für eine Variable bekannt ist, wird mit partiellen Wertinformationen der Variable operiert.

Constraints können verschiedene Domänen haben. Zu den bekanntesten gehören ganze Zahlen [32], reelle Zahlen [24], Mengen [12] oder Bäume [9]. Diese Arbeit befaßt sich zum einen mit Constraint Programmierung auf endlichen Teilmengen der ganzen Zahlen (*Finite Domain)* [20, 29, 30] und zum anderen mit der Domäne der endlichen, ganzzahligen Mengen (*Finite Set*) [20, 21].

3.1 Constraint Programmierung mit Finite Domains (FD)

In diesem Abschnitt wird die Programmierung mit Finite Domain Constraints vorgestellt. Zunächst werden neue Begriffe eingeführt, mit denen wichtige Mechanismen erklärt werden. Zu diesen gehören: Propagierung, Zerlegung und Exploration.

Finite Domain, Constraint. Ein *endlicher Bereich* (*finite domain*) ist eine endliche Teilmenge der ganzen Zahlen. Ein *Constraint* ist eine prädikatenlogische Formel. Typische Constraints in Finite Domain Problemen zeigen folgende Beispiele:

$$x = 67 \qquad x \in \{0, 1, \ldots, 9\} \qquad x = y \tag{3.1}$$

$$x^2 - y^2 = z^2 \qquad x + y + z < u \qquad x + y \neq 5 * z \tag{3.2}$$

$$x_1, \ldots, x_9 \quad \text{sind paarweise verschieden} \tag{3.3}$$

Domain Constraint. Ein *Bereichsconstraint* (*domain constraint*) wird $x \in D$ geschrieben, wobei D ein endlicher Bereich ist. Ein Bereichsconstraint $x \in \{n\}$ ist äquivalent zu $x = n$.

Basic Constraint. Ein *Basic Constraint* gibt vor, welche Werte eine Variable einnehmen kann. Hierzu gehören folgende drei Formen:

$$x = n \qquad x = y \qquad x \in D,$$

wobei x, y Variablen, n eine Konstante und D ein endlicher Bereich ist. Anwendungsbeispiele hierfür sind in (3.1).

Non-basic Constraint. Ein *Non-basic Constraint* drückt eine Relation zwischen Variablen aus. Beispiele sind in (3.2) und in (3.3) wobei letzterer ein symbolischer Constraint ist.

Finite Domain Problem. Ein *Finite Domain Problem* ist eine endliche Menge P quantorenfreier Constraints, so daß P zu jeder in den Constraints vorkommenden Variable einen endlichen Bereich enthält. Eine *Variablenbelegung* ist eine Funktion, die von einer Variable auf eine ganze Zahl abbildet.

Lösung. Eine *Lösung* ist eine Variablenbelegung, die jeden Constraint in P erfüllt. Betrachtet man nur die Variablen in P, so hat ein Finite Domain Problem höchstens endlich viele Lösungen.

3.2 Propagierung

Constraint Propagierung ist eine Inferenzregel für Finite Domain Probleme, die die endlichen Bereiche von Variablen einschränkt, d.h. sie schließt inkonsistente Werte aus den endlichen Bereichen aus.

Propagierer. Ein *Propagierer* ist eine Implementierung eines Non-basic Constraint. Dabei kann es zu einem Non-basic Constraint auch mehrere Propagierer geben, die sich durch ihre Implementierung unterscheiden. Seine Aufgabe besteht darin, inkonsistente Werte aus den Bereichen seiner Variablen auszuschließen.

Space. Constraint Propagierung findet in einem Umfeld statt, das *Space* genannt wird. Ein Space besteht aus einem *Constraintspeicher* und einer Reihe von Propagierern, die mit dem Constraintspeicher verbunden sind (siehe Abbildung 3.1). Der Constraintspeicher enthält eine Konjunktion von Basic Constraints. Ein an den Constraintspeicher angehängter Propagierer überwacht seine Variablen, die sich im Constraintspeicher befinden. Wird eine dieser Variablen verändert, so versucht der Propagierer den Constraintspeicher zu aktualisieren.

Kommunikation. Propagierer *kommunizieren* miteinander über gemeinsame Variablen. Beispielsweise kommunizieren die Propagierer der Constraints

$$x + y = 9 \qquad 2 * x + 4 * y = 24 * z$$

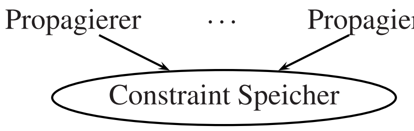


Abbildung 3.1: Space

über die gemeinsam genutzten Variablen x und y. Eine dichte Verkettung der Constraints führt zu einer starken Propagierung.

Constraintspeicher aktualisieren. Ein Propagierer eines Non-basic Constraint c berechnet einen Basic Constraint b_p. Damit wird der Constraintspeicher b_s auf $b_s \wedge b_p$ aktualisiert, falls b_p durch $b_s \wedge c$ impliziert wird, und falls b_p neue und konsistente Informationen liefert.

Determinierung. Ein Constraintspeicher determiniert eine Variable x, falls er den Wert von x kennt, d.h. falls es eine Zahl n gibt, so daß der Speicher den Constraint $x = n$ impliziert.

Inkonsistenz. Ein Propagierer ist inkonsistent, wenn keine Variablenbelegung existiert, die sowohl den Constraintspeicher als auch den durch den Propagierer implementierten Basic Constraint erfüllt. Man spricht von einem *Fehlschlag* (*failure*).

Subsumtion. Ein Propagierer wird subsumiert, wenn jede Variablenbelegung sowohl den Constraintspeicher als auch den vom Propagierer implementierten Basic Constraint erfüllt. In diesem Fall wird der Propagierer gelöscht. Ein Beispiel für diesen Fall ist:

$$x < y \qquad x \in \{3, 4, 5\} \qquad y \in \{6, 7, 8, 9\}$$

Inkonsistenz und Subsumtion wird spätestens beim Festlegen der Variablenwerte des Propagierers entdeckt.

Stabilität. Ein Propagierer ist stabil, wenn er entweder einen Fehlschlag erzeugt, oder wenn er keine neue Information dem Constraintspeicher hinzufügen kann.

Spacezustände. Ein Space schlägt fehl, falls einer seiner Propagierer fehlschlägt. Ein Space ist *stabil*, wenn alle seine Propagierer stabil sind, und ein Space ist *gelöst*, wenn er nicht fehlschlägt und keine Propagierer übrig sind.

Lösung eines Space. Eine Variablenbelegung wird eine Lösung eines Space genannt, wenn sie alle Constraints im Speicher und alle durch die Propagierer implementierten Constraints erfüllt.

3.2.1 Unvollständigkeit

Constraint Propagierung ist unvollständig. Es kann vorkommen, daß ein Space eine eindeutige Lösung hat, die aber durch Constraint Propagierung nicht gefunden wird. Auf der anderen Seite kann es vorkommen, daß Constraint Propagierung einen unlösbaren Space nicht erkennt, weil sie zu keinem Fehlschlag führt. Das folgende Beispiel verdeutlich dies. Es besteht aus drei Propagierern für die Constraints

$$x \neq y \qquad x \neq z \qquad y \neq z,$$

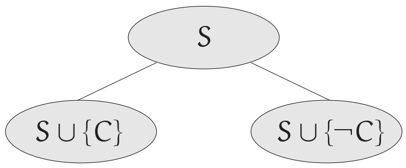
und dem Constraintspeicher

$$x \in \{0, 1\} \qquad y \in \{0, 1\} \qquad z \in \{0, 1\}.$$

Dieser Space hat keine Lösung. Dennoch kann kein Propagierer eine Inkonsistenz feststellen, oder dem Speicher neue Informationen hinzufügen. Aus diesem Grund wird eine Zerlegung notwendig.

3.3 Zerlegung

Liegt ein Space S zwar in einem stabilen, aber weder in einem gelösten noch in einem fehlgeschlagenen Zustand vor, so haben alle Propagierer einen Fixpunkt erreicht. Um fortzufahren wird eine Zerlegung von S notwendig. Hierzu wird ein Constraint C gewählt, mit dem S in zwei neue Space Strukturen zerlegt wird. Dem ersten wird ein Propagierer für den Constraint C hinzugefügt, so daß nun $S \cup \{C\}$ gelöst werden soll. Dem zweiten Space wird ein Propagierer für $\neg C$ hinzugefügt. Der Space S wird in diesem Zusammenhang auch Wahlpunkt (*choice point*) genannt.



Vollständigkeit. Die Kombination von Propagierung und Zerlegung ist eine vollständige Lösungsmethode für Finite Domain Probleme. Ist ein Problem gegeben, so enthält ein Space zunächst nur Basic Constraints und angehängte Propagierer. Nach dem Start der Propagierer erreichen diese nach endlicher Zeit einen stabilen Zustand. Haben sie eine Lösung oder einen Fehlschlag produziert, endet die Berechnung. Andernfalls wird eine Zerlegung gemacht. Die Wahl einer guten Zerlegung bestimmt die Heuristik .

3.3.1 Heuristik

Eine *Heuristik* ist eine Regel, die vorgibt, wie ein Zerlegungsconstraint beschaffen sein muß, um schnell zu einer (guten) Lösung zu kommen. Die Wahl der Zerlegungsconstraints bestimmt somit Größe und Gestalt des Suchbaums. Eine erste schnelle Lösung sollte möglichst wenige Knoten benötigen. Desweiteren sollte die beste Lösung einen möglichst kleinen Suchbaum aufbauen. Beides ist durch eine geeignete Wahl von Zerlegungsconstraints zu erzielen. Die Heuristik wird besser, je mehr Wissen über die Problemstellung bekannt ist.

3.3.1.1 Beispiele

Zu den bekanntesten allgemeinen Heuristiken zählen folgende Beispiele:

Naive. Die einfachste Heuristik heißt *naive*. Sie enummeriert ohne weitere Wissenseinwirkungen alle Belegungsmöglichkeiten. Dabei wählt sie die am weitesten links stehende Variable und weist ihr den kleinstmöglichen Wert zu. Diese Heuristik ist nicht zielgerichtet.

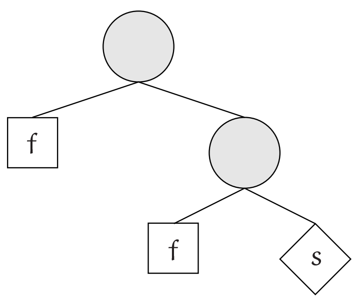
First Fail. Wesentlich effizienter ist die Heuristik *first fail*. Sie wählt die Variable mit dem kleinsten Bereich und weist ihr den kleinstmöglichen Wert zu. Diese Strategie verhindert das bei *naive* auftretende *Thrashing*. Unter Thrashing versteht man eine häufig im Suchbaum auftauchende Inkonsistenz. Im deutschen Sprachraum ist Thrashing auch als Flaschenhalsproblematik bekannt.

NbSusps. Die Heuristik *nbSusps* wählt eine Variable, die in den meisten Constraints vorkommt, in der Hoffnung, durch eine Wertzuweisung möglichst viele Propagierungseffekte in Gang zu setzen.

Neben diesen klassischen Strategien der Variablenenumeration existieren weiterhin Aufspaltungsstrategien [15] und Serialisierungsstrategien [3]. Letztere stammen ursprünglich aus dem Operations Research Gebiet [7] und wurden vielfältig in constraintbasierten Programmiersprachen aufgenommen [4, 8].

3.4 Exploration

Eine Iteration von Propagierung und Zerlegung wird *Exploration* genannt. Dabei wird systematisch ein *Suchbaum* aufgebaut. In der Regel wird nach der *Depth First Search (DFS)* Strategie exploriert. Die nachfolgende Abbildung zeigt einen solchen Suchbaum. Runde Knoten stellen Entscheidungspunkte dar, wohingegen Rechtecke mit einem `f` einen Fehlschlag und Rauten mit einem `s` eine Lösung bezeichnen.



3.4.1 Suchmaschinen und Suchstrategien

Die Objekte, die die Suche durchführen, heißen *Suchmaschinen*. Mozart stellt primär drei Suchmaschinen zur Verfügung: einfache Suche, visuelle Suche [27] und parallele Suche [28]. Die *einfache Suche* durchläuft den Suchbaum und gibt sukzessive

Lösungen an. Der *Explorer* ist eine interaktive, visuelle Erweiterung der einfachen Suche, die während des Suchprozesses den Suchbaum grafisch darstellt. Durch Interaktionen mit der Maus können einzelne Knoten angezeigt und weiter zerlegt werden. Die *parallele Suchmaschine* dagegen teilt die Suche auf mehrere Hosts auf. Dabei bekommt jeder Host einen eigenen Unterbaum zur Durchforstung zugewiesen.

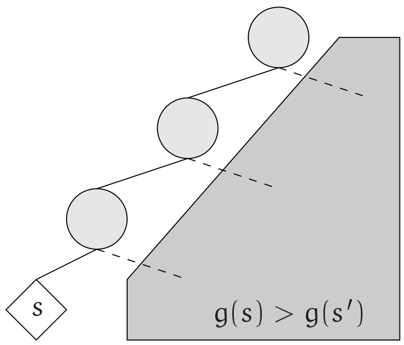
Neben der Differenzierung von Suchmaschinen gibt es noch die Unterscheidung zwischen *Suchstrategien*. Dabei wird vor der Suche festgelegt, ob nach einer, nach allen oder nach der besten Lösung gesucht wird. Letzteres bedient sich der Technik Branch & Bound. Schulte beschreibt in seiner Dissertation [29] ausführlich die Programmierung von Suchmaschinen.

3.4.2 Branch & Bound

Besteht die Aufgabe darin, die optimale Lösung zu finden, muß dem System eine Bewertungsfunktion mitgeteilt werden, welche über die Optimalität entscheidet. Die erste gefundene Lösung s wird zunächst als die optimale Lösung angesehen. Eine Bewertungsfunktion g berechnet einen Vergleichswert $g(s)$. Die nächste Lösung s' muß nun bezüglich eines Kriteriums besser ausfallen. Beispielsweise sind Minimierungen Optimierungsfunktionen:

$$g(s) > g(s').$$

Ein naiver Algorithmus durchläuft den gesamten Suchbaum und überprüft für alle Lösungen das Kriterium. Es ist von Vorteil, schon zu einem früheren Zeitpunkt zu erkennen, daß eine weitere Suche in einem Teilbaum keine bessere Lösung ergeben kann. Da die Suchmaschinen auf Constraints als Eingabeparameter basieren, liegt es nahe, das Kriterium als Constraint der Suchmaschine hinzuzufügen. Die nachfolgende Abbildung verdeutlicht das Vorgehen. Alle nicht vollständig zerlegten Knoten erhalten den zusätzlichen Constraint. Ein Propagierungsvorgang setzt ein und kann schon zu diesem Zeitpunkt ganze Teilbäume ausschließen. Diese Technik heißt *Branch & Bound*.



3.5 Modellierung

Ein Modell ist eine Repräsentation eines Problems als ein Finite Domain Problem. Es spezifiziert die Variablen, die Constraints und die Heuristik, die das Problem ausdrücken.

Nicht triviale Probleme lassen verschiedene Modelle, insbesondere verschiedene Heuristiken, zu. Sie unterscheiden sich unter anderem in der Gestalt ihrer Suchbäume. Die Kunst der Constraint Programmierung besteht darin, ein Modell und eine Heuristik anzugeben, so daß der Ressourcenverbrauch sich in Grenzen hält. Vor allem sollte eine Lösung in akzeptabler Zeit berechenbar sein.

3.6 Constraint Programmierung mit Finite Sets (FS)

Neben der Domäne der ganzen Zahlen existiert eine weitere, sehr nützliche Domäne: die der endlichen, ganzzahligen Mengen (*Finite Sets*). Auch für Finite Sets gilt die Einteilung in Basic und Non-basic Constraints. Desweiteren wirken die bekannten Mechanismen: Propagierung und Exploration. Die hier angegebene Beschreibung der Finite Sets ist an [22] angelehnt.

Sei sup eine beliebige positive ganze Zahl und sei $\mathcal{U} = \{0, \ldots, \mathrm{sup}\}$ eine universelle endliche Menge, so werden im folgenden Teilmengen von $\mathcal{U}$ betrachtet. Dabei werden Konstanten in Kleinbuchstaben und Variablen in Großbuchstaben geschrieben. Der Kleinbuchstabe n steht für eine Konstante und ist ein Element in $\mathcal{U}$; eine Mengenkonstante, die eine Teilmenge von $\mathcal{U}$ ist, wird durch s dargestellt. Eine FD Variable N wird als ein Element in $\mathcal{U}$ und eine Mengenvariable S als eine Teilmenge von $\mathcal{U}$ interpretiert.

Wie auch bei CP(FD) wird bei CP(FS) zwischen Basic und Non-Basic Constraints unterschieden. *Basic Constraints* sind Constraints, deren Erfüllbarkeit und Subsumtion effizient berechnet werden kann.

Basic Constraints. Die wichtigsten Basic Constraints $\mathcal{B}$ bestimmen die obere und untere Schranke einer Mengenvariable:

$$\mathcal{B} \quad ::= \quad \mathcal{B}_1 \wedge \mathcal{B}_2 \quad | \quad s \subseteq S \quad | \quad S \subseteq s \quad | \quad \ldots$$

Die Constraints $n \in S$ und $n \notin S$ sind abgeleitete Formen, da $n \in S \leftrightarrow \{n\} \subseteq S$ und $n \notin S \leftrightarrow S \subseteq \mathcal{U} \setminus \{n\}$, wobei $\mathcal{U} \setminus \{n\}$ eine endliche Menge ist, da $\mathcal{U}$ endlich. Die Kardinalität einer Mengenkonstante s wird mit $\#s$ bezeichnet. Weitere Basic Constraints bieten Kardinaltitätsangaben $\#S$ zu einer Mengenvariable S:

$$\mathcal{B} \quad ::= \quad \ldots \quad | \quad n \leq \#S \quad | \quad \#S \leq n$$

Non-Basic Constraints. Alle anderen Constraints, wie beispielsweise Vereinigung ($\cup$), Durchschnitt ($\cap$), Differenz ($\setminus$) und Disjunktheit ($\|$), sind Non-Basic Constraints

C:

$$\begin{aligned} \mathcal{C} &::= \mathcal{B} \quad | \quad \mathcal{C}_1 \wedge \mathcal{C}_2 \quad | \quad S = \mathcal{E} \quad | \quad S_1 \subseteq S_2 \quad | \quad S_1 \| S_2 \\ \mathcal{E} &::= S_1 \cup S_2 \quad | \quad S_1 \cap S_2 \quad | \quad S_1 \setminus S_2 \end{aligned}$$

Lösungen. Der Erfüllbarkeitstest eines Basic Constraints $\mathcal{B}$ ist einfach. Desweiteren gilt, ist $\mathcal{B}$ erfüllbar, kann es in eine gelöste Form gebracht werden, die für jede Mengenvariable S die größte untere Menge s_{glb}, die kleinste obere Menge s_{lub}[1] sowie die ganzen Zahlen n_{min} und n_{max} enthält, so daß $\mathcal{B}$

$$s_{glb} \subseteq S \subseteq s_{lub} \qquad \wedge \qquad n_{min} \leq \#S \leq n_{max}$$

subsumiert.

Offensichtlich gelten die Eigenschaften $\#s_{glb} \leq n_{min}$ und $n_{max} \leq \#s_{lub}$. Die untere bzw. obere Schranke einer Mengenvariable soll $\mathtt{glb}(S)$ bzw. $\mathtt{lub}(S)$ angeben. Ein erfüllbarer Basic Constraint $\mathcal{B}$ determiniert eine Mengenvariable S genau dann wenn $\mathtt{glb}(S) = \mathtt{lub}(S)$.

Propagierung. Constraint Propagierung der Finite Sets unterliegt den gleichen grundlegenden Mechanismen wie die für Finite Domains. Anhand eines Beispiels soll dieser Vorgang verdeutlicht werden:

Angenommen, der Constraintspeicher beinhaltet die Basic Constraints $\emptyset \subseteq S_1$ und $S_2 \subseteq \{1, \ldots, 5\}$, sowie die Propagierer der Non-Basic Constraints $S_1 \cup S_2 = \{1, \ldots, 5\}$ und $S_1 \| S_2$. Das Hinzufügen der Basic Constraints $1 \in S_1$ und $2 \notin S_2$ ergibt einen zwischenzeitlichen Speicher $\{1\} \subseteq S_1 \subseteq \{1, \ldots, 5\}$ und $\emptyset \subseteq S_2 \subseteq \{1, 3, 4, 5\}$. Die Constraint Propagierung setzt ein und aktualisiert den Constraintspeicher auf den neuen Stand $\{1, 2\} \subseteq S_1 \subseteq \{1, \ldots, 5\} \wedge \emptyset \subseteq S_2 \subseteq \{3, 4, 5\}$. Durch Hinzufügen eines weiteren Basic Constraints $\#S_2 = 3$ determiniert S_1 und S_2 mit dem Constraintspeicher $S_1 = \{1, 2\} \wedge S_2 = \{3, 4, 5\}$.

Verbindung zwischen Finite Domains und Finite Sets. Der Kardinalitätsconstraint $N = \#S$ bildet die Nahtstelle zwischen ganzen Zahlen und Mengen. Das FD Constraint System von Oz bietet darüber hinaus eine Verallgemeinerung der Basic Constraints $n \leq \#S$ und $\#S \leq n$ hin zu den Propagierern

$$N \leq \#S \qquad | \qquad \#S \leq N.$$

Diese beiden Propagierer richten die Verbindung zwischen beiden Systemen ein.

Zerlegung Wie bei Finite Domains reicht die Beschreibung des Problems und die Propagierung nicht aus, so daß Suche notwendig wird. Gefragt sind erneut Zerlegungsconstraints, die bei Finite Sets die Form $n \in S \wedge n \notin S$ haben. Auch hier sind vielfältige Strategien einsetzbar. Zu den wichtigsten Kriterien einer Zerlegung gehören: *Weight*, *Order* und *Elements*. Ähnlich wie bei Finite Domains findet hier eine Zerlegung über einer Sequenz von Finite Sets statt. Mit dem Kriterium *Order* wird eine Menge aus der Sequenz gewählt, während mit *Elements* ein Element der gewählten Menge zugeordnet wird. Hierbei können alle Elemente eine Gewichtung haben, die durch *Weight* angegeben ist.

[1] In der englischsprachigen Literatur heißt es **g**reatest **l**ower **b**ound und **l**east **u**pper **b**ound.

Kapitel 4

Modellierung

In diesem Kapitel wird ein mathematisches Modell für die Problemstellung vorgestellt. Aktivitäten, Teilnehmertypen und Ablaufpläne sind die Grundbestandteile des Modells. Eine Lösung bestimmt die Zuordnung von Aktivitäten und Teilnehmertypen zu Blöcken.

Das Modell wird in zwei Stufen aufgebaut. Das *Grundmodell* enthält alle Aktivitäten mit globalen Bearbeitungszeiten, d.h. eine Aktivität $\mathfrak{a}$ hat auf allen Teilnehmertypen die gleiche Bearbeitungszeit. Es wird hier nur der separate Datenfluß betrachtet.

Im *präzisen Modell* haben Aktivitäten lokale Bearbeitungszeiten, die jeder Teilnehmertyp vorgibt. Zusätzlich verfügen Teilnehmertypen über ein endliches Werkzeugmagazin. Aktivitäten werden durch Werkzeuge definiert. Ein Teilnehmertyp kann eine Aktivität $\mathfrak{a}$, die er prinzipiell anbietet, nur dann ausführen, wenn die benötigten Werkzeuge im Magazin enthalten sind. Aufgrund eines beschränkten Magazins ist die Bereitstellung von Aktivitäten ebenfalls beschränkt. Eine weitere Eigenschaft des präzisen Modells ist der vermischte Datenfluß.

4.1 Grundmodell

4.1.1 Basisstrukturen

Zunächst wird eine Grundmenge von Namen benötigt. Mit Hilfe dieser Namen werden Aktivitäten bezeichnet. Die Namen sollten einheitlich sein, d.h. vom gleichen Typ. In dieser Arbeit werden ganze Zahlen als Namen eingesetzt.

Definition 4.1.1. Die Menge aller Namen wird mit $\mathrm{Na} = \mathbb{N}$ bezeichnet.

Desweiteren ist eine Grundmenge für Zeitangaben, wie z.B. für Bearbeitungszeiten oder für Taktzeiten, notwendig. Das gewählte Zeitraster ist diskret, einheitlich und interpretationsspezifisch. Das heißt, daß alle Zeitangaben der Problemstellung sich auf eine Einheit, beispielsweise nur auf Sekunden oder nur auf Monate, beziehen, und daß die gewählte Einheit im Gesamtkontext sinngemäß ist.

Definition 4.1.2. Die Menge aller Zeitangaben wird mit $\mathbb{Ti} = \mathbb{N}$ bezeichnet.

Gleiches gilt für Kosten. Sie sind ebenfalls diskret, einheitlich und interpretationsspezifisch.

Definition 4.1.3. Die Menge aller Kosten wird mit $\mathbb{K} = \mathbb{N}$ bezeichnet.

Ferner wird eine Prozentangabe benötigt. Sie soll einen Anteil bzw. eine Zuverlässigkeit wiedergeben. Dabei heißen 0% völlig unzuverläßig oder als Anteil nicht vorhanden. Dagegen stehen 100% für absolut zuverläßig bzw. für einen ganzen Anteil. Aufgrund der Beschränkung auf ganze Zahlen, sollen Prozentangaben ebenfalls diesen Typ erhalten.

Definition 4.1.4. Die Menge $\mathbb{Pr}$ ist die Menge aller ganzen Zahlen zwischen 0 und 100. Sie drückt eine Prozentangabe aus.

$$\mathbb{Pr} = \{0, 1, \ldots, 100\}$$

Die Basis des Problems bilden Aktivitäten. Sie kommen in zwei Datenstrukturen vor, den Ablaufplänen und den Teilnehmertypen. Aktivitäten werden durch Namen bezeichnet.

Definition 4.1.5 (Aktivitäten). Eine Menge von *Aktivitäten* $\mathbb{A}$ ist eine endliche Teilmenge von Na .

Das nächste, wichtige Konstrukt ist die Menge aller möglichen Workflow Teilnehmertypen. Ein Element w aus dieser Menge ist ein vierstelliges Tupel, welches Angaben über die prinzipiell möglichen Aktivitäten (A), die Kosten (k), die Rüstzeit (rz) und die Stabilität (s) macht. Die Bezeichnung *prinzipiell möglich* drückt die *Fähigkeiten* des Teilnehmertyps aus und gibt an, welche Aktivitäten eingesetzt werden können. Ob sie in einer Teilnehmerkonfiguration tatsächlich eingesetzt werden, steht nicht fest. Die Stabilität ist eine Prozentangabe.

Definition 4.1.6 (Workflow Teilnehmertyp). Die Menge aller *Workflow Teilnehmertypen* $\mathbb{WFT}$, ist wie folgt definiert:

$$\mathbb{WFT} = \mathcal{P}(\mathbb{A}) \times \mathbb{K} \times \mathbb{Ti} \times \mathbb{Pr}$$

Hierbei steht $\mathcal{P}(\mathbb{A})$ für die Potenzmenge von $\mathbb{A}$. Allgemein drücken wir die Potenzmenge einer beliebigen Menge M durch $\mathcal{P}(M)$ aus.

Sei $w = (A, k, rz, s) \in \mathbb{WFT}$ ein Workflow Teilnehmertyp. Folgende Projektionen sind auf einem Workflow Teilnehmertyp definiert:

$$\begin{array}{llll}
\mathit{akt}: & \mathbb{WFT} \to \mathcal{P}(\mathbb{A}) & \text{mit} & \mathit{akt}(w) = A \\
\mathit{kost}: & \mathbb{WFT} \to \mathbb{K} & \text{mit} & \mathit{kost}(w) = k \\
\mathit{rz}: & \mathbb{WFT} \to \mathbb{Ti} & \text{mit} & \mathit{rz}(w) = rz \\
\mathit{stab}: & \mathbb{WFT} \to \mathbb{Pr} & \text{mit} & \mathit{stab}(w) = s
\end{array}$$

4.1.2 Ablauf und Partition

Im folgenden wird ein Gerüst für eine Ablaufmenge angegeben. Ein Ablauf ist eine partielle Information einer Prozeßdefinition. Er gibt eine relative Reihenfolge der Aktivitäten vor, jedoch keine exakten Zeitpunkte. Ziel ist das Finden einer gültigen Teilnehmerkonfiguration, die für alle Abläufe passend ist. Später wird hinsichtlich jedes Ablaufs und jedes Blocks die Taktdauer, die notwendige Anzahl von Teilnehmern und der Teilnehmertyp ermittelt.

Definition 4.1.7 (Ablauf). Ein Ablauf $\varphi = (A_\varphi, R_\varphi)$ ist ein gerichteter, azyklischer und zusammenhängender Graph mit Knoten $A_\varphi \subseteq \mathbb{A}$ und Kanten $R_\varphi \subseteq A_\varphi \times A_\varphi$. Die Menge aller Abläufe wird mit Φ bezeichnet. In diesem Zusammenhang spricht man auch von Vorranggraphen.

Die Knoten eines Ablaufs φ denotieren Aktivitäten, während Kanten $(a, b) \in R_\varphi$ als Vorrangsrelationen interpretiert werden, was bedeutet, daß Aktivität a im Ablauf φ vor Aktivität b ausgeführt wird. Für $(a, b) \in R_\varphi$ schreiben wir auch $a \overset{\varphi}{\to} b$.

Folgende Projektionen, die einen Ablauf auf die Menge seiner Aktivitäten bzw. auf die Menge seiner Kanten abbilden, sind auf Ablaufmengen definiert:

$$\begin{array}{llll} \mathit{akt}: & \Phi \to \mathcal{P}(\mathbb{A}) & \text{mit} & \mathit{akt}(\varphi) = A_\varphi \\ \mathit{rel}: & \Phi \to \mathcal{P}(\mathbb{A} \times \mathbb{A}) & \text{mit} & \mathit{rel}(\varphi) = R_\varphi \end{array}$$

Bezüglich eines Ablaufs $\varphi = (A_\varphi, R_\varphi)$ wird die Vorgänger- bzw. Nachfolgerfunktion, die eine Aktivität auf ihre Vorgänger bzw. auf ihre Nachfolger abbildet, wie folgt definiert:

$$\begin{array}{llll} \mathit{pred}_\varphi: & A_\varphi \to \mathcal{P}(A_\varphi) & \text{mit} & \mathit{pred}_\varphi(a) = \{b \mid b \overset{\varphi}{\to} a\} \\ \mathit{succ}_\varphi: & A_\varphi \to \mathcal{P}(A_\varphi) & \text{mit} & \mathit{succ}_\varphi(a) = \{b \mid a \overset{\varphi}{\to} b\} \end{array}$$

Desweiteren wird festgelegt, daß ein Ablauf φ genau eine Startaktivität $s \in A_\varphi$ mit $|\,\mathit{pred}_\varphi(a)\,| = 0$ und genau eine Schlußaktivität $t \in A_\varphi$ mit $|\,\mathit{succ}_\varphi(t)\,| = 0$ (wobei $t \neq s$) enthält. Dabei bezeichnet $|M|$ die Kardinalität der Menge M. Für alle anderen Aktivitäten $a \in A_\varphi \setminus \{s, t\}$ gilt:

$$\begin{array}{rcl} |\,\mathit{pred}(a)\,| & > & 0 \\ |\,\mathit{succ}(a)\,| & > & 0 \end{array}$$

Zuletzt wird eine Funktion benötigt, die die Vereinigung aller Abläufe einer Ablaufmenge definiert. Dies macht die Funktion $\mathit{res}: \mathcal{P}(\Phi) \to \Phi$. Sei $\Phi_z \in \mathcal{P}(\Phi)$ eine konkrete Menge von Abläufen. Dann ist die Vereinigung $\mathit{res}(\Phi_z) = (A_{\text{res}}, R_{\text{res}})$, wobei

$$A_{\text{res}} = \bigcup_{\varphi \in \Phi_z} \mathit{akt}(\varphi) \qquad\qquad R_{\text{res}} = \bigcup_{\varphi \in \Phi_z} \mathit{rel}(\varphi).$$

Darauf hat ein Mechanismus zu folgen, der die in Kapitel 2.3 vorgestellte Partition ermöglicht. Eine solche Graphfärbung muß auf einem Ablauf definiert sein. Zuerst wird der allgemeine Begriff *Partition* definiert.

Definition 4.1.8 (Partition). Ein Tupel $S_P = (S_1, \dots, S_n)$ nichtleerer Mengen bildet eine *Partition* einer Menge S, wenn gilt:

- alle Mengen des Tupels S_P sind paarweise disjunkt:

$$\forall i, j \in \{1, 2, \dots, n\} : \quad S_i \cap S_j = \varnothing$$

- die Vereinigung aller Mengen des Tupels S_P ergibt S:

$$S = \bigcup_{i=1}^{n} S_i$$

Die Menge S_i einer Partition S_P wird auch als der *i-te Block* bezeichnet.

Damit ist es möglich, eine Partition eines Ablaufs φ auszudrücken. Hierzu wird über der Menge der Aktivitäten von φ partitioniert, wobei die Partition zusätzlich eine Eigenschaft erfüllen muß, die durch die Kanten von φ induziert wird. Diese Eigenschaft ist das Mittel, das für den gerichtetet Datenfluß verantwortlich ist (siehe Abschnitt 2.3.1).

Definition 4.1.9. Ein Tupel $S_P = (S_1, \dots, S_n)$ ist eine Partition von einem Ablauf $\varphi = (A_\varphi, R_\varphi)$, wenn gilt:

1. S_P ist eine Partition von A_φ
2. für alle Kanten $a \overset{\varphi}{\rightarrow} b$ aus R_φ gilt:

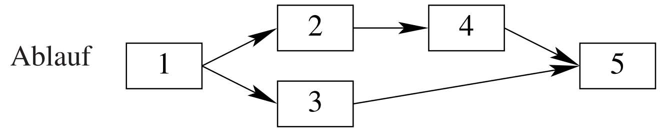
$$a \in S_i, b \in S_j \quad \text{mit } i \leq j \tag{4.1}$$

Übertragen auf das Workflow Problem bedeutet eine Partition $S_P = (S_1, \dots, S_n)$ eines Ablaufs φ, daß jeder Menge S_i ein Teilnehmertyp W_i zugeordnet wird. Ein Teilnehmer des Blocks i hat die Aufgabe, alle Aktivitäten aus S_i an einem Objekt durchzuführen. Das Objekt darf erst nach der Abarbeitung aller Aktivitäten freigegeben werden. Die Weitergabe erfolgt nur an einen Teilnehmer W_j, der für einen späteren Block S_j mit $j > i$ zuständig ist. Dieser Vorgang wird auch *Pipelining* genannt.

Definition 4.1.10 (Sequenz). Ein Tupel $A_S = (a_1, \dots, a_m)$ von m Aktivitäten bildet eine *Sequenz* eines Ablaufs φ mit Aktivitäten A_φ, wenn gilt:

- alle Aktivitäten a_i, a_j mit $i \neq j$ sind paarweise verschieden
- die Vereinigung aller Aktivitäten in A_S ist A_φ:

$$A_\varphi = \bigcup_{i=1}^{m} a_i$$



Partition	({1},	{2, 3},	{4},	{5})
Sequenz a)	(1,	2, 3,	4,	5)
b)	(1,	3, 2,	4,	5)

Abbildung 4.1: Für jeden Ablauf existiert mindestens eine Partition. Für jede Partition existiert mindestens eine Sequenz.

Die Sequenz ist die Bearbeitungsreihenfolge von Aktivitäten. Eine Sequenz A_S ist zu einem partitionierten Ablauf $S_P = (S_1, \ldots, S_n)$ *passend*, wenn A_S so in n Teilsequenzen aufgeteilt wird, daß die Aktivitäten der Teilsequenz $(A_S)_i$ genau den Aktivitäten der Menge S_i entsprechen. Das folgende Beispiel und Abbildung 4.1 verdeutlichen den Zusammenhang zwischen Ablauf, Partition und Sequenz.

Beispiel 4.1.11. Sei $\varphi = (A = \{1, 2, 3, 4, 5\}, R = \{(1, 2), (1, 3), (2, 4), (3, 5), (4, 5)\})$ ein Ablauf. Eine mögliche Partition ist beispielsweise $(\{1\}, \{2, 3\}, \{4\}, \{5\})$. Ein anderes Beispiel ist $(\{1\}, \{2\}, \{3, 4\}, \{5\})$. Beide Partitionen erfüllen Eigenschaft (4.1).

Beim letzteren Beispiel kann der Workflow zwei passende Sequenzen zur Abarbeitung wählen. Entweder $(1, 2, 3, 4, 5)$ oder $(1, 2, 4, 3, 5)$. Die Reihenfolge innerhalb einer Teilsequenz ist unwesentlich.

Eine ungültige Partition ist $(\{1\}, \{4, 3\}, \{2\}, \{5\})$. Hier befindet sich die Aktivität 4 in der zweiten Menge und Aktivität 2 in der dritten Menge. Dies würde den Workflow veranlassen, Aktivität 4 vor Aktivität 2 auszuführen. Dies steht jedoch im Widerspruch zur Kante $2 \xrightarrow{\varphi} 4$.

Auch die Ausführung der Sequenz $(1, 3, 2, 4, 5)$ ist nicht erlaubt, da nach Ausführung der Aktivität 3 mit dem dritten Block fortgefahren wird, um im Anschluß wieder die Arbeit im zweiten Block aufzunehmen.

Hinweis. In diesem Abschnitt wurden lediglich die Eigenschaften einer Partition dargestellt. Auf die Konstruktion einer Partition wird in den Kapiteln 5 und 6 näher eingegangen.

4.1.3 Zeitangaben

Es werden eine Reihe von Zeitangaben verwendet. Eine davon wurde schon im Kapitel 4.1.1 vorgestellt, die Rüstzeit. Eine neue Zeitangabe ist die konstante, *maximale Taktzeit*. Sie gibt das maximale Intervall vor, in dem neue Prozesse gestartet bzw. instantiiert werden.

Definition 4.1.12 (Maximale Taktzeit). Die *maximale Taktzeit* wird durch die Konstante $mT \in \mathbb{T}\hat{\imath}$ angegeben.

Die Anzahl der Workflow Teilnehmer ist eine obere Grenze für die Anzahl der maximalen Objekte im Workflow. Allerdings gilt das nur für optimal ausbalancierte Konfigurationen. In der Regel arbeiten die Teilnehmerstationen unterschiedlich schnell, so daß Wartezeiten an langsamer arbeitenden Stationen entstehen. Um Staus zu vermeiden, werden Objekte in Pufferzonen geparkt. In diesen Fällen können mehr Objekte als Teilnehmer im Workflow enthalten sein.

Eine Verminderung der Taktzeit in einem Block erhält man, indem die Zahl der Teilnehmer in diesem Block erhöht wird. Eine gleichmäßige Verminderung der Taktzeit aller Blöcke führt zu einer erhöhten Kapazität von Objekten im gesamten Workflow.

Eine weitere konstante Zeitangabe ist die globale Übergangszeit. Diese definiert die Zeit, die zwischen zwei Aktivitäten anfällt, wenn ein Wechsel von einer Aktivität zur nächsten stattfindet. Beide Aktivitäten müssen in einem gemeinsamen Block liegen. Eine Übergangszeit von der letzten Aktivität eines Blocks auf die erste Aktivität des nächsten Blocks gibt es nicht. Stattdessen fällt hier die Rüstzeit ins Gewicht. Das Bild 2.3 auf Seite 22 veranschaulicht die zeitlichen Zusammenhänge.

Definition 4.1.13 (Globale Übergangszeit). Die *globale Übergangszeit* wird durch die Konstante $guz \in \mathbb{T}\mathbb{i}$ bezeichnet.

Die wichtigsten Zeitangaben sind die Bearbeitungszeiten der Aktivitäten. Im Grundmodell wird die *globale Bearbeitungszeit* angewendet. Global heißt in diesem Zusammenhang, daß die Bearbeitungszeit einer Aktivität unabhängig vom Teilnehmertyp ist, d.h. die Bearbeitung einer Aktivität dauert auf allen Teilnehmertypen gleichlang.

Definition 4.1.14 (Globale Bearbeitungszeit). Die Funktion $bz : \mathbb{A} \to \mathbb{T}\mathbb{i}$ gibt zu einer Aktivität a die globale Bearbeitungszeit $bz(a)$ an.

Für das weitere Verfahren ist es wichtig, die Gesamtbearbeitungszeit pro Block (*Blockbearbeitungszeit*) zu kennen. Diese setzt sich aus den Bearbeitungszeiten der Aktivitäten in diesem Block, deren Übergangszeiten und der Rüstzeit des Blocks zusammen. Die Rüstzeit hängt vom eingesetzten Teilnehmertyp ab, während die Aktivitäten von einer gegebenen Partition und von einem Ablauf abhängig sind. Es wird zunächst eine einfache Definition der Blockbearbeitungszeit angegeben.

Definition 4.1.15 (Blockbearbeitungszeit). Sei $A \subseteq \mathbb{A}$ eine Menge von Aktivitäten, $w \in \mathbb{WFT}$ ein Teilnehmertyp, $guz \in \mathbb{T}\mathbb{i}$ eine konstante Übergangszeit. Dann beschreibt die Funktion $bbz : \mathcal{P}(\mathbb{A}) \times \mathbb{WFT} \to \mathbb{T}\mathbb{i}$ die Bearbeitungszeit aller Aktivitäten in A für einen Teilnehmer w wie folgt:

$$bbz(A, w) = \begin{cases} \left(\sum_{a \in A} bz(a)\right) + (rz(w) + (|A| - 1) * guz) & \text{, falls } |A| \geq 1 \\ 0 & \text{, sonst} \end{cases}$$

Eine Blockbearbeitungszeit fällt nur dann an, wenn Aktivitäten im Block A enthalten sind.

Da die Blockbearbeitungszeit bezüglich eines Ablaufs ausgedrückt wird, muß die Definition von *bbz* auf Abläufe erweitert werden.

Definition 4.1.16. Sei $\varphi \in \Phi$ ein Ablauf, A eine Menge von Aktivitäten und $w \in \mathbb{WFT}$ ein Teilnehmertyp. Dann beschreibt die Funktion $bbz : \Phi \times \mathcal{P}(\mathbb{A}) \times \mathbb{WFT} \to \mathbb{T}\mathbb{i}$ den Zeitanteil des Ablaufs φ in der Menge A hinsichtlich des eingesetzten Teilnehmertyps w.[1]

Sei $A' = A \cap akt(\varphi)$ die Menge der Aktivitäten des Ablaufs φ in der Menge A. Dann ist

$$bbz(\varphi, A, w) = bbz(A', w).$$

Eine Trennung der Blockbearbeitungszeit nach Abläufen ist sinnvoll, da bessere Abschätzungen für die Anzahl von Teilnehmern erzielt werden. Eine grobe Abschätzung würde die Bearbeitungszeiten aller Aktivitäten in einem Block heranziehen. Dabei wird nicht in Betracht gezogen, ob ein Ablauf alle Aktivitäten des Blocks beinhaltet. Beispiel 4.1.17 geht auf diesen Sachverhalt ein.

Beispiel 4.1.17. Seien

$$\begin{aligned} \varphi_1 &= (A_1 = \{1,2,3,4,5\}, R_1 = \{(1,2),(1,3),(2,4),(3,5),(4,5)\} \\ \varphi_2 &= (A_2 = \{1,2,6,5\}, R_2 = \{(1,2),(1,6),(2,4),(2,5),(6,5)\} \end{aligned}$$

zwei Abläufe. Diese werden in $\varphi_r = res(\{\varphi_1, \varphi_2\})$ vereint. Nun sei S_p eine Partition über $akt(\varphi_r)$, bestehend aus vier Blöcken. Eine passende Partition ist zum Beispiel das Tupel $(\{1\}, \{2,6\}, \{3,4\}, \{5\})$. Werden nun alle Bearbeitungszeiten auf 10 gesetzt, die globale Übergangszeit auf 1 und die Rüstzeit auf 50, so ergeben sich folgende Blockbearbeitungszeiten:

Block	φ_1	φ_2
1	$50 + 10 = 60$	$50 + 10 = 60$
2	$50 + 10 = 60$	$50 + 10 = 60$
3	$50 + 10 + 1 + 10 = 71$	0
4	$50 + 10 = 60$	$50 + 10 = 60$

Der erste Block enthält eine Aktivität, die von beiden Abläufen benötigt wird. Im zweiten Block befindet sich jeweils nur eine Aktivität, die einem Ablauf zugeordnet werden kann. Daher sind die Blockbearbeitungszeiten gleich. Würde man keine Unterscheidung nach Abläufen machen, so würde sich eine Blockbearbeitungszeit von 71 ergeben, was zu mehr, nicht notwendigen Teilnehmern führen kann. Der dritte Block ist ein Beispiel dafür, daß ein Ablauf überhaupt nicht in einem Block vertreten sein muß. Schließlich gilt für den vierten Block das gleiche wie für den ersten.

4.1.4 Workflow Teilnehmer

Dieser Abschnitt befaßt sich mit den Workflow Teilnehmertypen. Es soll herausgefunden werden, welcher Teilnehmertyp in welchem Block und in welcher Anzahl eingesetzt werden soll.

[1] In dieser Arbeit werden Überladungen von Projektionen, Funktionen und Relationen erlaubt.

Der Teilnehmertyp hängt von der zugrundeliegenden Aktivitätsmenge ab. Er muß diejenigen Aktivitäten prinzipiell bereit stellen können, die in der Menge enthalten sind.

Beispiel 4.1.18. Sei $S_1 = \{1\}$ eine Menge von Aktivitäten und seien $w_1, w_2 \in \mathbb{WFT}$ zwei Workflow Teilnehmertypen, wobei A_1 die Menge der Fähigkeiten von w_1 mit $akt(w_1) = \{1, 2, 3\}$ und A_2 die der von w_2 mit $akt(w_2) = \{2, 3, 4\}$ ist. Dann kann nur w_1 als Teilnehmer für die Menge S_1 eingesetzt werden. Für die Menge $S_2 = \{2, 3\}$ sind beide Teilnehmertypen möglich. Zu der Menge $S_3 = \{1, 4\}$ paßt kein Teilnehmertyp, da $1 \notin A_2$ und $4 \notin A_1$.

Eine Erweiterung dieser Betrachtung von einer Aktivitätsmenge auf eine Partition führt zum Begriff *Teilnehmertypkonfiguration*.

Definition 4.1.19 (Teilnehmertypkonfiguration). Eine *Teilnehmertypkonfiguration* Ws bestehend aus $n \in \mathbb{N}$ Elementen ist ein Tupel der Form $Ws \in \mathbb{WFT}^n$.

Eine Teilnehmertypkonfiguration Ws ist zu einer Partition $S_P = (S_1, S_2, \ldots, S_n)$ passend, falls gilt:

$$\forall i \in \{1, 2, \ldots, n\} \; : \; S_i \subseteq akt(Ws_i) \tag{4.2}$$

Sowohl die Partition als auch die Teilnehmertypkonfiguration liegen zunächst in unbestimmter Form vor. Eigenschaft (4.2) schränkt die möglichen Kandidaten für jeden Block ein. Je mehr Informationen über eine Partition vorliegen, desto genauer ist die Eingrenzung der Kandidaten. Umgekehrt führt eine Festlegung auf eine konkrete (auch partielle) Konfiguration zu einer Präzisierung der Partition. Abschnitte 5.2 und 6.1ff. gehen näher auf diesen Sachverhalt ein.

Ist ein konkreter Teilnehmertyp für einen Block bekannt, folgt die Berechnung der Anzahl von Teilnehmern. In die Berechnung geht die Blockbearbeitungszeit des Blocks, die Stabilität des Teilnehmertyps, die maximale Taktzeit und der mögliche Verlust im gesamten Netzwerk ein.

Der Verlust ist eine Prozentangabe und beschreibt, inwiefern weitere Fehlerquellen Einfluß auf das Objekt nehmen können. Hierzu gehören insbesondere Transportfehler, die beim Weiterreichen des Objektes anfallen können.

Definition 4.1.20 (Verlust). Der *Verlust* wird durch die Konstante $v \in \mathbb{Pr}$ angegeben.

Damit wird die Anzahl der Teilnehmer für eine beliebige Menge von Aktivitäten respektive eines Ablaufs berechnet. Die Betrachtungsweise verläuft analog zur Blockbearbeitungszeit in Abschnitt 4.1.3.

Definition 4.1.21 (Anzahl Teilnehmer). Sei $\varphi \in \Phi$ ein Ablauf, $A \in \mathcal{P}(\mathbb{A})$ eine Menge von Aktivitäten, $w \in \mathbb{WFT}$ ein Teilnehmertyp, so daß die Bedingung $A \subseteq akt(w)$ erfüllt ist. Desweiteren sei $\mathrm{mT} \in \mathbb{Ti}$ die konstante, maximale Taktzeit, $v \in \mathbb{Pr}$ der konstante Verlust. Dann wird die Anzahl benötigter Teilnehmer bezüglich eines Ablaufs φ und einer Aktivitätsmenge A durch die Funktion $anz : \Phi \times \mathcal{P}(\mathbb{A}) \times \mathbb{WFT} \to \mathbb{N}$ angegeben. Die Abbildungsvorschrift lautet:

$$anz(\varphi, p, w) = \left\lceil \frac{bbz(\varphi, w, p) * (100 + v)}{stab(w) * \mathrm{mT}} \right\rceil$$

Aus der Formel sind mehrere Abhängigkeiten ablesbar. Zum einen erhöht sich der Bedarf von Teilnehmern, je geringer die Stabilität des Typs ausfällt. Zum anderen wächst der Bedarf, je größer der allgemeine Verlust geschätzt wird. Schließlich ist mehr Bedarf vorhanden, falls die maximale Taktzeit vermindert wird.

4.1.4.1 Anzahldominanz

Im Grundmodell werden Lösungen für separate Datenflüsse (siehe Abschnitt 2.2.2) betrachtet. Dies bedeutet, daß zu jedem Zeitpunkt maximal ein Ablauf bearbeitet wird. Alle anderen Abläufe ruhen. Dennoch muß die Lösung eine Infrastruktur enthalten, die alle geforderten Abläufe ausführen kann. Dies führt dazu, daß die Anzahl der Teilnehmer bezüglich einer Menge von Aktivitäten für jeden Ablauf getrennt berechnet wird. Verlangt ein Ablauf φ_1 für eine Menge von Aktivitäten fünf Teilnehmer, der Ablauf φ_2 vier, muß die gesuchte Lösung mindestens fünf Teilnehmer enthalten, da eine Lösung mit vier Teilnehmern die Bedingungen des Ablaufs φ_1 nicht erfüllen kann. Demnach wird der Ablauf gesucht, der die meisten Teilnehmer hinsichtlich einer Menge von Aktivitäten verlangt. Dieser Ablauf ist *anzahldominant*. Ein Block kann auch mehrere anzahldominante Abläufe haben, die die gleiche Anzahl von Teilnehmern verlangen.

Definition 4.1.22 (Anzahldominanz). Sei $Q \in \mathcal{P}(\Phi)$ eine Menge von Abläufen, $w \in \mathbb{WFT}$ ein Workflow Teilnehmer und $A \in \mathcal{P}(\mathbb{A})$ eine Menge von Aktivitäten. Ein Ablauf $\varphi \in Q$ ist *anzahldominant*, wenn gilt

$$\forall \varphi_i \in Q, \varphi \neq \varphi_i : \quad \mathit{anz}(\varphi, A, w) \geq \mathit{anz}(\varphi_i, A, w)$$

Die Anwendung der Anzahldominanz ist für die Bestimmung der Kosten notwendig (siehe Abschnitt 4.1.6 und 4.1.7).

4.1.4.2 Zeitfenster

In diesem Abschnitt werden verschiedene Zeitfenster vorgestellt, die zur Beschreibung weiterer Eigenschaften des Workflow Problems dienen. Insbesondere kann eine untere Schranke für die Anzahl von Teilnehmern angegeben werden (siehe Abschnitt 5.4).

Die Zeitspanne, die zwischen dem Zeitpunkt eines neu eingesetzten Teilnehmers und dem Zeitpunkt des nächst notwendigen Teilnehmers anfällt, wird als *absolutes Zeitfenster* bezeichnet. Das absolute Zeitfenster hängt von der maximalen Taktzeit, dem Verlust und der Stabilität des jeweilig eingesetzten Teilnehmers ab. Die Größe eines absoluten Zeitfensters ist aus Definition 4.1.21 ableitbar. Abbildung 4.2 veranschaulicht die nachfolgenden Definitionen.

Definition 4.1.23 (Absolutes Zeitfenster). Sei $\mathrm{mT} \in \mathbb{Ti}$ die maximale Taktzeit, $v \in \mathbb{Pr}$ der Verlust und $\mathrm{st} \in \mathbb{Pr}$ eine Prozentangabe, die hier als die Stabilität eines Teilnehmertyps interpretiert wird. Dann gibt die Funktion $\mathit{absZ} : \mathbb{Ti} \times \mathbb{Pr} \times \mathbb{Pr} \rightarrow \mathbb{Ti}$ das *absolute Zeitfenster* an. Die Abbildungsvorschrift lautet

$$\mathit{absZ}(\mathrm{mT}, \mathrm{st}, v) = \left\lceil \frac{\mathrm{st} * \mathrm{mT}}{100 + v} \right\rceil .$$

Je länger die maximale Taktzeit und je größer die Stabilität, desto größer ist das absolute Zeitfenster. Im Gegensatz dazu verringert sich das absolute Zeitfenster, je größer der Verlust wird.

Das *relative Zeitfenster* hingegen ist der Zeitbereich zwischen dem Ende der Blockbearbeitungszeit und dem Zeitpunkt, an dem ein weiterer Teilnehmer notwendig wird. Schließlich ist noch die *Restbearbeitungszeit* von Interesse. Sie gibt an, wieviel Zeit im letzten Zeitfenster von der Blockbearbeitungszeit belegt wurde.

Definition 4.1.24 (Relatives Zeitfenster/Restbearbeitungszeit). Sei $\mathrm{t}_{\mathrm{Gbz}} \in \mathbb{T}\mathbb{i}$ eine Blockbearbeitungszeit und $\mathrm{t}_{\mathrm{az}} \in \mathbb{T}\mathbb{i}$ ein absolutes Zeitfenster. Dann berechnet die Funktion *restZ* : $\mathbb{T}\mathbb{i} \times \mathbb{T}\mathbb{i} \rightarrow \mathbb{T}\mathbb{i}$ die *Restbearbeitungszeit*, wobei

$$\mathit{restZ}(\mathrm{t}_{\mathrm{Gbz}}, \mathrm{t}_{\mathrm{az}}) = \mathrm{t}_{\mathrm{Gbz}} \bmod \mathrm{t}_{\mathrm{az}}.$$

Die Funktion *relZ* : $\mathbb{T}\mathbb{i} \times \mathbb{T}\mathbb{i} \rightarrow \mathbb{T}\mathbb{i}$ ergibt das *relative Zeitfenster* mit der Abbildungsvorschrift

$$\mathit{relZ}(\mathrm{t}_{\mathrm{Gbz}}, \mathrm{t}_{\mathrm{az}}) = \mathrm{t}_{\mathrm{az}} - \mathit{restZ}(\mathrm{t}_{\mathrm{Gbz}}, \mathrm{t}_{\mathrm{az}})$$

Das nachfolgende Beispiel und Abbildung 4.2 verdeutlichen den Zusammenhang der Fenster. Es sind zwei beliebige Mengen gegeben, deren Blockbearbeitungszeit der schraffierten Fläche entspricht. Beide Mengen benötigen drei Teilnehmer. Der Teilnehmertyp der ersten Menge besitzt eine höhere Stabilität, was sich in einem größeren, absoluten Zeitfenster äußert. Der Vorteil eines größeren Zeitfensters liegt darin, daß wesentlich mehr Bearbeitungszeit und damit eventuell mehr Aktivitäten aufgenommen werden können, ohne die Teilnehmersummen zu erhöhen.

Beispiel 4.1.25. Sei $\mathrm{mT} = 540$, $\nu = 5$ und habe Teilnehmertyp w_1 eine Stabilität von 95 und Teilnehmertyp w_2 eine Stabilität von 80. Dann hat der erste Typ ein absolutes Zeitfenster von 488 Zeiteinheiten und der Letzte eines von 411. Existiert nun im ersten Block eine Blockbearbeitungszeit $\mathrm{t}_1 = 1012$, beträgt das relative Zeitfenster $1464 - 1012 = 452$ Zeiteinheiten. Im zweiten Block sind es $1233 - 1150 = 83$. Die Restbearbeitungszeit des ersten Blocks ist $1012 - 976 = 36$ und die des zweiten $1150 - 822 = 328$.

4.1.5 Taktzeit

Nach Ermittlung der Teilnehmerzahl in einem Block, wird die Taktzeit für diesen Block ermittelt. Die Taktzeit wird immer kleiner oder gleich der maximalen Taktzeit ausfallen. Dies garantiert die Abbildungsvorschrift von *anz*.

Es ist wichtig zu erfahren, wie groß die Taktzeiten für alle Mengen einer Partition unter Beachtung aller Abläufe sind. Die längste dieser ermittelten Taktzeiten entspricht der *effektiven Taktzeit*. Es kann durchaus zu Lösungen kommen, die eine wesentlich kleinere, effektive Taktzeit aufweisen als die geforderte, maximale Taktzeit. Dies ist ein Kriterium für die Taktoptimierung des Problems. Werden zwei Lösungen mit gleichen Kosten gefunden, wird die mit der kürzeren, effektiven Taktzeit favorisiert.

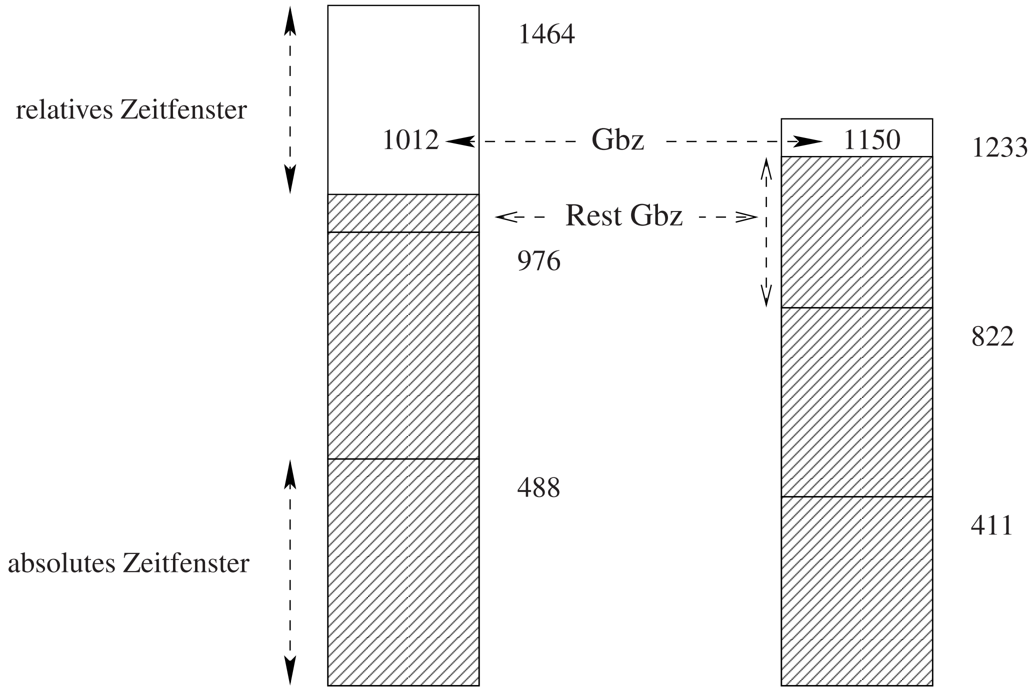


Abbildung 4.2: Zeitfenster: 95% Stabilität (links), 80% Stabilität (rechts)

Man kann das Problem auch verändern und nur noch die Minimierung der effektiven Taktzeit fordern. Dies würde jedoch nicht mehr die Kosten minimieren, da weitere Teilnehmer notwendig werden.

Definition 4.1.26 (Taktzeit). Sei $\varphi \in \Phi$ ein Ablauf, $A \in \mathcal{P}(\mathbb{A})$ eine beliebige Aktivitätsmenge und sei $w \in \mathbb{WFT}$ ein Teilnehmertyp, der die Bedingung $A \subseteq akt(w)$ erfüllt. Sei $m \in \mathbb{N}$ die Anzahl der benötigten Teilnehmer für die Menge A und $v \in \mathbb{Pr}$ der konstante Verlust. Dann definiert die Funktion *takt* : $\Phi \times \mathcal{P}(\mathbb{A}) \times \mathbb{WFT} \times \mathbb{N} \rightarrow \mathbb{N}$ die *Taktzeit*. Die Abbildungsvorschrift lautet:

$$takt(\varphi, p, w, m) = \left\lceil \frac{bbz(\varphi, p, w) * (100 + v)}{stab(w) * m} \right\rceil$$

Man erkennt die Verknüpfung der Taktzeit mit der Anzahl von Teilnehmern. Sie ist in der Formulierung des Problems als ein Constraint Problem hilfreich, da weitere einschränkende Informationen dem System hinzugefügt werden. Dies ist selbst dann der Fall, wenn die effektive Taktzeit unbeachtet bleibt.

4.1.5.1 Taktdominanz

Analog zur Anzahldominanz wird der Begriff Taktdominanz eingeführt. Ein Ablauf ist bezüglich einer Menge von Aktivitäten taktdominant, wenn er die längste Taktdauer beansprucht.

Definition 4.1.27 (Taktdominanz). Sei $Q \in \mathcal{P}(\Phi)$ eine Menge von Abläufen, $w \in \mathbb{WFT}$ ein Workflow Teilnehmer und $A \in \mathcal{P}(\mathbb{A})$ eine Menge von Aktivitäten. Ein Ablauf $\varphi \in Q$ ist *taktdominant*, wenn gilt

$$\begin{aligned} \forall \varphi_i \in Q, \varphi \neq \varphi_i : \quad & \mathit{takt}(\varphi, A, w, m) \geq \mathit{takt}(\varphi_i, A, w, m_i) \\ \text{mit} \quad m \; & = \; \mathit{anz}(\varphi, A, w) \\ m_i \; & = \; \mathit{anz}(\varphi_i, A, w) \end{aligned}$$

Man muß beachten, daß ein anzahldominanter Ablauf nicht gleichzeitig taktdominant sein muß. Dies gilt auch umgekehrt.

4.1.6 Kosten

Ziel des Problems ist, die günstigste Lösung zu finden. Die Kosten ergeben sich aus der Wahl der Teilnehmertypen und deren eingesetzten Anzahl. Beiden Größen liegt eine Menge von Aktivitäten zugrunde. Zusätzlich hängen die Kosten von Abläufen ab.

Es soll daran erinnert werden, daß die Anzahl der Teilnehmer pro Ablauf bestimmt wurde. Dies hat Einfluß auf die Kosten. Die Anzahl der Teilnehmer entspricht der Anzahl der Teilnehmer der anzahldominanten Abläufe.

Definition 4.1.28 (Kosten). Sei $Q \in \mathcal{P}(\Phi)$ eine Menge von Abläufen. Desweiteren sei $A \in \mathcal{P}(\mathbb{A})$ eine beliebige Aktivitätenmenge und $w \in \mathbb{WFT}$ der eingesetzte Teilnehmertyp. Außerdem sei $\varphi \in Q$ anzahldominant gegenüber A. Dann gibt die Funktion $\mathit{kost} : \mathcal{P}(\Phi) \times \mathcal{P}(\mathbb{A}) \times \mathbb{WFT} \to \mathbb{K}$ die Kosten in Abhängigkeit von einem Ablauf an. Die Abbildung lautet:

$$\mathit{kost}(Q, A, w) = \mathit{kost}(w) * \mathit{anz}(\varphi, A, w)$$

4.1.7 Problemstellung und Lösung

In diesem Abschnitt wird eine Formalisierung der Problemstellung angegeben und ein skizzierter Lösungsweg aufgezeigt, indem dargestellt wird, wie eine Lösung aussieht und wie sie aus dem bisher gezeigten hergeleitet wird.

4.1.7.1 Definition von Problem und Lösung

Definition 4.1.29 (Problemstellung). Die Menge von Problemstellungen der Workflow Ressourcenoptimierung Γ hat folgende Definition:

$$\Gamma = \mathcal{P}(\Phi) \times \mathcal{P}(\mathbb{WFT}) \times \mathbb{Ti} \times \mathbb{Pr} \times \mathbb{Ti}$$

Ein konkretes Problem ist ein Tupel der Form $\gamma = (Q, W, \mathrm{mT}, \nu, guz) \in \Gamma$, wobei $Q \in \mathcal{P}(\Phi)$ die Menge der Ablaufpläne, $W \in \mathcal{P}(\mathbb{WFT})$ die Menge der vorhandenen Workflow Teilnehmertypen, mT die geforderte, maximale Taktzeit, ν der allgemeine

Verlust und guz die globale Übergangszeit ist. Folgende Projektionen sind auf Γ definiert:

$$\begin{array}{lll} abl: & \Gamma \to \mathcal{P}(\Phi) & \text{mit} \quad abl(\gamma) = Q \\ wft: & \Gamma \to \mathcal{P}(\mathbb{WFT}) & \text{mit} \quad wft(\gamma) = W \\ takt: & \Gamma \to \mathbb{T}\mathbb{i} & \text{mit} \quad takt(\gamma) = \text{mT} \\ ver: & \Gamma \to \mathbb{P}\mathbb{r} & \text{mit} \quad ver(\gamma) = \nu \\ guz: & \Gamma \to \mathbb{T}\mathbb{i} & \text{mit} \quad guz(\gamma) = guz \end{array}$$

Eine Lösung besteht aus folgenden Informationen: Anzahl Blöcke (n), Partition (S), Teilnehmertypkonfiguration (Ws), Anzahl der Teilnehmer pro Block (Ms), effektive Taktzeit (eT) und Gesamtkosten (K_G).

Definition 4.1.30 (Lösung). Die Menge aller *Lösungen* $\mathcal{L}$ ist wie folgt definiert:

$$\mathcal{L} = \mathbb{N} \times (\mathcal{P}(\mathbb{A}))^{\mathbb{N}} \times \mathbb{WFT}^{\mathbb{N}} \times \mathbb{N}^{\mathbb{N}} \times \mathbb{T}\mathbb{i} \times \mathbb{K}$$

Sei $(n, S, Ws, Ms, \text{eT}, K_G) \in \mathcal{L}$ eine Lösung. Folgende Projektionen sind auf $\mathcal{L}$ definiert:

$$\begin{array}{lll} bloecke: & \mathcal{L} \to \mathbb{N} & \text{mit} \quad bloecke(L) = n \\ part: & \mathcal{L} \to (\mathcal{P}(\mathbb{A}))^{\mathbb{N}} & \text{mit} \quad part(L) = S \\ wft: & \mathcal{L} \to \mathbb{WFT}^{\mathbb{N}} & \text{mit} \quad wft(L) = Ws \\ anz: & \mathcal{L} \to \mathbb{N}^{\mathbb{N}} & \text{mit} \quad anz(L) = Ms \\ takt: & \mathcal{L} \to \mathbb{T}\mathbb{i} & \text{mit} \quad takt(L) = \text{eT} \\ kost: & \mathcal{L} \to \mathbb{K} & \text{mit} \quad kost(L) = K_G \end{array}$$

Um die Eigenschaften aller Lösungen einer bestimmten Problemstellung beschreiben zu können, bedarf es einer Relation, die zu einer Problemstellung alle Lösungen angibt.

Definition 4.1.31. Sei $\gamma \in \Gamma$ eine beliebige Problemstellung. Dann gibt die Relation $loes : \Gamma \to \mathcal{P}(\mathcal{L})$ eine Menge von Lösungen zum Problem γ an. Jede Lösung dieser Menge erfüllt alle Constraints in γ.

4.1.7.2 Lösungsweg

In diesem Abschnitt wird ein Lösungsweg skizziert, wobei $(Q, W, \text{mT}, \nu, guz) \in \Gamma$ eine beliebige Problemstellung sei. Man beachte, daß die Partition und Teilnehmertypkonfiguration in der Regel vor der Suche nicht-determiniert sind. Das ist erst nach endlich vielen Suchschritten der Fall. Die nachfolgende Skizze geht von determinierten Strukturen aus.

Zuerst ist festzulegen, welche Aktivitäten beteiligt sind.

$$A_{\text{Res}} = akt(res(Q))$$

Danach wird mit $n \in \mathbb{N}$ die Anzahl von Blöcken so gewählt, daß eine Partition von A_{Res} folgende Form hat: $S = (S_1, S_2, \ldots, S_n)$. Desweiteren ist die Teilnehmertypkonfiguration Ws ein n-Tupel $(Ws_1, Ws_2, \ldots, Ws_n)$. Für die nachfolgenden Berechnungen wird angenommen, daß beide Angaben bekannt sind.

Daraus ergibt sich die Anzahl der Teilnehmer für jeden Block, die mit Ms bezeichnet werden.

$$\forall i \in \{1, 2, \ldots, n\}: \quad \varphi \in Q \text{ ist anzahldominant bzgl. } S_i \\ \Downarrow \\ Ms_i = \mathit{anz}(\varphi, S_i, Ws_i)$$

Aus diesen Angaben läßt sich die effektive Taktzeit $\text{eT} \in \mathbb{T}\mathbb{i}$ ermitteln. Dazu wird die größte Taktzeit pro Block benötigt, die mit Hilfe der Taktdominanz ausgedrückt wird. Aus diesen Angaben wird wiederum das Maximum gewählt, das die effektive Taktzeit ist.

$$\text{eT} = \max \left\{ t \;\middle|\; \begin{array}{ccc} i \in \{1, 2, \ldots, n\} & \wedge & \exists \varphi \in Q : \varphi \text{ ist taktdominant bzgl. } S_i \\ & & \Downarrow \\ & & t = \mathit{takt}(\varphi, S_i, Ws_i, Ms_i) \end{array} \right\}$$

Letztendlich ist es einfach, die Gesamtkosten $K_G \in K$ zu bestimmen:

$$K_G = \sum_{i=1}^{n} (\mathit{kost}(Ws_i) * Ms_i)$$

Alles zusammengenommen ergibt, daß ein Tupel

$$(n, S, Ws, Ms, \text{eT}, K_G) \in \mathcal{L}$$

eine Lösung darstellt.

4.2 Präzises Modell

Im präzisen Modell wird die Granularität des Problems verfeinert. Dazu werden Werkzeuge und lokale Bearbeitungszeiten eingeführt. Beide Angaben geben genauere Informationen über die Problemstellung. Es stellt sich die Frage, ob die zusätzlichen Informationen zu einem besseren Filterungsverhalten führen, oder ob sie das Problem schwieriger gestalten. Desweiteren wird der separate Datenfluß durch einen vermischten ersetzt. Es muß geklärt werden, ob dadurch eine bessere Ausbalancierung der Taktzeiten erreicht wird, bzw. ob die Kosten gesenkt werden.

4.2.1 Werkzeuge

Mit Hilfe von Werkzeugen ist eine weitere Einschränkung des Lösungsraums möglich. Dies ist auf folgende Punkte zurückzuführen: Erweiterung der Teilnehmertypen um ein Werkzeugmagazin, Werkzeugteilung durch Aktivitäten und Werkzeugkosten. Zunächst wird die Menge der Werkzeuge eingeführt.

Definition 4.2.1. Die Menge aller Werkzeuge wird mit $\mathbb{W}\mathbb{z} = \text{Na}$ bezeichnet.

Eine Aktivität wird durch eine Menge von Werkzeugen charakterisiert.

Definition 4.2.2 (Werkzeuge einer Aktivität). Sei $a \in \mathbb{A}$ eine Aktivität. Eine Funktion $tool : \mathbb{A} \to \mathcal{P}(\mathbb{Wz})$ gibt zu einer Aktivität a die notwendigen Werkzeuge an.

Es ist möglich und sehr nützlich, daß zwei Aktivitäten ein Werkzeug teilen. Dadurch wird Platz im Magazin des Teilnehmers gespart und vermindert gleichzeitig die Kosten für Werkzeuge. Eine gemeinsame Werkzeugnutzung zweier Aktivitäten $a_1, a_2 \in \mathbb{A}$ liegt vor, wenn beide in einer Menge $A \subseteq \mathbb{A}$ liegen, der ein Workflow Teilnehmer $w \in \mathbb{WFT}$ zugewiesen wurde, und wenn die Bedingung

$$|tool(a_1) \cap tool(a_2)| \geq 1$$

gilt.

Ein Teilnehmertyp kann eine Aktivität bereitstellen, wenn er über alle notwendigen Werkzeuge für den Einsatz verfügt. Die Verfügbarkeit von Werkzeugen wird durch das Werkzeugmagazin, das eine beschränkte Anzahl von Plätzen hat, bestimmt. Ist das Magazin klein, kann nur eine kleine Auswahl von einzusetzenden Aktivitäten getroffen werden, obwohl möglicherweise viele Aktivitäten prinzipiell zur Verfügung stehen.

Die Teilnehmertypdefinition 4.1.6 wird somit erweitert.

Definition 4.2.3 (Erweiterter WFT). Die Menge aller erweiterten Workflow Teilnehmertypen $\widetilde{\mathbb{WFT}}$ ist wie folgt definiert:

$$\widetilde{\mathbb{WFT}} = \mathcal{P}(\mathbb{A}) \times \mathbb{K} \times \mathbb{Ti} \times \mathbb{Pr} \times \mathbb{N}$$

Sei $\tilde{w} = (A, k, rz, s, mg) \in \mathbb{WFT}$ ein erweiterter Workflow Teilnehmertyp. Hierbei sind $A \in \mathcal{P}(\mathbb{A})$ die Fähigkeiten, $k \in \mathbb{K}$ die Kosten, $rz \in \mathbb{Ti}$ die Rüstzeit, $s \in \mathbb{Pr}$ die Stabilität und $mg \in \mathbb{N}$ die Größe des Werkzeugmagazins. Folgende neue Projektion wird ergänzt:

$$mag : \quad \widetilde{\mathbb{WFT}} \to \mathbb{N} \quad \text{mit} \quad mag(\tilde{w}) = mg$$

4.2.2 Lokale Bearbeitungszeiten

Eine weitere Verfeinerung des Problems ist die Einführung von lokalen Bearbeitungszeiten. Demnach hängt die Bearbeitungszeit einer Aktivität von dem Teilnehmertyp, der sie ausführt, ab.

Definition 4.2.4 (Lokale Bearbeitungszeit). Sei $w \in \mathbb{WFT}$ ein Teilnehmertyp. Eine Funktion $bz : \mathbb{WFT} \times \mathbb{A} \to \mathbb{Ti}$ gibt mit $bz(w, a)$ die lokale Bearbeitungszeit einer Aktivität $a \in akt(w)$ auf dem Teilnehmertyp w an.

Hinweis. Aufgrund der lokalen Bearbeitungszeiten muß die Funktion der Blockbearbeitungszeit angepaßt werden. Diese sieht nun wie folgt aus:

$$bbz(A, w) = \begin{cases} \left(\sum_{a \in A} bz(w, a)\right) + (rz(w) + (|A| - 1) * guz) & \text{, falls } |A| \geq 1 \\ 0 & \text{, sonst} \end{cases}$$

4.2.3 Vermischter Datenfluß

Während der separate Datenfluß zu jedem Zeitpunkt maximal einen aktiven Ablauf verlangt, erlaubt der vermischte Datenfluß mehrere aktive Abläufe. Das heißt, daß mehrere, verschiedene Objekte (von verschiedenen Abläufen) in einem vorgegebenen Verhältnis an einem Workflow beteiligt sind.

Beispiel 4.2.5. Seien $\varphi_1, \varphi_2 \in \Phi$ zwei Abläufe und sei das Verhältnis 30:70. Angenommen, es befinden sich zehn Objekte im Workflow, dann fallen drei von ihnen auf den Ablauf φ_1 und sieben auf den Ablauf φ_2.

Eine Erweiterung der Definition des Ablaufs (Def. 4.1.7) um eine Verhältnisangabe ergibt folgende Definition.

Definition 4.2.6 (Erweiterter Ablauf). Die Menge aller *erweiterten Abläufe* ist definiert als:

$$\tilde{\Phi} = \Phi \times \mathbb{Pr}$$

Sei $\tilde{\varphi} = (\varphi, \mathtt{pr}) \in \tilde{\Phi}$ ein erweiterter Ablauf, wobei φ der Ablauf und $\mathtt{pr}$ der prozentuale Anteil ist. Folgende Projektionen sind bekannt:

$$\begin{array}{llll} at: & \tilde{\Phi} \to \mathbb{Pr} & \text{mit} & at(\tilde{\varphi}) = \mathtt{pr} \\ abl: & \tilde{\Phi} \to \Phi & \text{mit} & abl(\tilde{\varphi}) = \varphi \end{array}$$

Eine Problemstellung, bestehend aus der erweiterten Ablaufmenge $\tilde{Q} \in \mathcal{P}\left(\tilde{\Phi}\right)$, muß gewährleisten, daß die Bedingung

$$\sum_{\tilde{\varphi} \in \tilde{Q}} at(\tilde{\varphi}) = 100$$

gilt, damit alle Abläufe in einem echten Verhältnis zueinander stehen.

Zunächst soll die Darstellung eines vermischten Flusses betrachtet werden. Bild 2.8 auf Seite 26 veranschaulicht einen vermischten Graphen, der aus den beiden Abläufen `Bestellung` und `Reklamation` (Abbildungen 2.1 und 2.6) gewonnen wurde. Die Aktivitäten der Abläufe wurden hierbei so umbenannt, daß es keine Überschneidungen mehr gibt. In die Berechnungen geht nur der neu entstandene Ablauf ein. Betrachtungen der Anzahl- bzw. Taktdominanz spielen keine Rolle. Dies ist so zu verstehen, daß ein Objekt simuliert wird, welches alle Aktivitäten aller Abläufe durchläuft. In der Realität ist dies nicht möglich, da beispielsweise ein konstruiertes Haus nicht gleichzeitig eine Brücke sein kann.

Satz 4.2.7. Seien $a, a' \in \mathbb{A}$. Aktivität a' ist eine Umbenennung von a gdw.

$$tool(a) = tool(a').$$

Die Aktivitätenumbenennung ist notwendig, da andernfalls neue Relationen entstehen würden, die falsche Informationen dem System hinzufügen. Angenommen, man würde Umbenennungen auslassen und mit dem vereinigten Graphen (Bild 2.7, Seite 25) arbeiten. Dieser enthält die Kante $12 \overset{\varphi}{\to} 13$, welche in der Prozeßdefinition B nicht zum Ausdruck kommt. Ein Objekt, das nach Prozeßdefinition B bearbeitet wird, erlaubt ohne Einschränkung durch Workflow Teilnehmer eine Aktivitätsequenz $(0, 11, 13, 12, \ldots)$. Ein Objekt, daß den vereinigten Graphen durchläuft, darf wegen der Relation $12 \overset{\varphi}{\to} 13$ nicht dieser Sequenz folgen. Man bedenke, daß nur simuliert wird, daß ein Objekt alle Aktivitäten durchläuft. Ein Objekt der Prozeßdefinition B muß weiterhin nach dieser Definition bearbeitet werden. Es dürfen keine neuen Einschränkungen entstehen. Deswegen werden die Aktivitäten so umbenannt, daß alle Prozeßdefinitionen ihre interne Logik behalten.

Für das nachfolgende Vorgehen sei eine erweiterte Ablaufmenge $\tilde{Q} \in \mathcal{P}\left(\tilde{\Phi}\right)$ mit Umbenennungen gegeben, so daß alle Aktivitätsmengen paarweise disjunkt sind.

$$\forall \tilde{\varphi}_i, \tilde{\varphi}_j \in \tilde{Q},\ \tilde{\varphi}_i \neq \tilde{\varphi}_j : \quad \mathit{akt}(\mathit{abl}(\tilde{\varphi}_i)) \cap \mathit{akt}(\mathit{abl}(\tilde{\varphi}_j)) = \emptyset$$

Außerdem sei $S = (S_1, S_2, \ldots, S_n)$ eine Partition der Menge $\mathit{akt}(\mathit{res}(\mathit{abl}(\tilde{Q})))$ in n Blöcke.

Die anteilsmäßige Blockbearbeitungszeit für einen erweiterten Ablauf $\tilde{\varphi} \in \tilde{Q}$ und für eine Menge S_i aus der Partition S lautet

$$\left\lceil \frac{\mathit{at}(\tilde{\varphi}) * \mathit{bbz}(\mathit{abl}(\tilde{\varphi}), S_i, w)}{100} \right\rceil,$$

wobei $w \in \mathbb{WFT}$ der eingesetzte Teilnehmer für S_i ist.

4.2.3.1 Vermischter Datenfluß

Es gibt zwei Formen des vermischten Datenfluß, den *rein-vermischten Datenfluß* und den *kombiniert-vermischten Datenfluß*.

Rein-vermischter Datenfluß Dieser zeichnet sich dadurch aus, daß die Blockbearbeitungszeitanteile in einem Block summiert werden, und daß nur bezüglich dieser Summe die Anzahl der Teilnehmer für einen Block ermittelt wird.

Kombiniert-vermischter Datenfluß In diesem Datenfluß wird erst die Anzahl der Teilnehmer pro Block bezüglich der Blockbearbeitungsanteile ermittelt. Anschließend werden die Anteile der Teilnehmerzahl summiert.

Die Form des rein-vermischten Datenflusses ist sehr eng kalkuliert. Der Vorteil gegenüber dem kombiniert-vermischten Modus ist, daß nicht verbrauchte Kapazitäten eines Ablaufs durch einen anderen mitbenutzt werden, und somit Teilnehmer gespart werden. Freie Kapazitäten sind am relativen Zeitfenster ablesbar. Ist dieses groß, dann können Restbearbeitungszeiten anderer Abläufe aufgenommen werden. Dagegen garantiert der kombinierte-vermischte Datenfluß, daß jedem Ablauf genügend Teilnehmer bereitgestellt werden, weil die Anzahl pro Ablauf berechnet wird. Man betrachte das nachfolgende Beispiel und Abbildung 4.3.

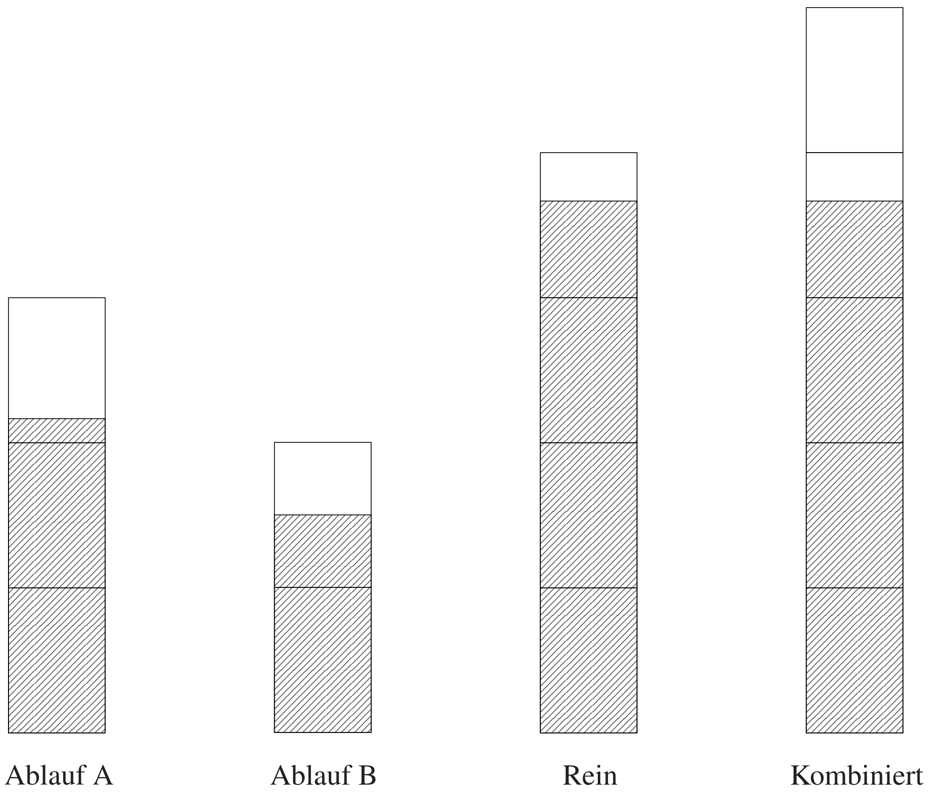


Abbildung 4.3: Modi des vermischten Datenfluß

Beispiel 4.2.8. Seien $\tilde{\varphi}_1, \tilde{\varphi}_2 \in \tilde{\Phi}$ erweiterte Abläufe und $A \in \mathcal{P}\,(akt(V))$ eine Menge von Aktivitäten, wobei $V = res(\{abl(\tilde{\varphi}_1), abl(\tilde{\varphi}_2)\})$ die Vereinigung der Abläufe $\tilde{\varphi}_1$ und $\tilde{\varphi}_2$ ist. Das Verhältnis beider Abläufe beträgt $50 : 50$. Das absolute Zeitfenster wird auf $z = 100$ festgelegt, die Blockbearbeitungszeit von $\tilde{\varphi}_1$ auf $t_1 = 220$ und die von $\tilde{\varphi}_2$ auf $t_2 = 150$ gesetzt. Zunächst ist die Teilnehmeranzahl in Abhängigkeit des absoluten Zeitfensters leicht abzulesen:

bbz	< 100	< 200	< 300	< 400	…
anz	1	2	3	4	…

Daraus leiten sich nachfolgende Teilnehmerzahlen für A ab.

$\boldsymbol{anz}(\tilde{\varphi}_1)$	$\boldsymbol{anz}(\tilde{\varphi}_2)$	$\boldsymbol{anz}(\tilde{\varphi}_1 + \tilde{\varphi}_2)$	$\boldsymbol{anz}(\tilde{\varphi}_1) + \boldsymbol{anz}(\tilde{\varphi}_2)$
3	2	4	5

Bei separater Betrachtung benötigt der Ablauf $\tilde{\varphi}_1$ drei und $\tilde{\varphi}_2$ zwei Teilnehmer. Es wird deutlich, daß der rein-vermischte Datenfluß (vorletzte Spalte) einen Teilnehmer weniger beansprucht als der kombiniert-vermischte Datenfluß (letzte Spalte). Dies bestätigt ebenfalls die Abbildung. Sie zeigt auch, wieviel Kapazität für die Blockbearbeitungszeit des kombiniert-vermischten Modus verbleibt, nämlich die eines zusätzlichen Teilnehmers. Im Gegensatz dazu ist im rein-vermischten Modus zu sehen, daß die

Restbearbeitungszeit des Ablaufs $\tilde{\varphi}_2$ vom Teilnehmer des Ablaufs $\tilde{\varphi}_1$ übernommen wird, da seine Kapazitäten ausreichend sind.

Da jetzt der Unterschied beider Verfahrensarten bekannt ist, muß für beide ein Algorithmus zur Lösungsfindung angegeben werden. Zunächst der für den rein-vermischten Datenfluß.

4.2.3.2 Lösung rein-vermischter Datenfluß

Die Aufgabenstellung unterscheidet sich von der des separaten Datenflußes nur durch die erweiterten Abläufe.

Sei S eine Partition von $akt(\mathrm{res}(abl(\tilde{Q})))$ in n Blöcke. Die Summe der anteilsmäßigen Blockbearbeitungszeiten der n Blöcke wird mit $G \in \mathbb{Ti}^n$ dargestellt. Dieses Tupel wird wie folgt mit Werten belegt:

$$\forall i \in \{1, 2, \ldots, n\} \; : \; G_i = \sum_{\tilde{\varphi} \in \tilde{Q}} \left\lceil \frac{at(\tilde{\varphi}) * bbz(abl(\tilde{\varphi}), S_i, Ws_i)}{100} \right\rceil ,$$

Es muß beachtet werden, daß bisher keine Festlegung auf lokale oder globale Bearbeitungszeiten stattfand. Dadurch wird die Flexibilität gewahrt. Sollen lokale Bearbeitungszeiten angewendet werden, muß die Funktion *bbz*, speziell die Funktion *bz*, angepaßt werden.

Für die Berechnung der Teilnehmersumme fällt zunächst die Anzahldominanz weg. Da die Blockbearbeitungszeit vorberechnet wurde, sieht das Tupel Ms leicht verändert aus:

$$\forall i \in \{1, 2, \ldots, n\} \; : \; Ms_i = \left\lceil \frac{G_i * (100 + \nu)}{stab(Ws_i) * \mathrm{mT}} \right\rceil$$

Ähnlich leicht verändert sieht die Berechnung der effektiven Taktzeit aus, die ohne Taktdominanz auskommt:

$$\mathrm{eT} = \max \left\{ t \;\middle|\; i \in \{1, 2, \ldots, n\} \, , \; t = \left\lceil \frac{G_i * (100 + \nu)}{stab(Ws_i) * Ms_i} \right\rceil \right\}$$

Die restlichen Schritte bis zur Bestimmung der Kosten sind analog zum separaten Datenfluß.

4.2.3.3 Lösung kombiniert-vermischter Datenfluß

Die Aufgabenstellung unterscheidet sich von der des separaten Datenflußes nur durch die erweiterten Abläufe.

Sei S eine Partition von $akt(\mathrm{res}(abl(\tilde{Q})))$ in n Blöcke. Die Anzahl der Teilnehmer hat nun einen anteilsmäßigen Aspekt. Die Anzahldominanz wird nicht angewendet. Das Tupel Ms berechnet sich nach

$$\forall i \in \{1, 2, \dots, n\} \; : \; Ms_i = \sum_{\tilde{\varphi} \in \tilde{Q}} \left\lceil \frac{at(\tilde{\varphi}) * anz(abl(\tilde{\varphi})), S_i, Ws_i)}{100} \right\rceil$$

Zuletzt wird die effektive Taktzeit ermittelt, die das Maximum der summierten, anteilsmäßigen Taktzeiten ist. Auch hier kommt die Taktdominanz nicht zur Anwendung.

$$\mathrm{eT} = \max \left\{ t \;\middle|\; i \in \{1, 2, \dots, n\}, \; t = \sum_{\tilde{\varphi} \in \tilde{Q}} \left\lceil \frac{at(\tilde{\varphi}) * takt(abl(\tilde{\varphi}), S_i, Ws_i, Ms_i)}{100} \right\rceil \right\}$$

Die Kostenberechnung ist analog zum separaten Datenfluß.

4.2.4 Kostenverfeinerung

Eine weitere Problemverfeinerung besteht darin, Kosten zu differenzieren. Es besteht nun die Möglichkeit, Blockkosten und Werkzeugkosten in die Gesamtkosten einzubeziehen. Somit ist eine Methode verfügbar, die Aktivitäten zur gemeinsamen Werkzeugnutzung zu bewegen, um damit eine Kostenminimierung anzustreben. Alle Werkzeuge haben die gleichen Kosten.

Definition 4.2.9. Die Kosten pro Werkzeug betragen $K_{\mathrm{Wz}} \in \mathbb{K}$.

Eine andere Minimierung der Kosten ist über die Blockanzahl möglich. Je mehr Blöcke eingesetzt werden, umso höher fallen die Kosten aus. Die Kosten pro Block sind ebenfalls einheitlich.

Definition 4.2.10. Die Kosten pro Block betragen $K_{\mathrm{Bl}} \in \mathbb{K}$.

Die Kosten für die Teilnehmer K_1 werden wie bisher berechnet. Die Blockkosten betragen bei n Blöcken $K_2 = n * K_{\mathrm{Bl}}$. Die Summe aller Werkzeuge muß ein Mehrfachvorkommen bei gemeinsamer Nutzung ausschließen.

$$K_3 = K_{\mathrm{Wz}} * \sum_{i=1}^{n} \left| \bigcup_{a \in S_i} tool(a) \right|$$

Zum Schluß fehlt nur noch die Gesamtsumme, die aus den hier vorgestellten Teilsummen besteht:

$$\tilde{K}_G = K_1 + K_2 + K_3$$

Kapitel 5

Propagierung

Nachdem im vorigen Kapitel das mathematische Modell des Problems vorgestellt wurde, widmet sich dieses Kapitel den mathematischen Eigenschaften. Diese Eigenschaften reduzieren den Suchbaum, indem weitere mögliche Belegungen ausgeschlossen werden. Neben den basic und non-basic Constraints ist dem Benutzer die Möglichkeit gegeben, eigene Constraints zu formulieren. Desweiteren ist die Berechnung von bestimmten Schranken im Vorfeld möglich.

Für die folgenden Betrachtungen sei $\gamma = (Q, W, mT, \nu, guz) \in \Gamma$ eine Problemstellung. Die Menge aller verwendeten Aktivitäten ist $A_{res} = akt(res(Q))$. Die Betrachtungen richten sich nach dem Grundmodell. Eine Erweiterung auf das präzise Modell ist trivial und wird hier nicht weiter ausgeführt.

5.1 Basic Constraints

Allen Variablen jeder Lösung wird ein initialer Bereiche zugewiesen. Diese Variablen gilt es zu determinieren, d.h. mit Werten zu belegen.

Die Zahl der Blöcke jeder Lösung besteht aus minimal einem Block und maximal aus der Anzahl aller am Prozeß beteiligten Aktivitäten.

Filter 5.1.1 (Anzahl Blöcke). Sei $N = |A_{res}|$ die Anzahl der in γ vorkommender Aktivitäten. Dann gilt für alle Lösungen $l \in loes(\gamma)$:

$$1 \leq bloecke(l) \leq N.$$

Eine Partition besteht aus Mengen. Mengen werden über ihre Elemente oder über ihre Mächtigkeit beschrieben. Jede Menge der Partition enthält mindestens ein Element, d.h. sie hat mindestens eine Mächtigkeit von eins. Eine Menge kann maximal alle Elemente besitzen, abzüglich derer, die schon für andere Mengen bestimmt sind (mindestens $n - 1$).

Filter 5.1.2 (Partition). Für alle Lösungen $l \in loes(\gamma)$ gilt: sei $S = part(l)$ die Partition und $n = bloecke(l)$ die Anzahl Blöcke von l. Dann beträgt die Mächtigkeit aller S_i mit $1 \leq i \leq n$:

$$1 \leq |S_i| \leq (|A_{res}| - n + 1).$$

Jeder Block einer Lösung bekommt einen Teilnehmertypen zugewiesen. Es muß einer der Typen aus der Problemstellung sein. Die Anzahl der Teilnehmer pro Block ist zunächst nach oben offen und nach unten auf einen begrenzt.

Filter 5.1.3 (Teilnehmer). Für alle Lösungen $l \in loes(\gamma)$ gilt: sei $n = bloecke(l)$ die Anzahl Blöcke von l. Dann gilt für alle $1 \leq i \leq n$:

$$wft(l)_i \in wft(\gamma)$$
$$1 \leq anz(l)_i$$

Bleiben noch die effektive Taktzeit und die Kosten. Der Takt darf nicht null und nicht länger als die vorgegeben, maximale Taktzeit sein.

Filter 5.1.4 (Takt). Für alle Lösungen $l \in loes(\gamma)$ gilt:

$$1 \leq takt(l) \leq takt(\gamma).$$

Die Kosten dürfen nicht negativ ausfallen. Nach oben gibt es keine Begrenzung.

Filter 5.1.5 (Kosten). Für alle Lösungen $l \in loes(\gamma)$ gilt:

$$1 \leq kost(l).$$

5.2 Non-basic Constraints

In Kapitel 4.1.2 wurde dargestellt, welche Eigenschaften eine Partition im Allgemeinen besitzt. In diesem Kapitel wird die Bildung einer Partition mit Hilfe von Filteralgorithmen konkretisiert. Filter haben die Aufgabe, inkonsistente Aktivitätszuweisungen an Blöcke auszuschließen.

Wie schon in Kapitel 3.2.1 dargestellt, ist die Propagierung unvollständig. Daher muß die Partition nach Anwendung von Filtern nicht determinieren. Eine eindeutige Partition liegt erst nach der Explorierung (Kapitel 6) vor.

Die Start- bzw. Endaktivität ist in der ersten bzw. in der letzten Menge der Partition enthalten. Zudem ist bekannt, daß es sich um einelementige Mengen handelt. Desweiteren sind die Teilnehmertypen für den ersten bzw. letzten Block bekannt, weil genau ein Teilnehmertyp zur Startaktivität und genau ein Teilnehmertyp zur Endaktivität existiert. Da die Plätze für die beiden Sondertypen vergeben sind, werden sie als Zuweisungsmöglichkeiten für andere Plätze ausgeschlossen. Gleiches gilt für die Aktivitäten. Dies folgt aus der paarweisen Disjunktheit der Partition.

Filter 5.2.1. Sei a_s die Start- und a_e die Endaktivität. Außerdem sei t_s der Teilnehmertyp für a_s und t_e der für a_e. Dann gilt für alle Lösungen $l \in loes(\gamma)$ mit einer Partition $S = part(l)$:

$$\begin{aligned} S_1 &= \{a_s\} \\ S_n &= \{a_e\} \\ (wft(l))_1 &= t_a \\ (wft(l))_n &= t_e \end{aligned}$$

Es gibt zwei Quellen, die eine Einschränkung der Zuweisungsmöglichkeiten der Aktivitäten bieten. Dies sind die Kanten in den Ablaufplänen und der Vergleich der Fähigkeiten einzelner Teilnehmertypen.

Eine Kante $a \xrightarrow{\varphi} b$ sagt aus, daß Aktivität b nie vor Aktivität a bearbeitet wird, d.h. b darf in keinem früheren Block als a platziert werden. Beide Aktivitäten können nur dann in einem gemeinsamen Block enthalten sein, wenn es einen Teilnehmer gibt, der beide beherrscht. Ansonsten ist b in einem späteren Block zu platzieren.

Filter 5.2.2. Sei $R_{res} = rel(res(Q))$. Dann gilt für alle Lösungen $l \in loes(\gamma)$ mit einer Partition $S = part(l)$:

$$\begin{array}{rr} \forall (a,b) \in R_{res}: & \exists w \in W: \quad \{a,b\} \subseteq akt(w) \\ & \Downarrow \\ & a \in S_i \;\wedge\; b \in S_j \quad \text{mit } 1 \leq i \leq j \leq n \end{array}$$

Existiert kein solcher Teilnehmertyp, so folgt:

$$a \in S_i \;\wedge\; b \in S_j \quad \text{mit } 1 \leq i < j \leq n$$

Allgemeiner kann man alle möglichen Paare von Aktivitäten auf ihre Zugehörigkeit zu einem Block in Abhängigkeit der Teilnehmertypen untersuchen. Gibt es zwei Aktivitäten, die von keinem Teilnehmertyp gemeinsam bereitgestellt werden, liegen sie in unterschiedlichen Blöcken.

Filter 5.2.3. Gegeben sei A_{res}. Dann gilt für alle Lösungen $l \in loes(\gamma)$:

$$\begin{array}{rr} \forall a,b \in A_{res}: & \nexists w \in W: \quad \{a,b\} \subseteq akt(w) \\ & \Downarrow \\ & a \in S_i \;\wedge\; b \in S_j \quad \text{mit } 1 \leq i,j \leq n,\ i \neq j \end{array}$$

Teilnehmertypen sind in der Lage, die Partitionsmengen noch weiter einzuschränken. Eine Menge kann nämlich nur die Aktivitäten enthalten, die der Teilnehmer prinzipiell zur Verfügung stellt. Dies hat automatisch Einfluß auf die Mächtigkeit der Aktivitätenmenge; sie kann maximal so groß sein wie die Anzahl der Fähigkeiten des Teilnehmers.

Filter 5.2.4. Für alle $l \in loes(\gamma)$ gilt: sei $S = part(l)$ die Partition, $n = bloecke(l)$ die Zahl der Blöcke und $Ws = wft(l)$ die Teilnehmertypkonfiguration von (l). Für alle Blöcke S_i mit $1 \leq i \leq n$ sind folgende non-basic Constraints zu erfüllen:

$$S_i \subseteq akt(Ws_i)$$
$$|S_i| \leq |\ akt(Ws_i)\ |$$

Die Organisation der Teilnehmertypen muß so festgelegt sein, daß jede Aktivität aus A_{res} in der Organisation eine Zuweisung erfährt. Das heißt, legt man die Aktivitäten aller Teilnehmertypen der Lösung zusammen, so bilden diese eine Obermenge für die Aktivitäten aller Prozesse. Dieser redundante Constraint vermittelt weiteres Wissen über die geforderten Teilnehmertypen.

Filter 5.2.5. Sei A_{res} die Menge aller an den Prozessen beteiligten Aktivitäten. Dann gilt für alle Lösungen $l \in loes(\gamma)$ mit der Teilnehmertypkonfiguration $Ws = wft(l)$:

$$A_{res} \subseteq \bigcup_{i=1}^{n} akt(Ws_i).$$

5.3 Benutzerdefinierte Constraints

In der Praxis hat man teilweise genaue Vorstellungen über Prozeßlogik, Organisation und Infrastruktur. Deswegen muß dem Anwender ein Weg angeboten werden, seine Vorstellungen als Nebenbedingungen einzubringen. Die Nebenbedingungen werden in einer eigenen, domänenspezifischen Sprache angegeben und in die entsprechenden Filterungsalgorithmen übertragen. Die domänenspezifischen Filter sind Gegenstand in diesem Abschnitt.

5.3.1 Teilnehmertypen

Die Planung beginnt mit den vorhandenen Ressourcen. Daraus ergeben sich weitere Eigenschaften hinsichtlich der Belegung durch Aktivitäten. So ist zunächst die Möglichkeit gegeben, jedem Block einen bestimmten Teilnehmertyp zuzuweisen.

Filter 5.3.1. Sei i ein ausgesuchter Block und sei t ein bestimmter Teilnehmertyp. Dann wird der Teilnehmertyp t für den Block i für alle Lösungen $l \in loes(\gamma)$ festgelegt:

$$wft(l)_i = t.$$

Entgegengesetzt kann Typ t verboten werden:

$$wft(l)_i \neq t.$$

5.3.2 Anzahl Teilnehmer

Zu den effizientesten Modellierungsschritten gehören Aussagen über die Anzahl von Teilnehmern. Dazu gehören Aussagen zur gesamten Infrastruktur, Aussagen zu einem Block und Aussagen zu einem Teilnehmertyp. Die ersten beiden Aussageformen drückt der folgende Filter aus.

Filter 5.3.2. Sei i ein ausgesuchter Block und sei m eine Begrenzung der Anzahl von Teilnehmern. Dann gilt für alle Lösungen $l \in \mathit{loes}(\gamma)$ mit $Ms = \mathit{anz}(l)$: nach (5.1) ist m eine obere (bzw. untere) Schranke für die Anzahl von Teilnehmern im Block i und nach (5.2) ist m eine obere (bzw. untere) Schranke für die Gesamtanzahl von Teilnehmern.

$$m \geq Ms_i \qquad\qquad m \leq Ms_i \tag{5.1}$$

$$m \geq \left(\sum_{j=1}^{n} Ms_j\right) \qquad\qquad m \leq \left(\sum_{j=1}^{n} Ms_j\right) \tag{5.2}$$

Die Berechnung der Anzahl Teilnehmer von einem bestimmten Typ verläuft ähnlich. Hier wird zusätzlich der Teilnehmertyp von jedem Block überprüft.

Filter 5.3.3. Sei t ein ausgesuchter Teilnehmerytp und sei m eine Begrenzung der Anzahl von Teilnehmern des Typs t. Die Hilfsfunktion *isT* ist definiert als:

$$\mathit{isT}(x) = \begin{cases} 1 & \mathit{falls}\ x{=}t, \\ 0 & \mathit{sonst}. \end{cases}$$

Dann gilt für alle Lösungen $l \in \mathit{loes}(\gamma)$, wobei $Ws = \mathit{wft}(l)$ die Teilnehmertypkonfiguration und $Ms = \mathit{anz}(l)$ die Anzahl der Teilnehmer beschreibt: m ist eine obere (bzw. untere) Grenze für die Anzahl von Teilnehmern des Types t, falls

$$m \geq \left(\sum_{j=1}^{n} Ms_j * \mathit{isT}(Ws_j)\right) \qquad\qquad m \leq \left(\sum_{j=1}^{n} Ms_j * \mathit{isT}(Ws_j)\right) \tag{5.3}$$

5.3.3 Aktivitäten

Ein weiterer Schwerpunkt der benutzerdefinierten Constraints ist die Modellierung der Aktivitäten. So ist es möglich, schon während der Problemformulierung eine Zuweisung von Aktivitäten zu einem Block anzugeben.

Filter 5.3.4. Sei i ein ausgesuchter Block und sei S_x eine Menge von Aktivitäten. Dann sind für alle Lösungen $l \in \mathit{loes}(\gamma)$ mit einer Partition $S = \mathit{part}(l)$ folgende Non-basic Constraints bekannt:

$S_x \subseteq S_i$	Zugehörigkeit
$S_x \cap S_i = \emptyset$	Ausschluß

Ähnlich wird die Aussage getroffen, daß bestimmte Aktivitäten gemeinsam in einem Block liegen, wobei dieser Block zunächst unbekannt ist.

Filter 5.3.5. Sei S_x eine Menge von Aktivitäten. Dann gilt für alle $l \in \mathit{loes}(\gamma)$ mit einer Partition $S = \mathit{part}(l)$:

$$\exists i : S_x \subseteq S_i$$

Zusätzlich zur Zuweisung von Aktivitäten zu einem Block, ist die Zuweisung zu einem bestimmten Teilnehmertyp möglich. Das heißt, die Aktivitäten müssen genau in den Blöcken liegen, die dem bestimmten Teilnehmertyp zugeordnet sind. Dabei dürfen die Aktivitäten auf mehrere Blöcke verteilt sein.

Filter 5.3.6. Sei t ein bestimmter Teilnehmertyp und sei S_x eine Menge von Aktivitäten. Dann gilt für alle Lösungen $l \in \mathit{loes}(\gamma)$ mit einer Partition $S = \mathit{part}(l)$ und Teilnehmertypkonfiguration $Ws = \mathit{wft}(l)$:

$$S_x \subseteq \bigcup_{i=1}^{n} \{S_i \mid Ws_i = t\}$$

Letztendlich sind auch Bedingungen über die Mächtigkeit von Aktivitätsblöcken erlaubt. Damit sind Ausdrücke über die Mindest- bzw. Maximalanzahl von Aktivitäten in einem Block ausdrückbar.

Filter 5.3.7. Sei i ein bestimmter Block und m die Anzahl von Aktivitäten, dann drücken folgende Constraints die Minimal- bzw. Maximalanzahl von Aktivitäten aus. Für alle Lösungen $l \in \mathit{loes}(\gamma)$ mit $S = \mathit{part}(l)$ gilt:

$$m \leq |S_i| \qquad\qquad m \geq |S_i|$$

5.3.4 Werkzeuge

Für Problemstellungen des präzisen Modells, in dem Werkzeuge eingesetzt werden, ist eine Bedingung für den maximalen Werkzeugaustausch erwünscht. Es dürfen nur die Werkzeuge ausgetauscht werden, die nicht in allen Prozessendefinitionen vorgesehen sind. Ein Austausch findet statt, wenn das System eine Prozeßdefinition wechselt.

Filter 5.3.8. Sei m der maximale Werkzeugaustausch und sei Q die Menge der Abläufe. Für alle Lösungen $l \in \mathit{loes}(\gamma)$ wird der maximale Werkzeugaustausch nach folgendem Schema berechnet, wobei $S = \mathit{part}(l)$ die Partition und $n = \mathit{bloecke}(l)$ die Zahl der Blöcke von l ist:

$$\forall \varphi_1, \varphi_2 \in Q \text{ mit } \varphi_1 \neq \varphi_2 \text{ gilt: } m \geq t_{\text{sum}} \quad \text{wobei}$$

```
t_sum = 0                      // Gesamtanzahl Werkzeugwechsel
for i=1 to n
   t_1 = tool(φ_1, S_i)        // Werkzeuge des Ablaufs φ_1 im Block S_i
   t_2 = tool(φ_2, S_i)        // Werkzeuge des Ablaufs φ_2 im Block S_i

   t_r=union(t_1,t_2)
   t_s=intersection(t_1,t_2)
   t_t=compl(t_r,t_s)          // Gemeinsame Werkzeuge ausschließen

   t_sum += card(t_t)          // Kardinalität von t_t aufsummieren
end
```

5.4 Schranken

Es ist möglich, eine erste untere Schranken für Kosten und Infrastruktur aus den Informationen der Problemstellung zu bestimmen. Grundlegend hierfür ist die Mindestanzahl von Teilnehmern, die natürlich für alle Prozeßdefinitionen gilt.

Die nachfolgenden Betrachtungen behandeln sowohl die Gesamtanzahl von Teilnehmern als auch die Anzahl von Teilnehmern eines bestimmten Typs.

5.4.1 Untere Schranke für Gesamtanzahl von Teilnehmer

In diesem Abschnitt wird ein Verfahren vorgestellt, das eine untere Schranke für die Gesamtanzahl von Teilnehmern zu einem Problem ermittelt. Dieses Verfahren bezieht sich zunächst nur auf die Problemstellung des Grundmodells mit einer Prozeßdefinition. Anschließend wird die Beobachtung sukzessive auf mehrere Prozesse, lokale Bearbeitungszeiten und vermischten Datenfluß erweitert.

Grundlegend für die Bestimmung einer untere Schranke ist die Formel *anz* nach Definition 4.1.21.

$$anz(A, w) = \left\lceil \frac{bbz(A, w) * (100 + v)}{st * mT} \right\rceil$$

Sie besagt, daß genügend Teilnehmer des Typs w zur Verfügung stehen müssen, um alle Aktivitäten der Menge A unter Beachtung aller Vorgaben (v, st und mT) zu bearbeiten. Der Verlust und die geforderte Taktzeit sind bekannt. Die Stabilität hängt vom Teilnehmer w ab.

Idee. Die zentrale Idee dieser Betrachtung ist die Annahme, daß eine Lösung genau dann die wenigsten Teilnehmer enthält, wenn die Partition aus einem Block besteht. Dadurch werden Rüstzeiten für weitere Blöcke gespart. Weiterhin wird angenommen, daß der stabilste Teilnehmer der Problemstellung sämtliche Aktivitäten beherrscht.

Für die Abschätzung der Kosten nach unten darf der stabilste Teilnehmertyp nicht mehr zu Rate gezogen werden. Stattdessen wird der günstigste Teilnehmertyp ausgesucht. Das heißt, es wird angenommen, daß für einen Block zwei Teilnehmertypen eingesetzt werden. Der stabilste wird zur Berechnung der Teilnehmeranzahl und der günstigste zur Berechnung der Kosten eingesetzt.

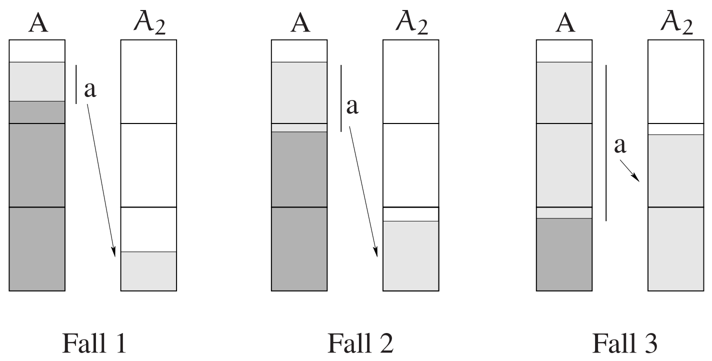
Satz 5.4.1. Sei $w_1 \in W$ der Teilnehmertyp mit der höchsten Stabilität, und sei $w_2 \in W$ der günstigste Teilnehmertyp. Desweiteren sei A eine Menge von Aktivitäten. Falls $rz(w_1) \geq guz$, dann beträgt die Mindestanzahl von Teilnehmern:

$$anz(A, w_1) \leq \sum_{1 \leq i \leq n} Ms_i$$

Daraus folgt die untere Schranke für die Kosten:

$$anz(A, w_1) * kost(w_2) \leq kost(l)$$

Beweis. Es wird die Korrektheit der unteren Grenze für die Anzahl der Teilnehmer bewiesen. Man nehme die Abbildung 4.2 auf Seite 49 als Anschauung zu Hilfe. Angenommen, die Partition besteht aus nur einer Menge A, die alle Aktivitäten enthält. Sei w der stabilste Teilnehmertyp. Des weiteren sei die Blockbearbeitungszeit $G_A = bbz(A, w)$ mit dem absoluten Zeitfenster Z_A und der Restzeit R_A gegeben. Es wird nun bewiesen, daß es keine Partition von A in $n > 1$ Mengen gibt, die weniger Teilnehmer benötigt. Der Beweis beginnt mit $n = 2$. Hierzu nehme man eine beliebige Aktivität $a \in A$ heraus und erhalte die Mengen $A_1 = A \setminus \{a\}$ und $A_2 = \{a\}$. Es folgt eine Fallunterscheidung nach der Bearbeitungszeit von a.



1. Fall: $bz(a) \leq R_A$
Dann bleibt $anz(A_1, w) = anz(A, w)$ und $anz(A_2, w) \geq 1$
$\Rightarrow anz(A_1, w) + anz(A_2, w) > anz(A, w)$.

2. Fall: $R_A < bz(a) < R_A + Z_A$
Dann ist $anz(A_1, w) = anz(A, w) - 1$ und $anz(A_2, w) \geq 1$
$\Rightarrow anz(A_1, w) + anz(A_2, w) \geq anz(A, w)$.

3. Fall: $bz(a) > R_A + Z_A$
Dann ist $anz(A_1, w) \leq anz(A, w) - 2$ und $anz(A_2, w) \geq 2$
Es gilt folgende Beobachtung:
$anz(A, w) - anz(A_1, w) \leq anz(A_2, w)$
da sich die Blockbearbeitungszeiten verändert haben:
$X = bbz(A, w) - bbz(A_1, w) = bz(A) + guz$
$Y = bbz(A_2, w) = bz(a) + rz(w)$
Daraus folgt:
$X < Y$ falls $rz(w) > guz$.
$\Rightarrow anz(A_1, w) + anz(A_2, w) \geq anz(A, w)$

Bisher wurde gezeigt, daß der Satz für eine Aktivität gilt. Es fehlt der Beweis für beliebige Teilmengen $A_2 \subseteq A$. Es reicht, wenn die Aktivitäten in A_2 auf eine einzelne Aktivität a reduziert werden. Man addiere hierzu alle Bearbeitungszeiten und Übergangszeiten. Damit können wieder die oben genannten Fallunterscheidungen angewendet werden.

Daß dies auch für mehr als zwei Blöcke gilt, folgt aus der Beobachtung, daß weitere Blöcke aus solchen Aufteilungen induktiv gebildet werden. □

Flexible Prozesse. Diese Beobachtung ist ebenfalls auf beliebige Ablaufmengen übertragbar. Hierzu wird zu jedem Ablauf eine untere Schranke bestimmt und davon das Maximum gewählt. Das Maximum steht für die minimale Anzahl von Teilnehmern, um einen bestimmten Prozeß unter Beibehaltung aller Bedingungen auszuführen. Das Maximum gilt insbesondere für alle anderen Prozeßdefinitionen.

$$\max\{\, anz(akt(\varphi), w_1) \mid \varphi \in Q\}$$

Lokale Bearbeitungszeiten. Etwas komplexer wird die Bestimmung einer unteren Schranke, wenn lokale statt globale Bearbeitungszeiten angewendet werden sollen. Um sicherzugehen, muß für eine Aktivität immer die kürzest mögliche Bearbeitungszeit in die Berechnungen eingehen. Nur so wird die Anzahl von Teilnehmern minimiert. Eine neue Funktion wird informell eingeführt, die zu einer Aktivität a aus einer Menge von Teilnehmertypen W die kürzeste Bearbeitungszeit liefert:

$$bz_{\min} : \mathbb{A} \times \mathcal{P}\,(\mathbb{WFT}) \rightarrow \mathbb{Ti}$$

Damit ändert sich die Berechnung der Blockbearbeitungszeit bbz. Wir wollen zu einer Menge von Aktivitäten die kürzesten Bearbeitungszeiten summieren. Gleichzeitig muß beachtet werden, daß die kleinst mögliche Rüstzeit, die ein Teilnehmertyp w_{rz} beinhaltet, addiert wird.

$$bbz_{\min}(A, W) = \left(\sum_{a \in A} bz_{\min}(a, W)\right) + (|A| - 1) * guz + rz(w_{rz})$$

Vermischter Datenfluß. Der nächste Schritt zur Abschätzung für den vermischten Datenfluß ist nicht mehr groß. Es reicht ein ähnliches Vorgehen wie in Abschnitt 4.2.3.1, in dem es um die Bestimmung einer Lösung zu einem vermischten Datenfluß ging.

Für den rein-vermischten Datenfluß werden die anteilsmäßigen Blockbearbeitungszeiten genommen, addiert und daraus die untere Schranke bestimmt. Beim kombiniert-vermischten Datenfluß werden die anteilsmäßigen unteren Schranken addiert.

Rein-vermischter Datenfluß. Die Blockbearbeitungszeit g wird bezüglich einer Menge von Abläufen Q bestimmt. Da mit lokalen Bearbeitungszeiten gearbeitet wird, werden nur die kürzesten Bearbeitungszeiten mit $bbz_{\min}$ summiert. Die Menge W ist die Menge der Teilnehmertypen. Erst im zweiten Schritt wird die untere Schranke U_1 unter Verwendung der ermittelten Blockbearbeitungszeit g und der maximalen Stabilität $stab_{\max}$ bestimmt.

$$g = \sum_{\varphi \in Q} (at(\varphi) * bbz_{\min}(akt(\varphi), W))$$

$$U_1 = \frac{g * (100 + \nu)}{\mathrm{mT} * stab_{\max}}$$

Kombiniert-vermischter Datenfluß. Für den kombiniert-vermischten Datenfluß steht eine einfachere Abschätzung zur Verfügung, wobei hier ebenfalls $bbz_{\min}$ implizit verwendet wird:

$$U_2 = \sum_{\varphi \in Q} (at(\varphi) * anz(akt(\varphi), W))$$

5.4.2 Untere Schranke für Anzahl Teilnehmer eines bestimmten Typs

Die untere Schranke für einen Teilnehmertyp w wird nach den Aktivitäten bestimmt, die ausschließlich von w angeboten werden. Sie wird besser, je spezifizierter die Fähigkeiten von w sind.

Sei also A_{res} die Menge der vom Problem gestellten Aktivitäten. Sei w ein Teilnehmertyp, für den eine untere Schranke ermittelt werden soll. Dann wird die Menge der Aktivitäten A_w, über die nur w verfügt, wie folgt bestimmt:

$$A_w = A_{\text{res}} \cap akt(w) \setminus \bigcup_{w_i \in W \setminus \{w\}} akt(w_i)$$

Analog zu Kapitel 5.4.1 kann nun die Anzahl von Teilnehmern des Typs w hinsichtlich der Menge von Aktivitäten A_w bestimmt werden. Diesmal darf jedoch die Stabilität und Rüstzeit von w verwendet werden.

Kapitel 6

Problemzerlegung und Suche

In diesem Kapitel werden Heuristiken und Optimierungsstrategien für das Workflow Problem vorgestellt. Aufgrund der Problemdefinition ist es möglich, Zerlegungen auf zwei Datenstrukturen durchzuführen, zum einen auf Finite Domain und zum anderen auf Finite Set Variablen. Beide Möglichkeiten werden miteinander verglichen.

6.1 Hierarchie

Das Lösungsverfahren zum Workflow Problem verwendet eine dreischichtigen Heuristik: Heuristik über die Anzahl von Blöcken, Heuristik über die Workflow Teilnehmertypen und Heuristik über die Aktivitätenzuordnung.

Anzahl Blöcke. In der ersten Teilheuristik wird aufsteigend über die Anzahl der Blöcke iteriert. Es ist nicht von Beginn an klar, über wieviele Blöcke eine optimale Lösung verteilt ist. Es kann vorkommen, daß, obwohl eine erste Lösung mit acht Blöcken gefunden wurde, die optimale genau zehn verwendet.

Workflow Teilnehmer. Die zweite Teilheuristik beschäftigt sich mit der Zuweisung von Teilnehmertypen zu dem jeweiligen Block. Dadurch wird den Blöcken eine obere Schranke auferlegt, die besagt, daß der Block keine Aktivitäten enthalten darf, die nicht zu den Fähigkeiten des Teilnehmertyps gehören.

Aktivitätenzuordnung. Die letzte Teilheuristik weist schließlich jeder nicht-determinierten Aktivität einen Block zu. Hier sind verschiedene Heuristiken sowohl für Finite Domain als auch für Finite Set möglich.

6.2 Workflow Heuristiken

Eine gute Heuristik enumeriert frühzeitig gute Lösungen. Hierzu muß Wissen über das Problem in die Heuristik miteinfließen. In der Hierarchie ist dies auf der Ebene der

Heuristik	Beschreibung
minK	Wähle den Block mit den wenigstmöglichen Teilnehmertypen und weise ihm den günstigsten Teilnehmertyp zu.
maxAkt	Wähle den Block mit den wenigstmöglichen Teilnehmertypen und weise ihm den Teilnehmertyp mit meisten Fähigkeiten zu.
minAkt	Wähle den Block mit den wenigstmöglichen Teilnehmertypen und weise ihm den Teilnehmertyp mit wenigsten Fähigkeiten zu.
random	Wähle Block und Teilnehmertyp zufällig.

Tabelle 6.1: Spezielle Workflow Teilnehmer Heuristiken

Zuordnung von Workflow Teilnehmern und der Aktivitäten möglich. Auf der Ebene der Blockanzahl ist keine bessere Heuristik als `naive` bekannt.

6.2.1 Workflow Teilnehmer

So stellen sich folgende Fragen bezüglich der Heuristik für Workflow Teilnehmer:

1. Welchem Block soll zuerst ein Workflow Teilnehmertyp zugewiesen werden?
2. Welcher der möglichen Workflow Teilnehmertypen soll zugewiesen werden?

Für die Blockauswahl liegt kein spezielles Wissen vor. Deswegen ist die allgemeine Heuristik *First Fail* (siehe Abschnitt 3.3.1.1) vorzuziehen. Sie verhindert Flaschenhalsprobleme, indem immer der Block mit den wenigsten Zuweisungsmöglichkeiten gewählt wird.

Die Wahl des Teilnehmertyps für einen ausgesuchten Block kann nach verschiedenen Strategien erfolgen. Da die Kosten möglichst gering gehalten werden sollen, kommen zunächst die günstigsten Teilnehmer in Frage *(minK)*. Liegen die Ressourcen in der Eingabe aufsteigend sortiert nach ihren Kosten vor, entspricht *minK* der Heuristik *First Fail*. Eine andere Strategie begutachtet die Fähigkeiten möglicher Teilnehmertypen. Die Wahl eines Teilnehmers mit wenigen Fähigkeiten *(minAkt)* hat das Ziel, die Zuweisungsmöglichkeiten der Aktivitäten weiter einzuschränken. Die entgegengesetzte Strategie *maxAkt* wiederum bietet große Freiheiten, in der Hoffnung, durch wenige Restriktionen eine schnelle Lösung zu finden.

Tabelle 6.1 listet alle wesentlichen Heuristiken auf. Hierbei ist nur *minK* kostenorientiert. Die Heuristik *random* ist zu den allgemeinen Heuristiken zu zählen. Dennoch kann ihr Einsatz von Nutzen sein, wenn der Suchraum sehr groß ist und keine schnelle Lösung auf Anhieb gefunden wird.

Heuristik	Beschreibung
maxBzK	Wähle die längste Aktivität und weise sie dem Block mit dem günstigsten Teilnehmertyp zu.
minBzK	Wähle die kürzeste Aktivität und weise sie dem Block mit dem günstigsten Teilnehmertyp zu.
maxBzS	Wähle die längste Aktivität und weise sie dem Block mit dem größten Zeitfenster zu.
minBzS	Wähle die kürzeste Aktivität und weise sie dem Block mit dem größten Zeitfenster zu.
random	Wähle Aktivität und Block zufällig.

Tabelle 6.2: Spezielle Aktivität Heuristiken

6.2.2 Aktivitäten

Aus Sicht der Aktivitätenzuordnung müssen folgende Fragen beantwortet werden:

1. Welche Aktivität soll zuerst zugewiesen werden?
2. Welchem Block soll Aktivität zugewiesen werden?

Das primäre, übergeordnete Kriterium bilden die Kosten. Die Lösung muß billig sein. Desweiteren muß eine frühzeitige Abschätzung der Kosten vorhanden sein. Eine frühzeitige Abschätzung gewährleistet die Wahl der längsten Aktivität *(maxBz)*, da die Bearbeitungszeit direkten Einfluß auf die Anzahl der Teilnehmer und damit auf die Kosten hat. Je größer die Bearbeitungszeit, desto größer die Kosten. Im Gegensatz dazu wählt das Ordnungskriterium *minBz* die kürzeste Aktivität. Hier sollte erwartungsgemäß wenig Propagierung angeregt werden.

Eine kostengünstige Zuweisung der gewählten Aktivität erfüllt die Strategie *maxBzK*. Sie wählt für die Aktivität den Block mit dem günstigsten Teilnehmertyp. Einen anderen Ansatz verfolgt die Strategie *maxBzS*. Sie weist einer Aktivität den Block mit dem größten relativen Zeitfenster zu. Ziel ist, die Aktivitäten so auf die Blöcke zu verteilen, daß möglichst wenige Teilnehmer erforderlich werden. Die Strategie *maxBzS* beachtet bedingt die Kosten der Teilnehmertypen. Eine Aktivität wird nur dann einem günstigen Teilnehmer zugewiesen, wenn alle möglichen relativen Zeitfenster zu klein sind. Entgegengesetzt arbeiten die Heuristiken *minBzK* und *minBzS*, die jeweils die kürzeste Aktivität zuerst wählen.

Die Tabelle 6.2 listet alle wesentlichen Heuristiken für Aktivitäten auf. Die wichtigste Heuristik ist *maxBzK*. Sie kombiniert eine schnelle Approximation mit einer günstigen Lösungskonstruktion.

6.3 Optimierung

Besteht die Aufgabe darin, die beste Lösung zu finden, muß dem System eine Bewertungsfunktion mitgeteilt werden, welche über die Optimalität entscheidet. Hierzu eignet sich die Kostenfunktion *kost*. Die nachfolgende Diskussion stellte einige Optimierungsstrategien, die auf Branch & Bound (siehe Abschnitt 3.4.2) zurückgehen, vor.

Nur Kosten (k). Die einfachste Optimierung besteht darin, allein die Kosten zu minimieren. Dies wird durch die folgende Funktion ausgedrückt, wobei s die aktuell günstigste Lösung repräsentiert:

$$kost(s) > kost(s'). \tag{6.1}$$

Kosten und Takt addieren (kAT). Für das Workflow Problem reicht die Optimierungsfunktion in (6.1) aus. Interessant ist es, sowohl die Kosten als auch den Takt zu optimieren. Dabei sollen die Kosten Präferenz haben. Eine neue Lösung ist folglich dann besser, wenn sie entweder günstiger ist, oder bei gleichen Kosten einen niedrigeren Takt aufweist. Diese Regel kann auf zwei Arten implementiert werden, wobei jede ihre Nachteile hat. Eine Möglichkeit besteht darin, Kosten und Takt zu addieren. Dies ergibt folgende Optimierungsfunktion:

$$kost(s) + takt(s) > kost(s') + takt(s') \tag{6.2}$$

Der Nachteil dieser Funktion ist die Vermischung der Kosten mit dem Takt. Dies führt zu unerwünschten Ergebnissen. Hat beispielsweise eine Lösung s_1 die Kosten $kost(s_1) = 1000$ und einen Takt von $takt(s_1) = 400$, und eine Lösung s_2 die Kosten $kost(s_2) = 950$ und einen Takt von $takt(s_2) = 500$. Dann führt die Optimierungsfunktion zu folgenden Ergebnissen:

$$kost(s_1) + takt(s_1) = 1400$$
$$kost(s_2) + takt(s_2) = 1500$$

Obwohl die zweite Lösung günstiger ist, fällt ihre Bewertung aufgrund des hohen Taktes schlechter aus. Dies widerspricht der geforderten Regel. Dennoch ist die Optimierungsfunktion unter einer Einschränkung einsetzbar. Dazu werden die niederwertigen Stellen der Kosten für die Taktzahl "reserviert". Sei x die Stelligkeit von mT. Dann gilt für alle Lösungen s die Reservierung

$$kost(s) \mod 10^x = 0.$$

Man beachte, daß diese Bedingung für alle Kosten. d.h. Teilnehmer-, Block-, Werkzeugkosten etc., gelten muß.

Kosten und Takt addieren (kDT). Die zweite Methode beruht auf einer Disjunktion. Sie arbeitet zwar korrekt, regt aber aufgrund eines schwachen Branch & Bound Constraints wenig Propagierung an. Die Disjunktion äußert sich in der Unterscheidung zwischen zwei Lösungen mit ungleichen Kosten und zwei Lösungen mit gleichen Kosten aber unterschiedlichem Takt. Die mangelnde Propagierung resultiert somit aus der zugelassenen Kostengleichheit.

$$
\begin{aligned}
\mathsf{K}_1 &= \begin{cases} 1 & \text{falls } \mathit{kost}(s) = \mathit{kost}(s'), \\ 0 & \text{sonst} \end{cases} \\
\mathsf{K}_2 &= \begin{cases} 1 & \text{falls } \mathit{kost}(s) > \mathit{kost}(s'), \\ 0 & \text{sonst} \end{cases} \\
\mathsf{T} &= \begin{cases} 1 & \text{falls } \mathit{takt}(s) > \mathit{takt}(s'), \\ 0 & \text{sonst} \end{cases} \\
\mathsf{D} &= \mathsf{K1} \wedge \mathsf{T} \\
\mathsf{K}_2 + \mathsf{D} &=: 1
\end{aligned}
\tag{6.3}
$$

Nur Takt (t). Die oben genannten Optimierungsfunktionen führen zu Lösungen mit monoton fallenden Kosten. Auf der anderen Seite kann es auch Anwendungen geben, die nur den Takt optimieren. Ähnlich wie bei den Kosten, treten auch beim Takt zwei Optimierungsmöglichkeiten auf. Erstere (6.4) setzt auf einen streng monoton fallenden Takt.

$$
\mathit{takt}(s) > \mathit{takt}(s') \tag{6.4}
$$

Nur Takt bei fixen Kosten (tFK). Letztere optimiert den Takt einer Lösung mit fixen Kosten. Die Funktion ist ähnlich wie (6.3) definiert, was wiederum eine schlechte Propagierung nach sich zieht.

$$
\begin{aligned}
\mathsf{K} &= \begin{cases} 1 & \text{falls } \mathit{kost}(s) = \mathit{kost}(s'), \\ 0 & \text{sonst} \end{cases} \\
\mathsf{T} &= \begin{cases} 1 & \text{falls } \mathit{takt}(s) > \mathit{takt}(s'), \\ 0 & \text{sonst} \end{cases} \\
\mathsf{K} + \mathsf{T} &=: 1
\end{aligned}
\tag{6.5}
$$

Die Wahl der Optimierungsfunktion sollte nach folgenden Überlegungen getroffen werden. Sollen nur die Kosten oder nur der Takt optimiert werden, so sind die Optimierungen (6.1) und (6.4) vorzuziehen. Besteht die Anforderung darin, beides zu optimieren, so ist Funktion (6.2) einzusetzen, falls die Vorbedingungen erfüllbar sind. Sind die Vorbedingungen unerfüllbar, dann kann entweder Optimierung (6.3) oder eine Kombination aus (6.1) und (6.5) benutzt werden. Letzteres ist eine zweigeteilte Suche. Zunächst wird hinreichend nach einer günstigen Lösung gesucht. Anschließend werden die Kosten der Lösung als Parameter dem zweiten Suchvorgang mitgegeben.

6.3.1 Weitere Bewertungsfunktionen

Es gibt auch andere Bewertungsfunktionen, die in diesem Abschnitt nur theoretisch vorgestellt werden. In der Praxis wurden sie nicht getestet.

Bei der Suche nach einer optimalen Lösung werden hauptsächlich nur die Kosten betrachtet. Alternativ kann man überlegen, wieviele Aktivitäten optimal auf Teilnehmertypen verteilt sind. In diesem Zusammenhang heißt *optimal*, daß diese Aktivität auf keinem anderen Teilnehmertyp weniger Kosten verursacht.

Definition 6.3.1. Eine Funktion $\lambda : \mathbb{A} \times \mathbb{WFT} \times \mathbb{WFT} \to \mathbb{WFT}$ vergleicht zwei Teilnehmertypen in Bezug auf eine Aktivität und gibt denjenigen zurück, der besser geeignet ist.

Eine Funktion *isOpt* : $\mathbb{A} \times \mathbb{WFT} \times \mathcal{P}(\mathbb{WFT}) \to \{0, 1\}$ testet, ob ein Teilnehmertyp im Vergleich zu anderen Typen optimal bezüglich einer Aktivität ist.

Sei λ eine beliebige Vergleichsfunktion, a eine Aktivität und $W \in \mathcal{P}(\mathbb{WFT})$ eine Menge von Teilnehmertypen mit der Eigenschaft: $\forall w \in W :\ a \in akt(w)$. Dann lautet die Abbildungsvorschrift von *isOpt* für ein $w \in W$

$$isOpt(a, w, W \setminus \{w\}) \begin{cases} 1 & \text{falls } \forall w_j \in W :\ \lambda(a, w, w_j) = w, \\ 0 & \text{sonst} \end{cases}$$

Optimierung. Damit ist die Optimierungsfunktion für Branch & Bound eine Maximierung der optimalen Verteilung aller Aktivitäten. Die Optimierungsfunktion hat für Branch & Bound folgende Form, wobei $l = (n, S, Ws, Ms, eT, K_G)$ die aktuell beste Lösung, $l' = (n', S', Ws', Ms', eT', K'_G)$ die nächstbessere Lösung, und W die Menge der vorliegenden Teilnehmertypen ist:

$$\sum_{i=1}^{n} \sum_{a \in S_i} isOpt(a, Ws_i, W) < \sum_{i=1}^{n'} \sum_{a' \in S'_i} isOpt(a', Ws'_i, W) \qquad (6.6)$$

Vergleich. Die Ermittlung einer optimalen Zuteilung setzt sich aus den Faktoren Bearbeitungszeit, Stabilität und Teilnehmertypkosten zusammen. Eine Aktivität sollte von einem Teilnehmertyp bereitgestellt werden, wenn

- er sie am schnellsten ausführen kann,
- er sie mit größter Stabilität ausführen kann,
- er für sie der günstigste ist.

Eine Vergleichsfunktion λ sollte daher alle drei Punkte in die Ermittlungen einfließen lassen. Der Vergleich der Ressourcenkosten alleine ist leicht. Ebenso der Vergleich der lokalen Bearbeitungszeiten. Eine Verknüpfung der Kosten mit den Bearbeitungszeiten führt zu folgender Vergleichfunktion (eine Erweiterung um den Faktor Stabilität erfolgt nach dem gleichen Muster):

$$\lambda(a, w, w_j) = \begin{cases} w & \text{falls } \frac{kost(w)}{kost(w_j)} * \frac{bz(w,a)}{bz(w_j,a)} \leq 1 \\ w_j & \text{sonst} \end{cases} \qquad (6.7)$$

Nachteil. Der Nachteil dieser Strategie ist, daß eine einzige falsche Zuweisung einer langen Aktivität zu schlechten Ergebnissen führt. Der Extremfall besteht aus einer Aufgabenstellung mit vielen, kurzen Aktivitäten und einer im Verhältnis lang andauernden Aktivität. Dabei ist es denkbar, in eine Situation zu geraten, in der alle kurzen Aktivitäten optimal positioniert sind, und trotzdem eine Lösung mit hohen Kosten erzeugen. Die Ursache hierfür ist die lange Aktivität, die keine optimale Position einnimmt und hohe Kosten verursacht. Möglicherweise ist es besser, der langen Aktivität den Vorzug zu geben und auf einige optimale Positionen der kurzen Aktivitäten zu verzichten. Doch dies widerspricht der Optimalitätsfunktion (6.6).

6.3.2 Slack

Die Komplexität des Problems resultiert aus der Anzahl der verschiedenen Teilnehmer und der Anzahl benutzter Aktivitäten. Je mehr Teilnehmer zur Auswahl und je mehr Aktivitäten zur Zuordnung gegeben sind, desto länger dauert die Suche aufgrund der Tatsache, daß der Suchbaum tiefer und somit auch größer ausfällt.

Es stellt sich die Frage, wie der Suchbaum zusätzlich zu den Constraints auf der Ebene der Zerlegung und der Suche reduziert werden kann. Eine Möglichkeit bietet der folgende *Slack* Algorithmus.

Die interessanteste Größe sind die Kosten. Eine offene Frage ist, ab welchem Zeitpunkt im Suchbaum sich die Kosten nicht mehr ändern, obwohl noch nicht alle Aktivitätszuweisungen determiniert sind. Die Kosten hängen von der Anzahl und vom Teilnehmertyp ab. Der Typ wird durch eine übergeordnete Heuristik ausgesucht. Bleibt also die Anzahl. In die Berechnung der Anzahl geht u.a. die Blockbearbeitungszeit mit ein, die aus den Bearbeitungszeiten der einzelnen Aktivitäten, den Übergangszeiten und der Rüstzeit besteht. Je mehr Aktivitäten in einem Block enthalten sind, desto größer wird die Blockbearbeitungszeit und desto mehr Teilnehmer werden benötigt, um die Produktion in der vorgegebenen, maximalen Taktzeit zu garantieren.

In jedem Zerlegungsschritt wird pro Block zusätzlich das *relative Zeitfenster* berechnet (siehe Abschnitt 4.1.4.2). Ist die Summe der Bearbeitungszeiten aller nicht-determinierten Aktivitäten kleiner als das kleinste relative Zeitfenster, dann ist es unwesentlich, wie die Aktivitäten zugeteilt werden, da sich an der Anzahl der Teilnehmer nichts ändert. Die Suche kann in diesem Teilbaum abgebrochen werden. Diese Abfrage ist die zentrale Idee des Slack Algorithmus, wobei die Summe der Bearbeitungszeiten der *Slack* ist.

6.3.2.1 Effizienz

Die Effizienz des Slack Algorithmus hängt von den Bearbeitungszeiten der Aktivitäten und dem absoluten Zeitfenster ab. Je größer die absoluten Zeitfenster, desto größer können die relativen Zeitfenster ausfallen. Dies erlaubt einen größeren Slack, der mehr Aktivitäten aufzunehmen vermag, d.h. mehr Aktivitäten sind nicht-determiniert und ein möglicher Abbruch der Suche in diesem Teilbaum kann zu einem früheren Zeitpunkt erfolgen. Ähnliches gilt für kurze Aktivitäten. Je kürzer ihre Bearbeitungszeit, desto mehr Aktivitäten passen in den Slack.

Es ist möglich, eine Abschätzung der maximal ersparten Tiefenschritte anzugeben. Da neben den Bearbeitungszeiten auch Übergangszeiten anfallen, reicht eine einfache Division.

$$\text{maxSlackDepth} = \left\lfloor \frac{absZ(\text{mT}, \text{st}, \nu)}{guz} \right\rfloor$$

Beispiel 6.3.2. In Anlehnung an das Beispiel 4.1.25 auf Seite 48 kann man die maximale Slacktiefe bei einer globalen Übergangszeit von 80 Zeiteinheiten angeben. Für die Ressource mit der Stabilität von 95% gilt die maximale Slacktiefe $\lfloor 488/80 \rfloor = 6$, für die Ressource mit 80% Stabilität nur $\lfloor 411/80 = 5 \rfloor$ Tiefenschritte.

In Anbetracht der Tatsache, daß große Problemstellungen mit vielen Aktivitäten und engen, maximalen Taktzeiten einen sehr tiefen Suchbaum gestalten, bietet der Slack Algorithmus in diesen Fällen keine Verbesserung. Vielmehr muß darauf geachtet werden, daß die zusätzlichen Berechnungen keinen Overhead verursachen. Eine optimale Funktionsweise des Slack Algorithmus erhält man in Verbindung mit der Heuristik *maxBzK* (siehe Abschnitt 6.2). Sie enumeriert die kürzesten Aktivitäten zuletzt.

6.3.2.2 Probleme

Der Slack Algorithmus kann nicht in Kombination mit Taktoptimierungsstrategien verwendet werden, da die maximale Taktzeit im schlechtesten Fall erst im letzten Tiefenschritt determiniert.

Darüber hinaus weist der bisherige Slack Algorithmus zwei Probleme auf:

Es wird keine Lösung gefunden. Der erste Fall kann dann eintreten, wenn sich ein Slack ergibt, der kleiner als das kleinste Zeitfenster ist und noch nicht-determinierte Variablen existieren. Es wird demnach immer abgebrochen. Der Grund liegt in der falschen Annahme, daß nach einer noch besseren Lösung gesucht wird, nicht wissend, daß noch keine existiert.

Die gefundene Lösung ist nicht optimal. Der zweite Fall ähnelt dem Ersten. Eine Lösung wird gefunden und als Referenzlösung angegeben. Das Problem ist jedoch, daß eventuell zu Beginn der Suche eine potentiell bessere Lösung übersehen wurde, weil die Unterschreitung der Slackgrenze die Suche in diesen Ästen abgebrochen hat.

Beide Probleme werden gelöst, indem eine Kontrollvariable mitgeführt wird. Die Suche beginnt ohne den Slack Algorithmus. Bei Auftreten einer Lösung wird die Variable als Trigger auf 1 gesetzt, und zeigt der Suchmaschine an, daß fortan der Slack Algorithmus einzusetzen ist.

6.4 Finite Set Heuristiken

Die mathematische Formulierung der Workflow Problemstellung in Kapitel 4 erlaubt, Heuristiken über Mengen einzusetzen. Gefragt sind Zerlegungsconstraints, die bei Finite Sets die Form $n \in S_i \wedge n \notin S_i$ haben. Diese werden, bezogen auf das Workflow Problem, auf eine Partition angewendet. Um eine gute Heuristik zu finden, müssen folgende Fragen beantwortet werden:

- Welche Menge S_i ist aus der Partition S zu wählen?
- Welche nicht-determinierte Aktivität wird S_i zugewiesen?

Die Beantwortung dieser Fragen ist die Bestimmung drei wichtiger Kriterien: *Weight*, *Order* und *Elements*.

Weight. Welches Gewicht haben die Elemente einer Menge? Im Workflow Problem entspricht das Gewicht einer Aktivität ihrer globalen Bearbeitungszeit.

Order. Welche Menge aus S ist für den nächsten Zerlegungsschritt zu wählen? Die Wahl hängt von drei Faktoren ab:

Select Set Wähle die Menge mit dem kleinsten bzw. mit dem größten *Cost Set* Faktor.

Cost Set Der *Cost Set* Faktor ist entweder die Kardinalität einer Menge oder ein Gewicht. Lezteres wird unterschieden in: minimales bzw. maximales Element und gewichtete Summe der Menge. Der Faktor bezieht sich auf die Komponente der Menge.

Component Zu jeder Finite Set Variable sind drei Mengen bekannt: untere Schranke, obere Schranke und die Menge der unbekannten Elemente.

Beispiel 6.4.1. Angenommen, das Kriterium *Weight* führt die globale Bearbeitungszeit als Gewicht für jede Aktivität ein. Dann bedeutet `order(sel:max cost: weightMax comp:lowerBound`, daß im nächsten Zerlegungsschritt die Menge mit der längsten, determinierten Aktivität gewählt wird.

Elements. Welche noch nicht-determinierte Aktivität soll in S_i enthalten sein? Die Wahl des Elements hängt von zwei Faktoren ab:

Select Element Wähle Aktivität mit kleinstem bzw. maximalem *Element Order* Faktor.

Element Order Element Order ist entweder die Aktivität selbst oder ihre Gewichtung.

Beispiel 6.4.2. An das vorherige Beispiel anknüpfend würde `element(sel: max wrt: weight)` der Menge S_i als nächstes die längste, noch nicht-determinierte Aktivität, die in der oberen Schranke von S_i enthalten ist, zuweisen.

Allein aus diesen drei Kriterien sind viele Kombinationsmöglichkeiten für eine Heuristik denkbar. Eine Formulierung der FD Heuristik *maxBzK* mit den vorgestellten FS Kriterien ist nicht möglich, da zwei Gewichtungen vorliegen müßten, nämlich die der Teilnehmertypkosten und die der Bearbeitungszeiten. Eine ähnliche FS Heuristik würde lauten:

> *Bestimme die Mengen in* S*, die die längste nicht-determinierte Aktivität in ihrer oberen Schranke enthalten. Wähle die Menge, die dem günstigsten Teilnehmertyp unterliegt.*

Im schlechtesten Fall haben alle Mengen in S die längste Aktivität in ihrer oberen Schranke.

Tatsächlich ist keine zielgerichtete FS Heuristik mit den vorgestellten Kriterien für das Workflow Problem bekannt. Die Kardinalität der Blöcke und die Bearbeitungszeiten der Aktivitäten reichen als Vorwissen nicht aus. Ohne das Wissen über die Kosten der Teilnehmertypen ist keine treffende Aussage zum Invest möglich. Diese Beobachtung bestätigen empirische Tests.

Kapitel 7

Implementierung

In diesem Kapitel wird die Implementierung der Software *Woop (Workflow Optimizer)* [16] skizziert. Zunächst wird der Gesamtaufbau beschrieben. Anschließend wird auf den Kern dieser Arbeit und der Implementierung, das Skript, eingegangen. Es wird beschrieben, welche Vorzüge die Abstraktion 'Skript' besitzt, und wie sie hier eingesetzt werden. Zum Schluß wird die Implementierung der verwendeten Datenstrukturen erläutert.

Woop [16] wurde in Mozart/Oz [23] implementiert.

7.1 Aufbau von Woop

Woop ist grob in sieben Gruppen eingeteilt (siehe auch Abbildung 7.1). Jede Gruppe enthält einen oder mehrere Funktoren, die gemeinsam eine übergeordnete Aufgabe erfüllen. Grundlegend für alle Gruppen ist die Gruppe der *Strukturen*.

Hauptmodul. Wird die Software über die Kommandozeile gestartet, so überprüft das Hauptmodul zunächst alle Parameter, initialisiert die Grunddatenstrukturen und ruft den Frontend auf. Beim Start über den Desktop werden Standardparameter eingesetzt.

Frontend. Die Frontend Gruppe besteht aus mehrere Funktoren, die die Interaktionen des Anwenders steuern und kontrollieren.

GUI. Die GUI Gruppe enthält Funktoren, die für graphische Elemente zuständig sind. Zur Zeit wird das Tk Toolset unterstützt. Aufgrund der Modularchitektur ist ein Austausch zu Gunsten eines anderen GUI Systems, beispielsweise GTk, ohne Veränderung der anderen Gruppen möglich.

Kalkulator. Diese Gruppe besteht aus Funktoren, die hauptsächlich Formeln beinhalten. Sie berechnen aus einer eingegangenen Lösung weitere Daten. Darüber hinaus werden Schranken vorberechnet. Zusätzlich existiert ein Funktor, der Eingangsdaten und per Hand eingegebene Lösungen verifiziert.

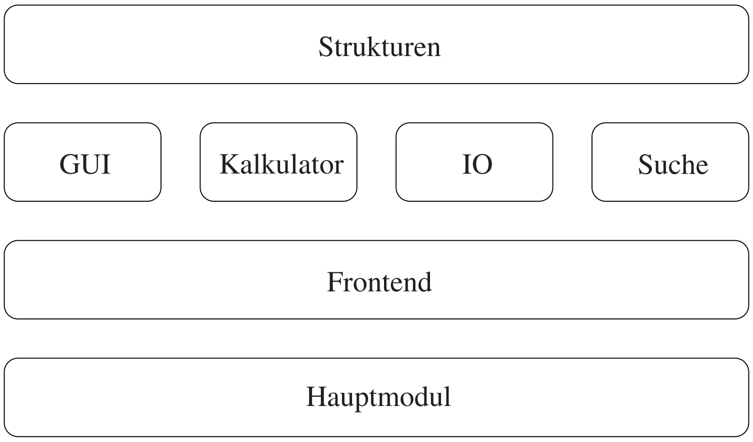


Abbildung 7.1: Aufbauskizze von Woop

IO. Die IO Gruppe importiert Eingangsdaten und exportiert Lösungen ins ASCII Format. Zusätzlich besteht die Möglichkeit, Lösung sofort nach ihrem Auftreten per Mail zu verschicken, bzw. in einer Datei zu exportieren.

Suche. In dieser Gruppe befinden sich zwei Funktoren, einer für das Suchskript, und und einer für die Controller der Suchmaschinen. Darüber hinaus enthält die Gruppe eine Applikation, die als ein Frontend zum Erstellen eigener FD Heuristiken eingesetzt wird.

Strukturen. Hier handelt es sich um einen einzelnen Funktor, der alle verwendeten Strukturen verwaltet.

7.2 Skript

Ein *Skript* ist ein Programm zur Berechnung von Lösungen. Es enthält ein Modell und eine Heuristik zu einer vorgegebenen Problemstellung. In Mozart/Oz ist ein Skript eine Prozedur mit einer Rückgabevariable, die nach Konvention `Root` genannt wird. Sie enthält eine Lösung.

```
proc {Script Root}
  %% Deklaration von Variablen
in
  %% Angabe von Constraints
  %% Angabe einer Heuristik
end
```

Die Ausführung eines Skripts übernimmt eine Suchmaschine, die die entsprechenden Propagierer und Zerlegungsalgorithmen implementiert. Damit ist ein Skript eine sinnvolle Abstraktion, die folgende Punkte fördert: dynamische Skriptgenerierung, Kontrolle über Constraintgruppen und dynamische Heuristikzusammensetzung.

Algorithmus 1 Dynamische Skriptgenerierung

```
functor
  export
    getScript: GetScript
define
  fun {GetScript Args}
    %% Deklaration von Variablen
    %% Wahl: Heuristik/Optimierung
    H_Akt = Args.h.akt
    H_Wft = Args.h.wft
    Opt   = Args.h.opt
    %% Trigger
    Trigger = Args.trigger
  in
    functor
      export
        script: Script
        order:  Order
    define
      <Fil_P> <Ord_P> <Sel_P> <Val_P> <Oo>
      <Script>

      Order = Oo.Opt
    end
  end
end
```

Dynamisch Skriptgenerierung. Aus einer Applikation heraus, wird ein Skript eingebettet in einem Funktor an eine Suchmaschine weitergereicht. Dieser Funktor erwartet ein Feld `script`, über das das Skript exportiert wird, und eventuell ein Feld *order*, über das die Optimierungsfunktion nach Außen bekannt wird.

Woop sieht eine weitere Einbettung des Skriptfunktors vor (siehe Algorithmus 1). Hierzu wird ein Hauptfunktor eingeführt, der eine Funktion `GetScript` exportiert. Diese wird von der Applikation zusammen mit dem Argument `Args` aufgerufen. `Args` enthält alle Daten, die die Problemstellung, die angewendeten Heuristiken und die Kontrollschalter beschreiben. Das Resultat der Funktion `GetScript` ist ein mit den neuen Daten aufbereiteter Skriptfunktor.

Kontrolle über Constraintgruppen. Die Kontrolle über Constraintgruppen eignet sich insbesondere zur Einblendung bzw. Ausblendung bestimmter Berechnungen. So ist es sinnvoll, die Berechnung von lokalen Zeiten auszublenden, wenn globale Zeiten gewählt wurden. Ähnlich verhält es sich mit der Auswahl des geeigneten Datenflusses.

Eine solche Constraintgruppe wird von einem Trigger (boolsche Variable) bewacht und hat die Form:

```
if Trigger.xGruppe then
  <Constraints>
end
```

Der Wert des Triggers wird in der Applikation festgelegt und zusammen mit `Args` übergeben.

Dieser Mechanismus erlaubt zudem ein bequemes Testen und Verifizieren des Skripts, ohne zuvor den Funktor zu kompilieren und die Applikation neu zu starten.

7.2.1 Dynamische Heuristikzusammensetzung

In Mozart/Oz besteht eine Heuristik aus den Kriterien *Filter, Order, Select* und *Value*. Zu jedem Kriterium gibt es eine Auswahl an vordefinierten Prozeduren. Zusätzlich besteht die Möglichkeit, eigene Prozeduren einzubinden, und damit eigene Heuristiken anzugeben. Die Schnittstelle ist {`FD.distribute Heur Xv`}. Hierbei repräsentiert `Heur` eine Heuristik und `Xv` einen Vektor von Finite Domain Variablen. Eine generische Heuristik wird über ein Record angegeben:

```
generic(order:     +Order
        filter:    +Filter
        select:    +Select
        value:     +Value)
```

Woop implementiert einen flexibleren, speziell auf die Anwendung zugeschnittenen Ansatz, der unter anderem den Slack Algorithmus ermöglicht. Hierzu ist es notwendig, die Daten der Rootvariable mitzuführen.

Somit hat jedes Kriterium den vollen Zugriff auf die Rootvariable. Am Beispiel von *Value* sieht eine Ansammlung möglicher Prozeduren wie folgt aus:

```
Val_P = o(´minK´: proc {$ Args}
                    Sel  = Args.sel
                    Root = Args.root
                  in ... end
           ...)
```

Jede Prozedur ist unter ihrem entsprechenden Namen abgelegt. Analog ist `Fil_P` für Filter, `Ord_P` für Order und `Sel_P` für Select aufgebaut.

Woop übergibt an das Skript lediglich die Namen der Kriterien, nicht aber deren Prozeduren. Diese werden erst zur Laufzeit ermittelt und zu einer Heuristik zusammengesetzt. Im nachfolgenden Beispiel ist `H_Akt` ein Record, dessen Felder die Namen der zu verwendeten Prozeduren enthalten:

```
Gen = generic(filter: Fil_P.(H_Akt.filter)
              order:  Ord_P.(H_Akt.order)
              select: Sel_P.(H_Akt.select)
              value:  Val_P.(H_Akt.value))
```

Da die Argumente der Prozeduren in `Gen` nicht zur `FD.distribute` Schnittstelle passen, wird eine neue Zerlegungsprozedur definiert. Diese wird erst zur Laufzeit generiert. Algorithmus 2 skizziert den generischen Mechanismus. Die Funktion `GetDistr` gibt eine Zerlegungsprozedur `Distr` zurück. Der Parameter `Gen` enthält

Algorithmus 2 Zerlegung

```
fun {GetDistr Gen Vec}

  proc {Distr Root}
    {Space.waitStable}
    Fil = {Gen.filter fil(root: Root vec:   Vec)}
  in
    if {IsAtom Fil} then skip else
      case Fil
      of C then
        Ord = {Gen.order  ord(root: Root fil: Fil)}
        Sel = {Gen.select sel(root: Root ord: Ord)}
        Val = {Gen.value  val(root: Root sel: Sel)}
      in
        choice Sel = Val {Distr {Adjoin Root r(Vec: C)}}
        []     Sel\=:Val {Distr {Adjoin Root r(Vec: Fil)}}
        end
      end
    end
  end
in
  Distr
end
```

die Heuristik. Dagegen vermerkt der Parameter `Vec`, ob es sich um eine Zerlegung für Workflow Teilnehmer oder für Aktivitäten handelt.

Die Zerlegungsprozedur bekommt beim Aufruf die Root Variable des Skripts als Argument. Erst mit `Root.Vec` wird der Vektor bestimmt, auf dem operiert wird. Nachdem die Propagierung einen Fixpunkt erreicht hat, werden nichtdeterminierte FD Variablen herausgefiltert. Anschließend wird eine Variable bestimmt, die einen Wert erhalten soll. Die Zerlegung findet im `choice` Block statt.

7.3 Datenstrukturen

In diesem Abschnitt werden die in der Implementierung verwendeten Datenstrukturen vorgestellt. Hierbei wird sowohl auf die Finite Domain [30] als auch auf die Finite Set [21] Bibliothek zugegriffen. Die Implementierung ist eine direkte Umsetzung des mathematischen Modells.

Mozart benutzt für Finite Domains und Finite Sets positive ganze Zahlen. Der größte Wert für eine FD Variable wird durch `FD.sup` repräsentiert. Eine FD Variable, die weder eine Begrenzung nach unten noch nach oben hat, bekommt den gesamten Bereich zugewiesen. Dies drückt `FD.decl` aus.

Wir gehen davon aus, daß ein Problem $\gamma = (Q, W, \mathrm{mT}, \nu, \mathrm{guz})$ gemäß Abschnitt 4.1.7 gestellt ist. Alle Aktivitäten und alle Teilnehmertypen werden durch positive, ganze

Zahlen dargestellt. Die Menge aller verwendeten Aktivitäten ist `Akts`$=$ *akt*(*res*(Q)). Im folgenden wird `Akts` als eine Liste von Aktivitäten interpretiert.

Ablauf. Ein Ablauf ist eine festgelegt Record Struktur. Sowohl die Aktivitäten als auch die Vorrangsrelationen werden in einer Liste verwaltet.

Anzahl Blöcke. Die Anzahl der Blöcke wird durch die FD Variable `B` repräsentiert. Die obere Schranke für die Anzahl von Blöcken beträgt gemäß Filter 5.1.1 die Anzahl verwendeter Aktivitäten `AA = {Length Akts}`.

```
B = {FD.int 1#AA}
```

Partition. Die Partition ist ein Tupel der Länge `B`, dessen Komponenten FS Variablen sind. Die obere Schranke für jede FS Variable ist die Menge aller verwendeten Aktivitäten `Akts` (siehe auch Definition 4.1.9).

```
 S = {FS.var.tuple.upperBound B Akts}
{FS.partition S {FS.value.make Akts}}
```

Aktivitätzuordnung. Ein wesentliches Problem von `FS.partition` ist, daß der Constraint $a \in S_i \wedge b \in S_j$ mit $1 \leq i \leq j \leq B$ nicht ausgedrückt werden kann. Hierzu wird eine weitere FD Struktur notwendig. Die Implementierung verwendet ein FD Record, dessen Features den Aktivitätsnamen entsprechen. Jede Aktivität wird somit auf einen Block abgebildet.

```
P = {FD.record p Akts 1#B}
```

Damit sieht der oben genannte Constraint wie folgt aus: `P.a=<:P.b`.

Was noch fehlt, ist die Verbindung zwischen der Partition und der Aktivitätzuordnung. Die Aussage ist: eine Aktivität `a` bildet auf Block `i` ab, gdw. die Aktivität `a` in der Menge `S.i` enthalten ist. Die Implementierung verwendet dafür reifizierte Constraints.

```
for Ai in Akts do
  for Bi in 1..B do
    (P.Ai =: Bi) = {FS.reified.isIn Ai S.Bi}
  end
end
```

Teilnehmertypkonfiguration. Die Teilnehmertypkonfiguration wird durch ein FD Tupel der Länge `B` dargestellt (Filter 5.1.3).

```
Ws = {FD.tuple m B 1#AT}
```

Anzahl Teilnehmer. Die Anzahl der Teilnehmer wird durch ein FD Tupel der Länge `B` repräsentiert. Jeder Block besteht aus mindestens einem Teilnehmer. Die Anzahl nach oben ist auf `FD.sup` begrenzt (Filter 5.1.3).

```
Ms = {FD.tuple m B 1#FD.sup}
```

Takt. Die FD Variable für den effektiven Takt ist nach oben durch die vorgegebene, maximale Taktzeit begrenzt (Filter 5.1.4).

```
ET = {FD.int 1#mT}
```

Kosten. Die Kosten werden durch eine nach oben auf `FD.sup` begrenzte FD Variable dargestellt (Filter 5.1.5).

```
GK = {FD.decl}
```

Root Variable. Legt man die vorgestellten Datenstrukturen zusammen, so ergibt sich die folgende Struktur für die Root Variable des Skripts:

```
Root = root(b:B s:S p:P ws:Ws ms:Ms et:ET gk:GK)
```

Kapitel 8

Evaluierung

In diesem Kapitel werden die in den Abschnitten 6.2 und 6.3 vorgestellten Heuristiken und Optimierungsfunktionen anhand eines Szenarios evaluiert. Bei dem Szenario handelt es sich um zwei flexible Geschäftsprozesse (siehe Anhang A.2). Die Heuristiken werden sowohl bezüglich des Grundmodells als auch bezüglich des präzisen Modells bewertet.

Alle ausführlichen Benchmarktabellen sind im Anhang A.3 zu finden. Sie sind wie folgt zu lesen: die Heuristiken für Teilnehmertypen und Aktivitäten werden durch die Punkte *Heur. WFT* und *Heur. Akt.* angegeben. Mit *Opt.* ist die Optimierungsfunktion gemeint. Der Punkt *Kosten* gibt den errechneten Invest und *Takt* die effektive Taktzeit an, wohingegen die Anzahl der Teilnehmer in der Spalte *Anz. WFT* abzulesen ist. Die Lösungen wurden in der Zeitspanne *Zeit* gefunden, wobei insgesamt x *Wahlpunkte* und y *Fehlschläge* aufgetreten sind. Wird nach der optimalen Lösung gesucht, so gibt *Anz. Lös.* die Anzahl der zuvor gefundenen Lösungen an.

Im folgenden wird eine Kombination von Heuristiken durch ein Paar (*Heur. WFT*, *Heur. Akt.*) dargestellt.

8.1 Grundmodell

Die Evaluierung beginnt mit der Untersuchung verschiedener Kombinationen von Heuristiken für Teilnehmertypen und Aktivitäten für das Grundmodell. Dabei steht die Bestimmung der effizientesten Heuristik im Vordergrund. Anschließend werden Workflow Teilnehmertypen mit vielen Fähigkeiten und Problemstellungen mit variabler Blockanzahl untersucht. Um die Ergebnisse bewerten zu können, werden sie mit der besten bekannten Lösung verglichen.

Beste Lösung. Beobachtungen haben ergeben, daß die beste Lösung des Grundmodells aus 10 Blöcken besteht, und keine Ressource des Typs *riesig* enthält. Die Kosten betragen 32 527 000 bei 41 Workflow Teilnehmern. Die nachfolgende Tabelle beschreibt die Infrastruktur. Die Einträge in der *WFT* Zeile enthalten die abgekürzten Teilnehmertypnamen (siehe auch Tabelle A.2 auf Seite 105):

Block	1	2	3	4	5	6	7	8	9	10
WFT	S	g	S	g	m	S	g	m	s	S
Anzahl	1	5	1	7	4	1	10	1	10	1

8.1.1 Workflow Heuristiken

Um der besten Lösung nahe zu kommen, starten die Tests mit einer festgesetzten Zahl von Blöcken (10) und einem festgesetzten Pool von Ressourcen (ohne *riesig*). Die nachfolgenden Tests beziehen sich auf das Grundmodell. Die Benchmarks A.3.1, A.3.2 und A.3.3 auf Seite 105 fassen die ersten Ergebnisse der verschiedenen Heuristiken zusammen. Unterstützt werden sie durch die Abbildungen 8.1 und 8.2.

Aus den Benchmarks geht hervor, daß eine erste Lösung zeitlich schnell und nach wenigen Wahlpunkten gefunden wird. Der Takt liegt meist weit unter der geforderten, maximalen Taktzeit von 540. Dies liegt daran, daß die Zahl der Workflow Teilnehmer im Vergleich zur besten Lösung mit bis zu 4 Teilnehmern mehr relativ hoch ist. Lediglich der Benchmark mit den Heuristiken *(minK, minBzK)* kann sehr schnell eine erste Lösung mit geringen Kosten und wenigen Teilnehmern angeben.

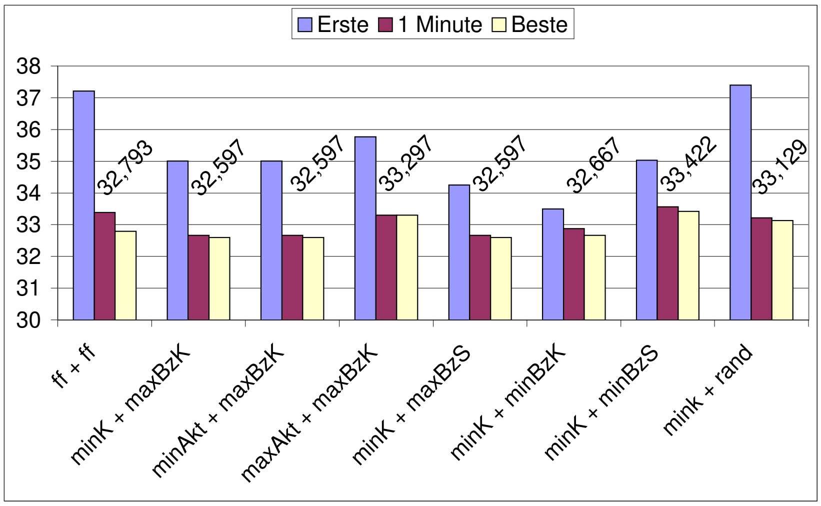


Abbildung 8.1: *Kostenvergleich:* Erste Lösung, beste Lösung nach einer Minute und beste Lösung nach einer Stunde. Kostenangaben beziehen sich auf die beste Lösung nach einer Stunde.

Innerhalb nur einer Minute verbesern sich die Ergebnisse deutlich. Die Kosten liegen hier bei ca. 33 Mio. Bessere Heuristiken, wie zum Beispiel *(minK, maxBzK)*, *(minAkt, maxBzK)* und *(minK,maxBzS)* liegen nahe am bekannten Optimum.

Hierbei fällt auf, daß die ersten beiden Kombinationen gleiche Ergebnisse erzielen. Dies liegt in diesem Fall an der Konstruktion des Problems, die die Heuristiken *minK*

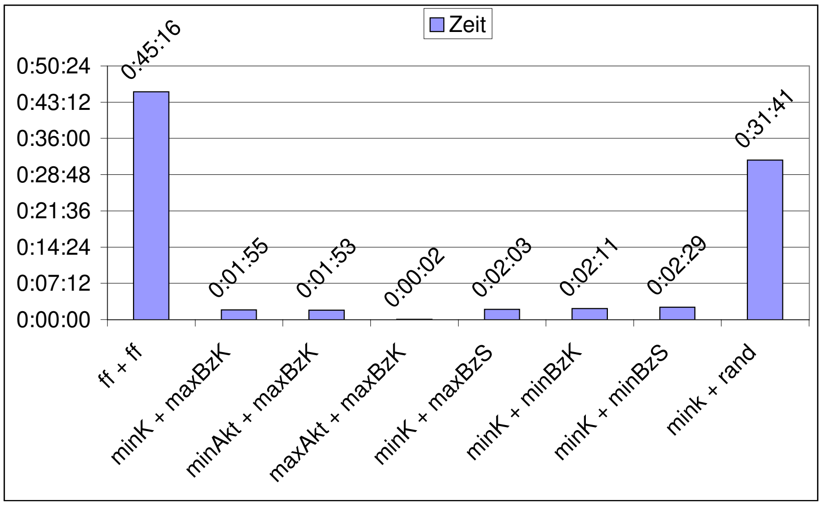


Abbildung 8.2: *Beste Lösung.* Suchzeit

und *minAkt* betrifft: die Kosten eines Teilnehmertyps steigen mit der Zahl seiner Fähigkeiten.

Auch die Kombination *(minK, maxBzS)* verhält sich ähnlich in der Optimierung wie *(minK, maxBzK)*. Tests haben ergeben, daß die relativen Zeitfenster zu klein sind, so daß häufig auf die Heuristik maxBzK ausgewichen wird. Beide Ähnlichkeiten werden ebenfalls durch die Suchdauer belegt.

Die Vorteile der genannten, speziellen Heuristiken fallen insbesondere im Vergleich zu der allgemeinen Heuristikkombination *(ff, ff)* auf. Letztere schafft es nicht, innerhalb einer Stunde eine gleichwertige Lösung zu berechnen. Ein weiteres Indiz für die Zielstrebigkeit der gennanten speziellen Heuristiken ist die geringe Anzahl von Lösungen, d.h schlechtere Lösungen werden durch Branch & Bound sofort ausgeschlossen.

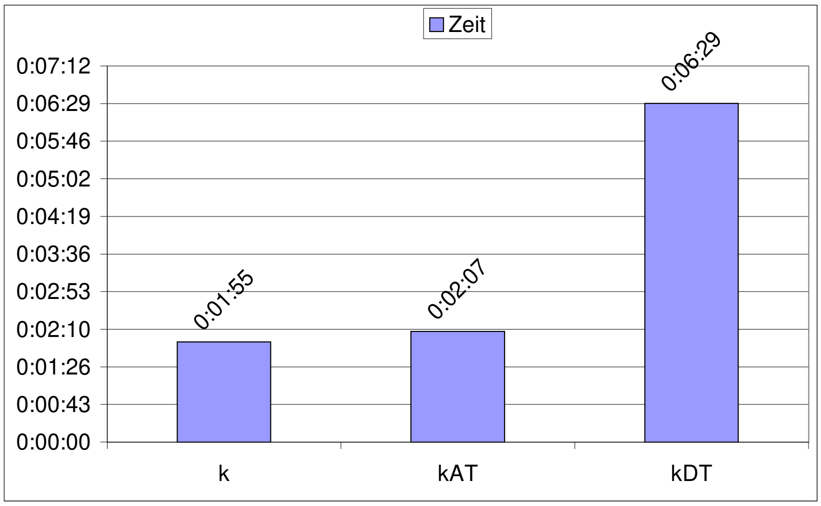
Als eine schlechte Kombination hat sich *(maxAkt, maxBzK)* erwiesen. Sie berechnet eine schlechte Gesamtlösung, die zudem nicht in der langen, verbliebenen Suchzeit verbessert werden kann. Das ist ein Indiz dafür, daß die Suche schon frühzeitig in einen schlechten Ast verzweigt ist. Hierfür gibt es zwei Gründe, die auf die Heuristik *maxAkt* zurückzuführen sind: Ressourcen mit vielen Fähigkeiten üben geringe Einschränkungen auf die Partition aus. Desweiteren haben Ressourcen mit vielen Fähigkeiten üblicherweise hohe Kosten.

Überraschend gute Ergebnisse berechnet die Kombination *(minK, minBzK)*. Diese sind nur geringfügig schlechter als die von *(minK, maxBzK)*. Dagegen schneidet die Kombination *(minK, minBzS)* schlecht ab. Beide berechnen ihre beste Lösung innerhalb zweieinhalb Minuten. Die verbliebene Zeit reicht beiden zur Verbesserung der Ergebnisse nicht aus. Die Vermutung liegt nahe, daß eine späte Zuweisung langandauernder Aktivitäten die Propagierung und letztendlich auch die Approximierung hemmt.

Der letzte Benchmark *(minK, rand)* zeigt, daß eine zufällige Zuteilung von Aktivitäten keinen Erfolg hat.

8.1.1.1 Kostenoptimierung

Die gleichzeitige Optimierung von Kosten und Takzeit weist große Unterschiede der Optimierungsfunktionen auf. Zwar berechnen die Funktionen *k*, *kAT* und *kDT* die gleichen Lösungen, jedoch benötigen sie dafür, wie aus der nachfolgenden Abbildung zu ersehen ist, unterschiedlich viel Zeit. Die Funktion *kAT* approximiert geringfügig langsamer als *kAT*. Dagegen dauert die Approximation *kDT* aufgrund der Betrachtung einer Disjunktion wesentlich länger. Die Zahl der Wahlpunkte von *kDT* ist um das Vierfache höher als bei *kAT*. Alle drei Benchmarks beziehen sich auf die Kombination *(minK, maxBzK)*.



8.1.2 Ressourcen mit vielen Fähigkeiten

Je mehr Teilnehmertypen mit vielen Fähigkeiten zur Verfügung stehen, desto schwieriger gestaltet sich die Suche. Insbesondere fällt anfangs eine schwächere Propagierung auf, die durch schwache Einschränkungen der Partition zustande kommt. Dadurch werden Teilnehmertypen mit vielen Fähigkeiten bevorzugt.

Die beste Lösung eines auf 10 Blöcke fixierten Benchmarks (siehe Benchmark A.3.4 und Abbildung 8.3) enthält den Teilnehmertyp *riesig*, obwohl bekannt ist, daß günstigere Lösungen auf diese Ressource verzichten. Es ist ebenfalls zu erkennen, daß Teilnehmertypen mit vielen Fähigkeiten die Zahl der Ressourcen reduzieren, nicht aber die Gesamtkosten.

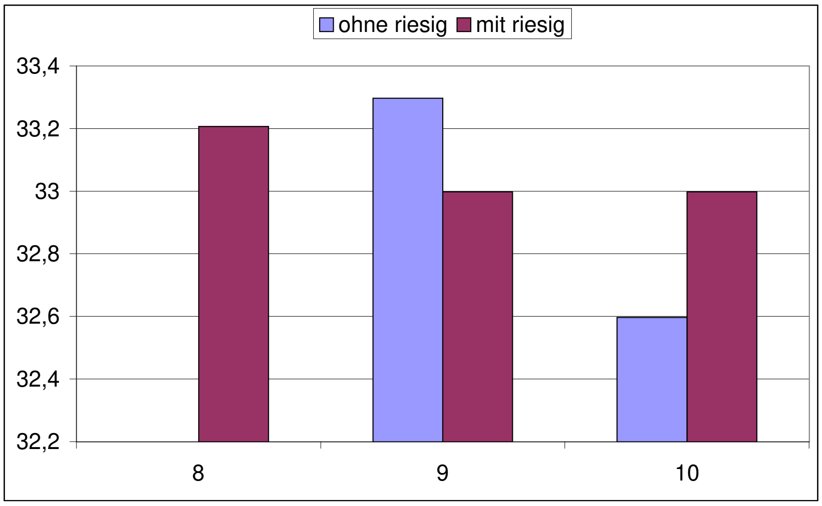


Abbildung 8.3: *Variable Blöcke.* Kosten für 8, 9 und 10 Blöcke; sowohl mit als auch ohne den Einsatz der Ressource *riesig*.

Der Einsatz des Teilnehmertyps *riesig* steht scheinbar im Widerspruch zur Heuristik *minK*, die die billigste Ressource wählt. Die Antwort lautet: Propagierung und Partition. Die Aktivitäten in einem Block bilden eine Teilmenge der Fähigkeiten der Ressource des jeweiligen Blocks (Filter 5.2.4 auf Seite 62). Die Fähigkeiten des Teilnehmertyps *riesig* umfassen fast alle Aktivitäten. Diese sind eine obere Schranke für alle Blöcke (siehe auch Abschnitt über Finite Sets auf Seite 37). Dadurch können nur wenige Werte für diese Blöcke ausgeschlossen werden; ergo liegt eine geringe Propagierung vor. Daraus folgt, daß die unteren Schranken der Blöcke ebenfalls nur wenige Veränderungen erfahren, was wiederum dazu führt, daß auch die Teilnehmertypen mit geringen Fähigkeiten einsetzbar sind. Aus diesen wählt die Heuristik die günstigste Ressource. Nach Beendigung der Propagierung bleibt für manche Blöcke nur noch der Teilnehmertyp *riesig* übrig.

Besser ist in diesem Schritt die Wahl der nächstteureren Ressource, was zu einer ähnlichen Lösung wie der optimalen führen kann. Die gleiche Problemstellung ohne die Ressource *riesig* führt zu wesentlich mehr Propagierung, dementsprechend auch zu besseren Lösungen.

Fazit ist, daß auch die Heuristik *minK* bei Ressourcen mit vielen Fähigkeiten nicht optimal funktioniert.

Parallele Suche. Die beste gefundene Lösung der parallelen Suche ist besser als die der einfachen Suche (siehe Benchmark A.3.5), was die bessere Effizient der parallelen Suche beweist. Doch auch diese Lösung beinhaltet den Teilnehmertyp *riesig*. Dies zeigt, wie schwierig Probleme werden, wenn Teilnehmertypen mit vielen Fähigkeiten, die durch geeignetere Typen ersetzt werden können, zugelassen werden. Allein der

Suchraum für die Zuweisung der Aktivitäten ist enorm groß.

8.1.3 Variable Blöcke

Es ist schwierig zu bestimmen, aus wievielen Blöcken die kostengünstigste Lösung besteht. Dies belegen die Benchmarks A.3.6 und A.3.7 auf Seite 107 und Abbildung 8.3.

Der erste Benchmark verzichtet auf den Teilnehmertyp *riesig*. Alle Lösungen bestehen aus mindestens neun Blöcken. Keine Lösung weist günstigere Kosten auf als die bisher beste, die aus zehn Blöcken besteht.

Dem zweiten Benchmark steht der Teilnehmertyp *riesig* zur Verfügung. Die erste Lösung besteht bereits aus acht Blöcken. Allerdings liegen die Kosten weit über denen der günstigsten Lösung. Daraus folgt, daß die günstigste Lösung nicht unbedingt eine minimale Anzahl von Blöcken hat. Tatsächlich ist es nicht möglich, im Vorfeld die Anzahl der Blöcke der optimalen Lösung zu bestimmen. Das gleiche gilt für die Anzahl der Ressourcen.

Weitere Tests zeigen, daß die Komplexität eines Problems mit steigender Blockanzahl wächst. Je mehr Blöcke vorliegen, desto mehr Zuweisungsmöglichkeiten sind für Aktivitäten gegeben.

8.2 Präzises Modell

Zunächst werden die Erweiterungen des präzisen Modells, wie *lokale Zeit* und *Werkzeuge*, getrennt betrachtet. Anschließend werden beide Erweiterungen zusammen mit dem vermischten Datenfluß analysiert.

8.2.1 Lokale Zeit

Es ist festzustellen (siehe Benchmark A.3.8 auf Seite 107), daß die Verwendung von lokalen Zeiten zu exakteren Ergebnisse führt. Gleichfalls fällt die Bestimmung der Organisation und Infrastruktur genauer aus. Das Ergebnis zeigt eine Lösung mit reduzierten Kosten auf. Dies liegt daran, daß günstige Teilnehmertypen mehr Aktivitäten zugewiesen bekommen.

Die zusätzlichen Informationen der lokalen Zeiten haben zu keiner Verschlechterung der Propagierung geführt.

8.2.2 Werkzeuge

Werkzeuge erhöhen die Komplexität des Problems. Die Ergebnisse dieser Untersuchung sind in Tabelle A.3.9 auf Seite 107 zu finden.

Der Einsatz von Werkzeugen und Werkzeugmagazinen führt zu neuen Lösungen. Ressourcen mit kleinen Magazinen erzwingen Lösungen, die ihre Aktivitäten auch auf teure Ressourcen verteilen. Daraus ist zu schließen, daß im wesentlichen die Problemstellung und damit der Lösungsraum verändert wird. Eine effizientere Suche ist mit Werkzeugen nicht ableitbar.

Sind die Werkzeugmagazine zu klein, oder ist die Zahl der Ersatzwerkzeuge zu groß, und zwar so, daß jede Kombination von Aktivitäten mehr Werkzeuge fordert als Magazinplätze zur Verfügung stehen, dann wird das System diese Inkonsistenz nicht entdecken. Stattdessen werden alle Aktivitätszuweisungen probiert. An dieser Stelle sind weitere Constraints notwendig.

Gleichfalls führen kleine Werkzeugmagazine zu Lösungen mit vielen Blöcken. Ein Anstieg der Blockanzahl erhöht die Zahl der Zuweisungsmöglichkeiten für Aktivitäten und steigert somit die Komplexität (siehe Abschnitt 8.1.3).

8.2.3 Vermischter Datenfluß

Aus den vorherigen Abschnitten geht hervor, daß die Erweiterungen keine effizientere Suche ermöglichen. Stattdessen verändern sie die Problemstellung und den Lösungsraum. Gleiches ist vom vermischten Datenfluß zu erwarten.

Ziel des vermischten Datenflusses ist nicht nur die gleichzeitige Ausführung verschiedener Prozeßdefinitionen, sondern auch die optimale Nutzung der Kapazitäten (Blocktaktzeiten). Eine gute Organisation spart weitere Ressourcen im Vergleich zum separaten Datenfluß. Benchmark A.3.10 listet die Ergebnisse auf.

Zunächst ist zu erkennen, daß der rein-vermischte Datenfluß kostengünstiger ist als der rein-vermischte. Diese Überlegung wurde schon in Beispiel 4.2.8 durchgerechnet, und ist mit der Anzahl der Teilnehmer zu belegen. Das macht in diesem Benchmark einen Unterschied von einem bis zwei Teilnehmern aus.

Ohne Werkzeugangaben, lokale Zeiten und differenzierte Kosten ist ein direkter Vergleich zwischen separatem und vermischtem Datenfluß möglich. Hierbei hat jeder Prozeß einen 50%igen Anteil. Die letzten beiden Zeilen des Benchmarks A.3.10 sind ein Test, bei dem alle Workflow Teilnehmer ein Werkzeugmagazin mit 100 Plätzen haben.

Die Kosten dieses Benchmarks betragen 31 394 000 Einheiten. Dies ist um 1 133 000 Einheiten günstiger als beim separaten Datenfluß. Eine Erklärung bietet die veränderte Infrastruktur:

Block	1	2	3	4	5	6	7	8	9	10
WFT	S	g	S	g	m	S	g	m	k	S
Anzahl	1	4	1	6	6	1	9	1	10	1

Zweimal wird im vermischten Datenfluß auf eine *große* Ressource verzichtet. Stattdesen werden zwei weitere, günstigere *medium* Ressourcen eingesetzt. Hier hat sich also die Verwendung freier Kapazitäten in Form von Restbearbeitungszeiten als nützlich erwiesen. Insgesamt benötigt der vermsichte Datenfluß mit 40 Teilnehmern einen weniger als der separate Datenfluß.

8.3 Slack

Die Benchmarks mit Slack Option (Kapitel 6.3.2) bleiben unverändert. Empirische Tests haben gezeigt, das die Summe der verbliebenen Bearbeitungszeiten stets größer als das kleinste, relative Zeitfenster ist. Während das Zeitfenster meist um die 60 Zeiteinheiten pendelt, beträgt die Zahl der nichtdeterminierten Variablen im Schnitt sieben. Das heißt, das kleinste, relative Zeitfenster ist kleiner als die Summe der Übergangszeiten nichtdeterminierter Variablen. Damit ist der Slack Algorithmus in diesem Szenario unbrauchbar.

8.4 Zusammenfassung

Die speziellen Workflow Heuristiken arbeiten effizienter als die allgemeinen Heuristiken. Insbesondere die Kombination *(minK, maxBzK)* bringt die besten und schnellsten Ergebnisse. Als Optimierungsfunktion eignet sich in erster Linie die Kostenfunktion *k*. Mit Einschränkungen auch die Funktion *kAT*. Dagegen ist die Optimierung mit *kDT* ungeeignet.

Der Einsatz von Teilnehmertypen mit vielen Fähigkeiten steigert die Komplexität. Dies gilt insbesondere, wenn diese Teilnehmertypen durch günstigere Teilnehmertypen mit weniger Fähigkeiten ersetzt werden können.

Auch die Bestimmung der Blockanzahl der günstigsten Lösung ist nicht trivial. Man weiß, daß eine solche Lösung wenige Blöcke beansprucht, aber nicht die wenigsten. Darüber hinaus steigt die Komplexität bei zunehmder Zahl von Blöcken.

Zwar ist die Suche mit parallelen Suchmaschinen enorm effizient im Vergleich zur einfachen Suche, doch auch hier zeigt sich die Komplexität der Aktivitätzuweisung.

Eine Erweiterung des Grundmodells um lokale Zeitangaben ergibt präzisere Lösungen. Dieser Modellierungsschritt führt weder zu einer besseren noch zur schlechteren Effizienz.

Anders ist es bei Anwendung von Werkzeugen und Werkzeugmagazinen. Knappe Magazine erhöhen die Komplexität deutlich. Der Einsatz von Werkzeugen ist nur im Zusammenhang mit Kostenoptimierung sinnvoll.

Die Berechnung von vermischten Datenflüssen ist effizient. Hierbei reduziert die reinvermischte Variante im Vergleich zum separaten Datenfluß verstärkt die Kosten.

Kapitel 9

Zusammenfassung

In dieser Arbeit wurde ein umfangreiches, mathematisches Modell zu einer neuen Klasse von Planungsproblemen vorgestellt. Es handelt sich um Optimierungsprobleme für flexible Prozesse mit kontinuierlicher Versorgung. Zur Lösung dieser Probleme wurde die Implementierung *Woop* in Mozart/Oz [23] unter Anwendung der Constraint-Technik entwickelt.

9.1 Verwandte Arbeiten

Das Anwendungsspektrum dieser Klasse von Planungsproblemen ist breit gefächert. In der Wirtschaft wird die Optimierung von Geschäftsprozessen als Business Reengineering bezeichnet. In der Industrie finden sich flexible Prozesse mit kontinuierlicher Versorgung im Bereich des Computer Integrated Manufacturing (CIM) und im Bereich der Fertigung [6]. Ähnlich zeitkritische und rechenaufwendige Prozesse sind in der Welt der Computer beispielsweise im vernetzten Workflow Management anzutreffen [13].

Trotz des breiten Anwendungsspektrums sind nur wenige verwandte Arbeiten bekannt. Die meisten basieren auf einfachen Modellen und haben nur ein Optimierungskriterium: Kosten oder Durchlaufzeit.

Scheduling. Im Operations Research und im Constraint Programming Bereich wurden Scheduling Probleme, insbesondere die Klasse der Shop Probleme (JobShop, FlowShop etc.) intensiv untersucht [2, 11, 17, 5, 10]. Shop Probleme haben jedoch ein einfaches Modell, und im Gegensatz zu dieser Arbeit besteht ihre Aufgabe in der Optimierung der Prozeßlogik, nicht aber der Organisation und Infrastruktur.

Höhere Petrinetze. Höhere Petrinetze erweitern klassische Petrinetze durch Attribute, Zeit- und Hierarchieangaben. Oberweis [14] hat eine Vorgehensweise entwickelt, die auf Verifizierung und Analyse bestehender Geschäftsprozesse basiert. Dies ist ein wesentlicher Unterschied zu dem generischen Ansatz dieser Arbeit. Oberweis kann somit nur kleine Optimierungen der Kosten punktuell durchführen.

ARIS Toolset. Ähnliches gilt für den kommerziellen Marktführer im Bereich Business Process Reengineering, IDS Scheer AG [1], die das ARIS Toolset [25, 26] vermarktet. Geschäftsprozesse werden durch Simulationen analysiert und gegebenfalls punktuell optimiert.

Workflow. Michael Gillmann erläutert in seiner Dissertation [13] den bisher ähnlichsten Ansatz. Zusätzlich zum Ablauf betrachtet er den Kontrollfluß, der Schleifen und Verzweigungen enthalten kann. Auf der Basis von Markovketten bestimmt er für jeden Geschäftsprozeß die Durchlaufzeit und für jeden Block den Durchsatz. Damit ist eine weitere Bestimmung der Anzahl von Teilnehmern möglich. Markovketten basieren jedoch auf Wahrscheinlichkeiten, so daß Gillmanns Modell keine exakten Lösungen angeben kann. Ferner existiert in [13] im Unterschied zu dieser Arbeit kein gerichteter Fluß, d.h. jedes Objekt kann von einem Teilnehmer an einen beliebigen Teilnehmer weitergereicht werden. Daraus folgt, daß alle Teilnehmer eines Typs in einem Block vereint sind. Außerdem gibt Gillmann die zu verwendeten Teilnehmertypen vor. Eleganter ist es, die geeigneten Typen zu ermitteln.

9.2 Kernpunkte der Arbeit

Aufgabenstellung. Für eine Menge von Abläufen soll die kostengünstigste Organisation und Infrastruktur bestimmt werden. Dabei darf nur auf Teilnehmertypen aus einer gestellten Menge zurückgegriffen werden. Teilnehmertypen haben Eigenschaften: Fähigkeiten, Kosten, Stabilität, lokale Zeiten etc. Die Fähigkeiten verschiedener Teilnehmertypen dürfen überlappen.

Zentrale Idee des Lösungsverfahrens. Die zentrale Idee ist die Partitionierung der Menge aller an den Abläufen beteiligten Aktivitäten in Blöcke. Jede Aktivität wird unter Beachtung der Vorrangsregeln einem Block zugewiesen. Die Partitionierung garantiert den gerichteten Datenfluß. Zusätzlich wird jedem Block ein Teilnehmertyp zugeordnet, der alle Aktivitäten des Blocks ausführen kann.

Formalisierung. Die Problemstellung wurde mathematisch abstrahiert und als ein Workflow Problem modelliert. Die Abstraktion beinhaltet ein Grundmodell, daß sukzessive zu einem präzisen Modell erweitert wurde. Die Formalisierung des Grundmodells erlaubt genauere, formale Untersuchungen des Problems. Beispielsweise war die Definition der verschiedenen Zeitfenster hilfreich bei der Untersuchung verschiedener Heuristiken oder des Slackalgorithmus. Das präzise Modell wurde um lokale Zeitangaben, Werkzeuge, differenzierte Kosten und um den vermischten Datenfluß erweitert.

Lösungstechnik. Als Lösungstechnik wurde die Constraint Programmierung gewählt, die eine direkte Umsetzung des mathematischen Modells ermöglicht. Lösungen werden korrekt, exakt und generisch aus partiellen Informationen berechnet. Das Constraint Modell ist effizient, skalierbar und approximiert schnell optimale Lösungen.

Heuristik. Die Heuristik ist dreischichtig aufgebaut: Blockanzahl, Workflow Teilnehmertypen und Aktivitäten. Die effizienteste, kostenorientierte heuristische Kombination ist *(minK, maxBzK)*. Die Heuristik *minK* wählt den Block, dem die wenigsten Teilnehmertypen zugewiesen werden können und teilt ihm den kostengünstigsten Teilnehmertyp zu. Die Heuristik *maxBzK* wählt die Aktivität mit der längsten Bearbeitungszeit – dies ergibt eine bestmögliche Propagierung – und weist sie dem Block mit dem günstigsten Teilnehmertyp zu.

Das Grundmodell ist so formuliert, daß Heuristiken sowohl auf Finite Domains als auch auf Finite Sets aufsetzen können. Für Finite Domain Heuristiken wurde eine Schnittstelle entwickelt, die in jedem Zerlegungsschritt den Zugriff auf die gesamte Root Variable erlaubt.

Suche. Die Suche verwendet die Branch & Bound Technik. Die beste Optimierung ist die Reduzierung von Kosten. Weitere Ideen zur Suche beschäftigten sich mit Slack Eigenschaften und mit Optimierungsfunktionen hinsichtlich der Anzahl bestplatzierter Aktivitäten. Beide Ideen haben sich als inpraktikabel erwiesen.

Präzises Modell. Die Erweiterung des Grundmodells um lokale Zeitangaben, Werkzeuge und differenzierte Kosten führte zu exakteren Ergebnissen. Das präzise Modell erforderte nur wenige Veränderungen im Grundmodell. Die Effizienz der Suche blieb bestehen. Eine verbesserte Propagierung aufgrund genauerer Informationen wurde nicht festgestellt.

Vermischter Datenfluß. Neben dem Modell für den separaten Datenfluß wurde ein leicht modifiziertes Modell für den vermischten Datenfluß angegeben. Dieser wurde weiterhin unterteilt in den rein-vermischten und den kombiniert-vermischten Datenfluß. Es wurde gezeigt, daß der kombiniert-vermischte Datenfluß keine besseren Lösungen liefert als der rein-vermischte Datenfluß. Zwar sind die Teilnehmertypen gleich, doch fällt deren Anzahl größer aus. Das Constraint Modell arbeitet auch mit dem vermischten Datenfluß effizient.

Interaktivität. Die Implementierung sieht eine Möglichkeit vor, den Anwender interaktiv in die Modellierung einzubeziehen. Über benutzerdefinierte Constraints kann der Anwender weitere partielle Informationen dem System mitteilen, und damit einen Teil des Lösungsraums vorgeben. Benutzerdefinierte Constraints haben sich als ein bewährtes Modellierungswerkzeug erwiesen. Desweiteren besteht die Möglichkeit, verschiedene Heuristiken zu testen.

9.2.1 Implementierung

Woop Frontend. Die Implementierung *Woop* ist ein umfangreicher Frontend, der als grafische Schnittstelle zwischen dem Anwender und den Suchmaschinen fungiert. Er gestattet die Wahl der Heuristik, der Suchmaschine und der Suchstrategie. Gefundene Lösungen werden protokolliert und grafisch aufbereitet. *Woop* ist in der Lage, Lösungen und Konfigurationen zu verwalten (Speichern, Laden, Export nach ASCII). Darüber hinaus besteht die Möglichkeit, von Hand berechnete Lösungen in den Frontend zu laden und zu verifizieren.

Dynamische Skriptgenerierung. Wichtigstes Merkmal ist die generische Erstellung des Suchskripts. Insbesondere ist die Kontrolle über Constraints und Heuristiken sehr nützlich. Einzelne Gruppen von Constraints können über ein Dialogfenster ein- bzw. ausgeschaltet werden. Sowohl für die Teilnehmer als auch für die Aktivitäten ist eine eigene Heuristik wählbar, so daß das Testen verschiedener Heuristikkombinationen möglich ist. Zusätzlich existiert die Möglichkeit, eigene Heuristiken auf der Basis vorgegebener, heuristischer Kriterien zu entwerfen. Außerdem ist eine Auswahl an Optimierungsfunktionen für die Suche nach der besten Lösung gegeben.

Woop Editor. Neben dem Frontend wurde ein Graph Editor entwickelt, der den Anwender beim Zeichnen von Graphen unterstützt. Vor dem Export in den *Woop* Frontend werden zwei Graphen automatisch erstellt: ein vereinigter Graph für den separaten Datenfluß und ein Graph für den vermischten Datenfluß. Zusätzlich wird die Richtigkeit der Graphen geprüft, d.h. alle Graphen werden verifiziert.

9.3 Offene Fragen

Das mathematische Modell ist Grundlage für die Formulierung der Problemstellung. Mit dessen Hilfe konnten viele Antworten, insbesondere im Bereich der Constraints und Heuristiken, gegeben werden. Trotz einer intensiven Auseinandersetzung mit der Problemstellung sind viele Fragen offen geblieben.

Ist ein Propagierer für die Partitionierung effizienter als die Modellierung mit reifizierten Constraints? Der zentrale Constraint dieser Aufgabenstelleung ist die Partitionierung und der gerichtete Datenfluß (siehe Seiten 26ff, 42 und 61). Die Implementierung verwendet bisher reifizierte Constraints. Die Frage ist, ob Projektoren (siehe hierzu die Dissertationen von Müller [20] und Würtz [34]) für einen eigenen Propagierer formuliert werden können. Wenn ja, gibt es eine Effizienzsteigerung?

Ist die Blockanzahl der besten Lösung auch ohne Suche bestimmbar? Es wurde gezeigt, daß eine Lösung mit der minimalen Blockanzahl nicht die beste sein muß. Können eventuell Schranken bzw. Constraints für die FD Variable, die die Blockanzahl repräsentiert, angegeben werden? Kann man weiterhin von der unteren Schranke der Blockanzahl auf die Anzahl von Teilnehmern schließen? Dies ist möglich, solange die Rüstzeit größer als die globale Übergangszeit ist.

Ab welcher Blockanzahl werden Lösungen schlechter? Intuitiv müßte man bei jedem neuen Block einen weiteren, günstigsten Teilnehmer hinzufügen. Werden die Gesamtkosten überschritten, wurde die maximale Anzahl von Blöcken gefunden.

Was ist die maximale Anzahl von Teilnehmertypen? Eine naive Antwort ist: es ist die Anzahl, die notwendig wird, um alle eigenen Aktivitäten auszuführen. Vorraussetzung hierfür ist, daß alle Aktivitäten in einem Block liegen. Sie können aber auf verschiedene Blöcke verteilt sein, womit weitere Rüstzeiten anfallen.

Welche Teilnehmertypen sind kein Bestandteil der besten Lösung? Aus den Evaluierungen für den komplexen Benchmark geht die Vermutung hervor, daß der Verzicht auf den Teilnehmertyp *riesig* zu wesentlich besseren Ergebnissen führt. Gibt es Kriterien, die den frühen Ausschluß dieser hochfähigen aber teuren Ressource rechtfertigen?

Sicherlich stehen diese Kriterien im Zusammenhang mit der Stabilität und den Kosten der Ressource sowie mit der Partitionierung.

9.3.1 Heuristik

Zwar approximiert die Heuristikkombination (*minK, maxBzK*) effizient, doch funktioniert sie nicht optimal. Im ungünstigen Fall würde sie die längste Aktivität einem billigen Teilnehmer zuweisen, während vorherige Aktivitäten, die in der Summe länger andauern, aufgrund der Ablaufspezifikation an einen teuren Teilnehmer verwiesen werden. Eine verbesserte Heuristik muß diesen Fall nach Möglichkeiten vermeiden.

Eine weitere Idee besteht darin, die Hierarchie der Heuristik um eine Stufe zu ergänzen. Die neue Stufe gibt die Anzahl der Ressourcen pro Block an. Nach jeder Teilnehmertypzuweisung findet eine aufsteigende Iteration über die Anzahl der Teilnehmer statt, d.h. eine vollständige Infrastruktur liegt vor, so daß die Gesamtkosten (ohne Werkzeuge) bekannt sind. Dabei gibt die Heuristik vor, daß immer der günstigste Teilnehmertyp zuerst iteriert wird, was zur Folge hat, daß die Kosten ebenfalls aufsteigend sind. Anschließend wird nach einer geeigneten Organisation gesucht. Wird eine passende Organisation gefunden, so liegt eine neue, bessere Lösung vor. Die nächstbeste Lösung muß sich aufgrund der Konstruktion dieser Heuristik im Punkt der Teilnehmertypzuweisung, nicht in der Anzahl, unterscheiden.

9.4 Ausblick

Das Grundmodell ist eine sehr gute Basis für weitere Untersuchungen des Problems. Insbesondere können mit dessen Hilfe die offenen Fragen möglicherweise beantwortet werden. Je mehr Eigenschaften daraus ableitbar sind, desto mehr Constraints verstärken die Propagierung. Weitere Ergänzungen sind im Bereich der benutzerdefinierten Constraints denkbar.

Auch im Bereich der Heuristik sind Verbesserungen wünschenswert. Trotz einer guten Approximation blieb der Beweis für die beste Lösung dieses Problems aus.

Aufgrund des praktischen Nutzens und der praktischen Vielfältigkeit des Problems besteht die Möglichkeit, *Woop* für das jeweilige Anwendungsspektrum zu starten. Dazu müssen sogenannte Sprachkataloge, ähnlich wie bei Java das Prinzip Localization/Internationalization, erstellt werden. Erweiterungen dieser Konstruktion sind im Bereich der Konfigurationsdateien notwendig. Hierzu müßte für jedes Einsatzgebiet ein entsprechender Scanner und Parser mit Gump [18] bereitgestellt werden.

Anhang A

Benchmark

A.1 Plattform

Alle Benchmarks der einfachen Suche fanden auf einem Athlon mit 1.2 GHz und 512 MB RAM statt. Nach einer Stunde wurden die Tests abgebrochen. Die Benchmarks der parallelen Suche wurden auf 7 Rechner verteilt: 4 Athlons mit 1.2 GHz und 512 MB RAM und 3 Athlons mit 0.7 GHz und 256 MB RAM.

Die Tests wurden mit dem Mozart 1.2.3 Programmiersystem durchgeführt.

A.2 Problemstellung: Komplexe, flexible Geschäftsprozesse

Dieses Beispielproblem besteht aus den Abläufen A.1 und A.2. Ersterer beinhaltet 52 Aktivitäten, letzterer hat eine Aktivität mehr.

Der Ressourcenpool besteht aus den in Tabelle A.2 aufgeführten Ressourcen. Ihre Fähigkeiten listet Tabelle A.1 auf. Dort sind neben den lokalen auch die globalen Zeiten angegeben. Darüber hinaus wird jeder Aktivität eine Menge von Werkzeugen zugeordnet.

Die geforderte, maximale Taktzeit beträgt 540 Zeiteinheiten bei einem Verlustfaktor von 5%. Die globale Übergangszeit beträgt 80 Zeiteinheiten und die Anzahl der Schwesterwerkzeuge ist auf eins festgesetzt.

Für den vermischten Datenfluß gilt ein Verhältnis für beide Abläufe von 50% zu 50%.

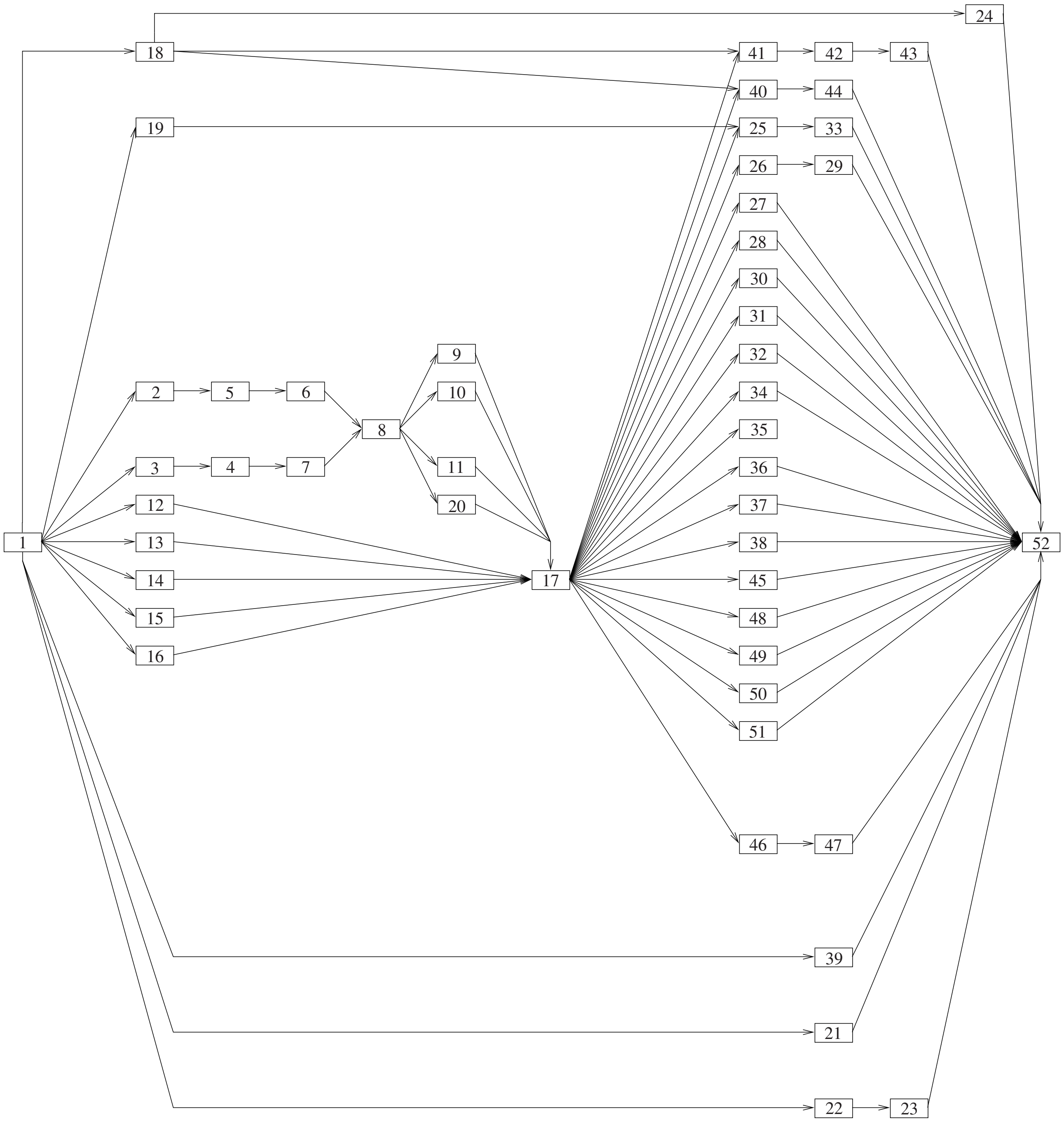


Abbildung A.1: Ablauf `abl1`

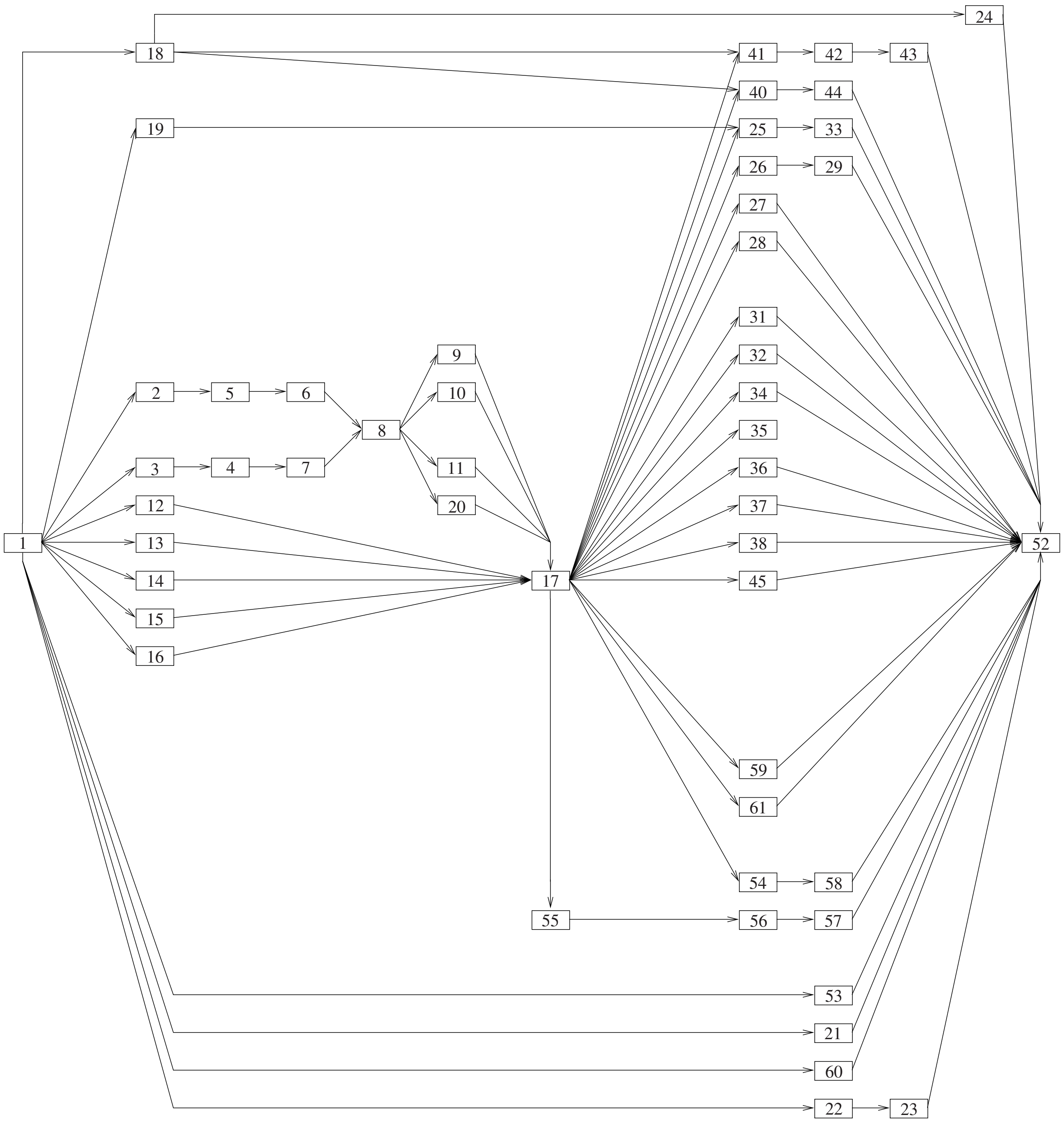


Abbildung A.2: Ablauf `abl2`

Akt.	Glob. Bz.	Wz.	5. Wft spezial	6. Wft klein	7. Wft medium	8. Wft groß	9. Wft riesig
1	0	1	Sonderaktivität auf 1. Wft				
2	147	2				155	147
3	143	3				155	143
4	112	4, 12				118	112
5	107	5				109	107
6	394	6				398	394
7	394	7, 14				386	394
8	0	8	Sonderaktivität auf 2. Wft				
9	72	9			60		72
10	864	10				856	864
11	864	11				868	864
12	690	12, 23				698	690
13	541	13		543	537	547	541
14	83	14		75	75	87	83
15	119	15		123	123	123	119
16	42	16		48	34	50	42
17	0	17	Sonderaktivität auf 3. Wft				
18	1056	18	1056		1068		
19	63	19		61	67	65	63
20	166	20		172	158	162	166
21	27	21		33	35	39	27
22	343	22		345	341	345	343
23	437	12, 19, 23		429	449	445	437
24	28	24		38	34	22	28
25	59	23, 25, 29		65	69	47	59
26	882	26		890	890	884	882
27	133	27		127	137	129	133
28	21	28		25	27	27	21
29	46	29		58	58	44	46
30	64	30		58	68	58	64
31	234	31		244	246	238	234
32	193	32, 36		185	195	181	193
33	166	33		170	168	178	166
34	100	34			112		100
35	574	35		580	578	580	574
36	163	36		153	165	169	163
37	178	37		186	186	170	178
38	168	38		164	164	170	168
39	726	39				718	726
40	141	40				131	141
41	141	41				133	141
42	449	42, 43		441	437	453	449
43	233	43				241	233
44	233	44, 47				229	233
45	18	45				24	18
46	51	46				63	51
47	114	47				122	114
48	51	48				63	51
49	391	49				387	391
50	56	50				62	56
51	92	51				96	92
52	0	52	Sonderaktivität auf 4. Wft				
53	1050	50, 53				1054	1050
54	51	54				63	51
55	236	55				242	236
56	53	14, 56				51	53
57	114	57				110	114
58	592	58				588	592
59	56	59				60	56
60	49	60		61	57	43	49
61	142	61				152	142

Tabelle A.1: Bearbeitungszeiten und Werkzeuge von Aktivitäten

	1. Wft (S)onder	2. Wft (S)onder	3. Wft (S)onder	4. Wft (S)onder	5. Wft (s)pezial	6. Wft (k)lein	7. Wft (m)edium	8. Wft (g)roß	9. Wft (r)iesig
Rüstzeit	0	1	1	0	200	200	200	200	200
Stabilität	95	95	95	95	95	95	95	95	95
Wz.-Magazin	10	10	10	10	10	10	20	35	40
Kosten	0	0	0	0	730000	75500	825000	951000	1020000

Tabelle A.2: Workflow Teilnehmertypen

A.3 Evaluierung

Alle aufgeführten Benchmarktabellen sind wie folgt zu lesen: die Heuristiken für Teilnehmertypen und Aktivitäten werden durch die Punkte *Heur. WFT* und *Heur. Akt.* angegeben. Mit *Opt.* ist die Optimierungsfunktion gemeint. Der Punkt *kosten* gibt den errechneten Invest und *Takt* die effektive Taktzeit an, wohingegen die Anzahl der Teilnehmer in der Spalte *Anz. WFT* abzulesen ist. Die Lösungen wurden in der Zeitspanne *Zeit* gefunden, wobei insgesamt x *Wahl punkte* und y *Fehlschläge* aufgetreten sind. Wird nach der optimalen Lösung gesucht, so gibt *Anz. Lös.* die Anzahl der zuvor gefundenen Lösungen an.

Die Abkürzungen der Heuristiknamen richten sich nach Tabelle 6.1 auf Seite 70 und 6.2 auf Seite 71.

Die Benchmarks A.3.1 A.3.2 und A.3.3 testen verschiedene Heuristiken und geben jeweils die erste Lösung, die beste Lösung nach einer Minute und die beste Lösung nach einer Stunde an. Alle nachfolgenden Tabellen enthalten sowohl die erste als auch die beste Lösung nach einer Stunde. Hierbei gehören zwei Zeilen zu einem Test. Die erste Zeile steht für die erste gefundene Lösung und die zweite Zeile für die beste gefundene Lösung.

A.3.1 Grundmodell

Die ersten drei Benchmarks testen verschiedene Heuristiken, um die Beste zu bestimmen. Die Tests gelten für das Grundmodell. Die Zahl der Blöcke ist auf 10 fixiert. Der Teilnehmertyp *riesig* steht nicht zur Auswahl.

Benchmark A.3.1 (Erste Lösung). Test verschiedener Heuristiken. Gesucht wird eine erste Lösung.

Heur. WFT	Heur. Akt.	Kosten	Takt	Anz. WFT	Zeit	Wahl-punkte	Fehl-schläge
ff	ff	37214000	518	44	00:01	41	10
minK	maxBzK	35002000	512	44	00:00	59	32
minAkt	maxBzK	35002000	512	44	00:00	60	33
maxAkt	maxBzK	35772000	517	44	00:01	59	12
minK	maxBzS	34247000	535	43	00:01	52	22
minK	minBzK	33492000	538	42	00:01	45	16
minK	minBzS	35027000	525	44	00:00	42	10
minK	rand	37395000	510	45	00:01	41	10

Benchmark A.3.2 (Beste Lösung nach einer Minute). Test verschiedener Heuristiken. Gesucht wird die beste Lösung nach einer Minute.

Heur. WFT	Heur. Akt.	Kosten	Takt	Anz. WFT	Zeit	Wahl-punkte	Fehl-schläge
ff	ff	33381000	538	41	00:22	6895	6698
minK	maxBzK	32667000	539	41	00:01	111	60
minAkt	maxBzK	32667000	539	41	00:01	112	61
maxAkt	maxBzK	33297000	538	41	00:01	571	490
minK	maxBzS	32667000	538	41	00:01	94	42
minK	minBzK	32877000	540	41	00:50	21517	21464
minK	minBzS	33562000	540	42	00:01	252	190
minK	rand	37395000	540	41	00:25	7030	6900

Benchmark A.3.3 (Beste Lösung). Test verschiedener Heuristiken und Optimierungsfunktionen. Gesucht wird die beste Lösung nach einer Stunde. Hierbei richten sich die Abkürzungen der Optimierungsfunktionen nach den Angaben in Kapitel 6.3

Heur. WFT	Heur. Akt.	Opt.	Kosten	Takt	Anz. WFT	Zeit	Wahl-punkte	Fehl-schläge	Anz. Lös.
ff	ff	k	32793000	538	41	45:16	803930	803653	39
minK	maxBzK	k	32597000	540	41	01:55	53500	53430	6
ff	ff	kAT	32793000	538	41	46:23	826051	825698	54
minK	maxBzK	kAT	32597000	537	41	02:07	56364	56202	25
minAkt	maxBzK	k	32597000	540	41	01:53	53501	52431	6
maxAkt	maxBzK	k	33297000	538	41	00:02	572	490	4
minK	maxBzK	kDT	32597000	537	41	06:29	203650	203482	25
minK	maxBzS	k	32597000	538	41	02:03	53486	53412	4
minK	minBzK	k	32667000	540	41	02:11	46783	46709	6
minK	minBzS	k	33422000	536	42	02:29	63243	63158	8
minK	rand	k	33129000	540	41	31:41	673085	672918	20

Benchmark A.3.4 (WFTs mit vielen Fähigkeiten). In diesem Benchmark wird zusätzlich der Teilnehmerty *riesig* erlaubt. Die Zahl der Blöcke ist auf 10 fixiert.

Heur. WFT	Heur. Akt.	Opt.	Kosten	Takt	Anz. WFT	Zeit	Wahl-punkte	Fehl-schläge	Anz. Lös.
minK	maxBzK	k	33753000	532	40	00:00	29	2	
			32998000	540	39	00:00	40	6	2
ff	ff	k	36285000	532	42	00:01	31	2	
			33264000	539	39	53:13	887097	886918	21

Benchmark A.3.5 (Parallele Suche). In diesem Benchmark wird zusätzlich der Teilnehmerty *riesig* erlaubt. Die Zahl der Blöcke ist auf 10 fixiert.

Heur. WFT	Heur. Akt.	Opt.	Kosten	Takt	Anz. WFT	Zeit	Wahl-punkte	Fehl-schläge	Anz. Lös.
minK	maxBzK	k	33538000	537	40	00:04			
			32816000	540	39	00:05			4
ff	ff	k	36285000	532	42	00:03			
			33264000	539	39	48:15			26

Benchmark A.3.6 (Variable Blöcke). Hier wird bei aufsteigender Blockanzahl die beste Lösung gesucht. Begonnen wird mit einem Block. Auf den Teilnehmertyp *riesig* wird verzichtet.

Heur. WFT	Heur. Akt.	Opt.	Kosten	Takt	Anz. WFT	Anz. Blöcke	Zeit	Wahl-punkte	Fehl-schläge	Anz. Lös.
minK	maxBzK	k	34947000	523	43	9	00:00	59	33	
			33297000	540	41	9	00:01	91	50	3
ff	ff	k	36459000	529	43	9	00:00	51	20	
			33549000	538	41	9	23:01	575144	574996	15

Benchmark A.3.7 (Variable Blöcke). Hier wird bei aufsteigender Blockanzahl die beste Lösung gesucht. Begonnen wird mit einem Block. Teilnehmertyp *riesig* ist erlaubt.

Heur. WFT	Heur. Akt.	Opt.	Kosten	Takt	Anz. WFT	Anz. Blöcke	Zeit	Wahl-punkte	Fehl-schläge	Anz. Lös.
minK	maxBzK	k	3376800	532	39	8	00:00	37	9	
			32998000	537	39	9	06:37	154535	154468	3
ff	ff	k	34971000	532	40	8	00:00	38	9	
			33207000	540	38	8	59:50	1651250	1651156	9

A.3.2 Präzises Modell

Die nachfolgenden Benchmarks werden um einzelne Punkte des präzisen Modells erweitert: lokale Zeiten, Werkzeuge bzw. Werkzeugmagazine und vermischter Datenfluß.

Benchmark A.3.8 (Lokale Zeiten). Dieser Test ist wie der für das Grundmodell, jedoch werden lokale statt globale Zeitangaben verwendet.

Heur. WFT	Heur. Akt.	Opt.	Kosten	Takt	Anz. WFT	Zeit	Wahl-punkte	Fehl-schläge	Anz. Lös.
minK	maxBzK	k	33100000	537	42	00:00	40	10	
			32345000	540	41	00:00	53	7	2

Benchmark A.3.9 (Werkzeuge). Dieser Benchmark ist wie der für das Grundmodell, jedoch mit zusätzlichen Werkzeugangaben: Werkzeugmagazine und Beschreibung der Aktivitäten durch Werkzeuge.

Heur. WFT	Heur. Akt.	Opt.	Kosten	Takt	Anz. WFT	Zeit	Wahl-punkte	Fehl-	Anz. Lös.
minK	maxBzK	k	36150000	535	44	00:01	113	80	
			33549000	538	41	06:50	177604	177541	4

Benchmark A.3.10 (Vermischter Datenfluß). Der Benchmark testet den vermischten Datenfluß. Hierbei steht *rv* für den reinen und *kv* für den kombinierten Datenfluß. Dieser Test enthält lokale Zeitangaben und Werkzeugbeschreibungen. Zusätzlich werden Kosten differenziert. Hierbei betragen die Blockkosten 230000 und die Werkzeugkosten 500 Geldeinheiten pro Werkzeug.

Der letzte Benchmark (letzten beiden Zeilen) stehen für einen Test, in dem alle Werkzeugmagazine 100 Plätze haben und die differenzierten Kosten wegfallen. Hiermit soll der vermischte Datenfluß mit dem separaten verglichen werden.

Dat. Fluß	**Heur. WFT**	**Heur. Akt.**	**Opt.**	**Kosten**	**Takt**	**Anz. WFT**	**Zeit**	**Wahl-punkte**	**Fehl-schläge**	**Anz. Lös.**
rv	minK	maxBzK	k	36583000	538	42	00:10	190	136	
				35757500	537	41	38:51	96697	96611	4
kv	minK	maxBzK	k	38485000	540	44	00:09	141	108	
				367085000	537	42	01:09	32118	32061	4
rv	minK	maxBzK	k	32149000	536	41	00:02	62	11	
				31394000	540	40	00:04	117	56	2

Literaturverzeichnis

[1] IDS Scheer AG. www.ids-scheer.de.

[2] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[3] Baptiste, Le Pape, and Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers, 2001.

[4] P. Baptiste, C. Le Pape, and W Nuijten. Incorporating efficient Operations Research algorithms in constraint-based scheduling. In *1st Joint Workshop on Artificial Intelligence and Operational Research*, 1995.

[5] Martin Bartusch. *Optimierung von Netzplänen mit Anordnungsbeziehungen bei knappen Betriebsmitteln*. PhD thesis, Universität Passau, Fakultät für Mathematik und Informatik, 1983.

[6] S. Bussmann and K. Schild. An agent-based approach to the control of flexible production systems. In *Proc. of the 8th IEEE Int. Conf. on Emergent Technologies and Factory Automation (ETFA 2001)*, volume 2, pages 481–488, 2001.

[7] J. Carlier and E. Pinson. An Algorithm for Solving the Job-Shop Problem. *Management Science*, 35(2):164–176, 1989.

[8] Yves Caseau and François Laburthe. Improved CLP scheduling with task intervals. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 369–383, Santa Margherita Ligure, Italy, 1994. The MIT Press.

[9] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[10] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):74–94, January-March 1990.

[11] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP Completeness*. W.H. Freeman & Company, 1979.

[12] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[13] Michael H. Gillmann. *Konfiguration verteilter Workflow-Management-Systeme mit Leistungsgarantien*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.

[14] Volker Guth, Andreas Oberweis, and Torsten Zimmer. Zeit- und Kostenanalyse von Geschäftsprozessen mit höheren Petri-Netzen. In *Fachtagung: Workflow-Management in Geschäftsprozessen im Trend 2000*, pages 99–110, 1997.

[15] Martin Henz and Tobias Müller. An Overview of Finite Domain Constraint Programming. In *Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies*, Singapore, 2000.

[16] Martin Homik. Woop: Workflow Optimizer (Software). Erhältlich unter http://www.ps.uni-sb.de/oz/dipl/.

[17] A. Jain and S. Meeran. A state-of-the-art review of job-shop scheduling techniques, 1998. Department of Applied Physics, Electronic and Mechanical Engineering, University of Dundee.

[18] Leif Kornstaedt. Gump: A Frontend-End Generator for Oz. Erhältlich unter http://www.mozart-oz.org/documentation/gump/index.html.

[19] F. Leymann and D. Roller. *Production Workflow: Concepts and Technologies*. Prentice-Hall, 2000.

[20] Tobias Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.

[21] Tobias Müller. Problem Solving with Finite Set Constraints in Oz. A Tutorial., 2001. www.mozart-oz.org.

[22] Tobias Müller and Martin Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, September 1997.

[23] Mozart Consortium. The Mozart programming system, 1999. Erhältlich unter www.mozart-oz.org.

[24] William Older and André Vellino. Constraint arithmetic on real intervals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 175–196. The MIT Press, 1993.

[25] August-Wilhelm Scheer. *Wirtschaftsinformatik. Studienausgabe. Referenzmodelle für industrielle Geschäftsprozesse*. Springer, 1998.

[26] August-Wilhelm Scheer. *ARIS, Modellierungsmethoden, Metamodelle, Anwendungen*. Springer, 2001.

[27] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.

[28] Christian Schulte. Parallel search made simple. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, number TRA9/00, pages 41–57, 55 Science Drive 2, Singapore 117599, September 2000.

[29] Christian Schulte. *Programming Constraint Services*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000.

[30] Christian Schulte and Gert Smolka. Finite Domain Constraint Programming in Oz. A Tutorial., 2001. www.mozart-oz.org.

[31] Gert Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[32] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Programming Logic Series. The MIT Press, 1989.

[33] WfMC. Workflow Management Coalition. www.wfmc.org.

[34] Jörg Würtz. *Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz*. Dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 1998.

Index